Recording Actor State in Scientific Workflows*

Ian Wootten, Omer Rana and Shrija Rajbhandari

School of Computer Science, Cardiff University, UK

Abstract. The process which leads to a particular data item, or its provenance, may be documented in a number of ways. The recording of actor state assertions – essentially data that a client or service actor may assert about itself regarding an interaction, is evaluated as a critical provenance component within a service-oriented architecture. Actor state data can be combined with assertions of interaction to enable better reasoning within a provenance system. The types of data that may be recorded as actor state are subjective, and dependent on the nature of the application and the eventual use that is likely to be made of this data. A registry system that allows monitoring tools to be related to user needs is described with reference to an application scenario.

1 Introduction

The provenance of a piece of data is the process that led to that piece of data [1, 2]. Typically, with service based projects, concern is primarily held with documenting the interaction between clients and services (actors) which were involved in a particular process as a means of capturing data provenance. Groth *et al.* [1] outline that for some data, its provenance is represented by some suitable documentation of process which led to it. This documentation includes in part the internal states of actors within the context of a particular interaction. Here, we make clear our definition of actor state data:

Actor State Data: That information regarding the state of an actor in the context of a specific interaction. A single assertion of actor state may concern the internal flow of data involved in interaction results within an actor, or the hosting environment state at a particular point in time. Assertions of actor state can only be recorded by the actor whom the data is about and not by a workflow enactment engine. An actor must therefore explicitly decide to make available such information to third parties. Our focus within this paper is primarily on how such actor state information can enhance documentation of interaction.

To capture the documentation of interaction between actors involved in a particular process, *interaction assertions* may also be used. Such assertions specify which actors are involved and the messages exchanged between them. Interaction assertions may be recorded by a workflow enactment engine (by copying

* This research is funded in part by EPSRC PASOA project GR/S67623/01.

all messages exchanged between actors in a workflow), or it may be explicitly recorded by the actors themselves.

The provenance of data is concerned with how we arrive at a particular data item, and assertions of actor state provide valuable information on how a particular actor state (for an involved actor) has been reached during the creation of that data item. This is achieved through the documentation of transformations made upon the input data within an actor, and the state the system (upon which an actor is hosted) was in when those transformations were made. Through capturing assertions regarding the transformations on input data we are able to determine what functionality an actor was invoking, i.e. what an actor was doing. Capturing the state of the system also records the context under which an actor was operating. Exposing this context allows insight into what conditions were set which could affect the overall data result.

Actor state assertion recording differs from interaction assertion recording – which may be recorded by a third party. All assertions concerning actor state become unverifiable if made by a third party, and as such the actor is the only party able to assert its own state.

Using actor state data it is possible to evaluate the behavior of an actor over the past and make predictions on its likely future behavior. Coupled with interaction assertions, such data may be used to evaluate whether a particular actor is the cause of an error or inaccurate result within a workflow instance. It also allows a better understanding of the performance patterns observed upon an actor, through using interaction assertions to determine what was being done when a particular behaviour pattern occurred. Within the context of a SOA, the description of internal flow of data within an actor would also constitute actor state data. An actor may make public the internal functions which were performed on the input data to obtain output data. From these functions, it is possible to construct a directed acyclic graph (DAG) detailing the process performed internally within an actor.

We attempt to develop a customisable architecture for the recording of actor state assertions. Familiarity with the concepts of Web Services and SOAs is assumed. The rest of this paper is organized as follows: section 2 provides requirements for an architecture which supports actor state assertion recording. Section 3 contains the types of data which may be recorded by an actor, and ways in which we may categorize such data. Section 4 presents an architecture to satisfy the requirements identified previously and an evaluation of a prototype system based on the architecture is provided. A summary of related work is given in section 5 and our concluding remarks are given in section 6.

2 Requirements

A provenance recording system must address a number of requirements, especially when used in the context of a SOA. The requirements for the storage of provenance information within a SOA have been highlighted in [1–3]. Here we identify requirements which influenced our prototype design:

An actor state assertion system should record the internal data flow within an actor in a verifiable, reproducible manner: It is possible to identify the activities undertaken by an actor as a DAG – which describes a series of transformations performed on some input data. Such graphs allow identification of how a particular data item was produced when followed in reverse. Hence, an actor that receives input `X` could transform it through a series of functions $f_i()$, eventually leading to an output `Y`. This can be achieved through instrumentation of that actor with relevant recording hooks describing such transformations and subsequent collection of this information by a provenance system.

An actor state assertion system should record in a non-repudiable manner any data generated by an actor: It is necessary to assume that information collected from any monitoring sources is accurate and provides a true reflection of the state of the actor. As sources are co-located with an actor this assumption should usually be satisfied (although the recording accuracy may differ between sources).

An actor state assertion system should be scalable, general and customisable: As previously stated in section 1, an actor needs to be viewed as a complete entity for actor state assertion, i.e. an actor asserts its own state. There exist a variety of mechanisms for capturing state information, and as such the available tools across potential applications differ significantly. It is therefore necessary for an actor state assertion system to able to be customised to cope with the variety of information sources and the platforms upon which it will be hosted. A pertinent question here relates to what data needs to be recorded and at what frequency. Once again, both of questions can only be answered when we identify such a systems' application domain.

3 Actor State Assertion Categories

Actor state assertion categories describe the types of actor state data that may be recorded. For each description, we use the term *node* to describe the system on which an actor is hosted, and *lifetime* to refer to the length of time for which the actor (client or service) is available. Persistent actors may be classified as having an infinite lifetime.

Static: That data which does not change throughout the lifetime of an actor. As a result, static data need only be recorded once during process execution. Such data items have been previously investigated, and include: (i) Per-Node: node identity, operating system, etc.; (ii) Per-Actor: actor identity, name, owner, version, capability, etc. Such information is similar to that published by an actor in a registry service in a SOA.

Dynamic: That data which may change during the lifetime of an actor. It is therefore necessary to record this data at periodic intervals over the lifetime of an actor. We assume that actors within our architecture do not maintain a persistent state. Such data items may include: (i) Per-Node: memory usage, network traffic,

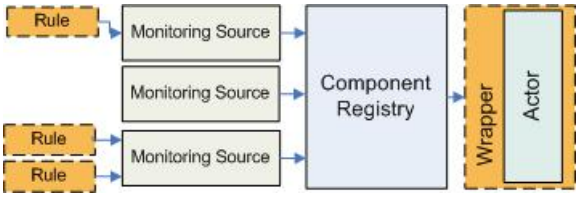

Fig. 1. Component Registry

etc. Such information needs to be recorded by the platform hosting the actor, and may be made available on demand. The accuracy of such dynamic data is dependent on the type of measurement tools being used; (ii) Per-Actor: service execution time, uptime, availability, etc. Such dynamic data is usually derived from other, less complex recorded metrics.

4 Conceptual Architecture

A registry based architecture for recording actor state is presented motivated by the requirements outlined in section 2. An end user may indicate which data is likely to be most significant to them – generally via the use of a configuration file. The basic architecture for a registry is given in figure 1. A component registry is co-located with an actor and holds details of the monitoring sources which are available on the platform hosting the actor. We consider the simplest case of one actor per platform in the first instance. The registry contains a description of interfaces through which monitoring sources may be contacted, and a mechanism to specify the time at which such requests may be scheduled.

A **monitoring source** may be a provider of a single piece of information or a number of metrics, and therefore a way of distinguishing the relevant useful information returned from the source becomes necessary. Within the registry, a number of **rules** may be associated with a single monitoring source, and specify the recorded actor state metrics desired during an actors' lifetime. Two types of rules exist within our architecture: $R_A(e)$ are **runtime rules** – and may be triggered to be executed by an external or actor administrator generated event e, e.g. a service invocation request. Such rules are immediately scheduled to execute when an event of type e is detected by an actor. This provides functionality to an end user who may only want to record actor state whilst a service is being invoked. $R_B(t_1, t_2)$ are **scheduled rules** which execute between a time interval $(t_2 - t_1)$ – where t_i is based on the actor clock. Using R_B it is possible to record the state of an actor outside the context of any particular interaction. Rules of type R_B must be defined and managed by an actor administrator, and cannot be accessed via a third party. For an actor that is long running, t_1 may correspond to the time when the actor was started, and t_2 set to a large value. The result produced as a consequence of running rules of type R_A and R_B may be: (i) raw data – in this case monitored data over the particular period in question is returned as an array of values. The data corresponds to

the raw data from the monitoring tool being used; (ii) interval data – in this case the output from the monitoring tool is sampled at periodic intervals defined by an actor administrator. An array of values is returned, corresponding to a value at each sample point; (iii) aggregate data – in this case only a single value for a particular metric is returned. Such a value may correspond to either the min, max or average over the particular period. An end user must therefore register their rules with the actor administrator if they require particular actor state information to be monitored. The presence of a registry allows re-use of rules between end users. Due to the tree-like hierarchy of rule-monitor-registry associations, the configuration of a registry is achieved through an XML file.

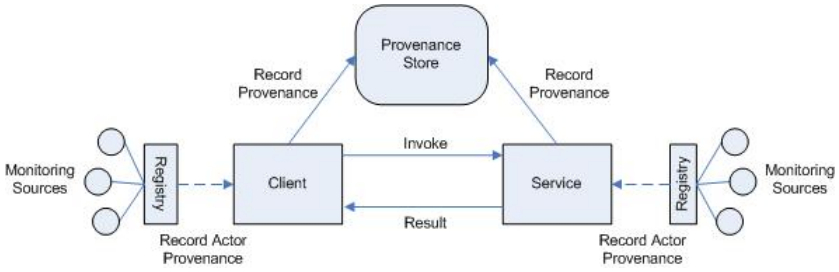

Fig. 2. Actor State Recording within a SOA using PReP [3]

Figure 2 shows how it is possible to capture both interaction and actor assertions within a SOA. The figure illustrates how actor assertions may be related to interaction assertions by extending the data that is submitted via the Provenance Recording Protocol (PReP) [3] to an external Provenance Store (a public and possibly remote repository). Both the client and service within such an architecture would have independent registries, containing references to locally available monitoring sources. Every time an interaction occurs between a client and service, each would submit their view of their interaction to a mutually agreed upon store.

Using Actor State Assertions: in a Web Service based workflow, a scientist does not have direct access to the system hosting the service actors. Using actor state information, such a user can make inferences about how or why a workflow has performed in a particular way. For example, consider a provenance-enabled workflow comprising of two services a and b, the workflow is executed twice to yield two sets of interaction assertions i_1 and i_2 in a Provenance Store. It also asserts actor state records a_1, a_2 and b_1, b_2 for each service actor. Both executions yield the same results, but the time of execution differ significantly. On inspection of i_1 and i_2, the experimenter has no means of determining the source of such a problem, due to only the data which has passed between a and b being recorded – as both interact in the same manner. Inspection of the actor's state record a_2 reveals performance metrics indicating high usage of a's resources

whilst being invoked. While b's performance shows no difference between b_1 and b_2, it is possible for the scientist to conclude that a was the most likely source of such a discrepancy. If the DAG used within an actor is also known, this can be used in conjunction with the recorded data to locate the function within an actor which may have been the source of a particular problem.

4.1 Implementation

A prototype of the architecture described in section 4 has been realised, and actor state recorded using an SOA-based approach. The Ganglia system (http://ganglia.sourceforge.net) has been used as a monitoring tool for obtaining metrics relevant to node state and recorded during service execution. Rules are described within our registry, along with information about how often they are to be executed and which monitoring sources they are associated with through configuration files expressed in XML. An actor state is recorded using rules associated with the registered monitoring source to a local provenance store. A coordinator process is responsible for checking which rules are valid at any given time.

In figure 3 we describe a rule written in XQuery which returns results from a Ganglia XML document. Within the query, `$ganglia:doc` refers to the document we are querying, which is bound to the XQuery at execution time. Using this description, we return the integer value (indicated in figure 3 as *VAL*) of the number of bytes in per second (bytes_in) (KB/s), packets in per second (pkts_in) and maximum transmission rate of the network (mtu). These values are then operated upon as indicated in figure 3 to determine the current (network) throughput of this actor.

```
let $N := data($ganglia:doc//METRIC[NAME="bytes_in"]/VAL)
let $X := data($ganglia:doc//METRIC[NAME="pkts_in"]/VAL)
let $R := data($ganglia:doc//METRIC[NAME="mtu"]/VAL)
let $bpp := $N div $X          (: Calculate number of Bytes Per Packet :)
let $tp := ($bpp*$N) div $R

return <metrics><throughput>{$tp}</throughput></metrics>
                               (: Return our result :)
```

Fig. 3. Example XQuery Rule Implementation

4.2 Evaluation

For evaluation, the registry prototype is used to record assertions regarding actor state during invocation of a data modeling service [4]. The modeller has a number of data processing techniques and neural network and statistical models which take incoming data sets from a client and generate models based upon them. There are a number of modeling algorithms exposed which vary the accuracy

of the model produced, possibly at the expense of incurring a greater computation time. Our experiments use Quantitative Structure-Activity Relationship (QSAR) to attempt to correlate biological activity to chemical compound structure described in the data set sent to it.

All experiments were conducted on a Ubuntu Linux System – AMD processor running at 1.83GHz with 512Mb of RAM. Both service and client actors are located on the same system, and for each experiment the service actor is invoked 100 times with input data sets of varying sizes, with the length of time of invocation recorded. Experiments are conducted for rules scheduled to be recorded simultaneously (at 1000ms intervals). A Ganglia monitoring daemon is installed making system metrics available as XML documents. This daemon is described as the monitoring source within the registry. The registry contains configuration files to enable scheduling of rules for runtime invocation, with each rule reflecting a single metric available through Ganglia. The results of rule execution are recorded to a local file system.

For our preliminarily benchmark, we note that the time taken to invoke the service when no assertions are made, upon a 34KB data set is approximately 1404ms. On recording rules scheduled with intervals of 1000ms, it is noted that the trend for overall time for execution of the service increases on addition of each rule, reaching a maximum of 38062ms for 5 rules. We can see by these results, that while our prototype is able to record actor state assertions, it incurs a significant overhead against a non-asserting actor operating on the same data set. Making the comparison against our initial requirements, whilst the prototype has been produced in a general and customisable manner, actors where large amounts of state data may be produced will evidently suffer from a performance degradation, especially where large rule sets are constructed. Further work therefore is necessary to modify our system to enable it to be scalable to such situations, such as the use of a cache for monitoring source data and its asynchronous capture.

	Size of Data Set (KB)									
No Of Rules	34	68	102	136	170	204	238	272	306	340
0	1404	3338	5604	8795	10382	15223	14422	23309	18114	24613
1	1752	3863	6233	8574	11220	13383	17154	18142	21574	26046
5	2416	5680	7686	15134	15826	15979	23082	28531	29919	38062

Table 1. Average Time to Complete Service Invocation

5 Related Work

In SOAs there are a number of requirements necessary for the capture of the provenance of interactions [1–3]. The PASOA project (http://www.pasoa.org/) has highlighted a method of representing assertions made about processes by the actors involved through use of a p-assertion [1], suggesting 3 different p-assertion types (Interaction, Relationship and Actor State). The p-assertion presents a

possible manner in which to represent assertions in our local provenance store and is already being used within other projects. At the German Aerospace Center for instance, p-assertions are being used in order to capture actor state elements such as computation completion states (crashed, interrupted etc) and workflow configuration parameters in the simulation of complex flight manoeuvres [5]. Such capture achieves a confidence in simulation results which a non actor state recording mechanism may not.

A number of common items that may be part of actor state across application domains have also been investigated by the EU Provenance project (http://www.gridprovenance.org), which were derived from the GLUE Information Model [6], though due to their application dependance, representation of them has not been specified. Our registry architecture attempts to provide a method whereby actor state assertion capture is formalised, but its content is left customisable.

Often, elements of actor state are captured through use of annotations. In the MyGrid (http://www.mygrid.org.uk/) project for instance, provenance data is generated from bioinformatics experiments and classed into: *the derivation path*: by which the results were generated from the input data, and *annotations*: associated with a particular object or collection of objects. Such annotations may include elements of actor state such as version data for workflows and resources [7]. Formal representation of actor state, as well as automation of its collection independent of application constraints, is desirable and would be useful in evaluating the state of actors across research disciplines.

The use of performance data to obtain insights into the relationship between application and hardware and software has previously been explored [8], enabling automatic model generation through performance analysis. Within job-based execution environments, work has been performed to enable provenance recording with a minimal level of system intrusion [9]. Automatic instrumentation of such applications with performance monitoring code is possible due to direct availability of implementation. The trade-offs between the level of intrusion to both the application system and user, necessary to capture adequate provenance information has previously been likened to a cube [9] where intrusion to the system and user are modelled on the x and y axis and the amount of available information on the z-axis. The most desirable system is described as one with no intrusion to system or user, but providing all information about the two. In service oriented systems instrumentation of actors may not be possible, due to their loosely-coupled interaction with the querying actor. The level of intrusion which is possible in such environments is therefore minimal, and as such so is the level of information able to be captured. Our system differs through exploring how service oriented systems (where direct knowledge of implementation may be unknown) may record assertions of actor state using resources which are not necessarily part of the application system, alongside interaction assertions. The combined use of such assertions is possible in two ways, understanding how an actor performed within the context of an interaction and secondly understanding what an actor was doing when a particular performance pattern occurred.

6 Conclusion

The use of assertions of client or service (actor) state are not often documented within service oriented architectures, despite them being a critical component in determining the process which led to a particular data item (its provenance). This has been due to the application dependent nature of such data. Actor state data can be combined with assertions of interaction between actors to enable better reasoning within a provenance system. We have identified a number of requirements for a system capable of actor state capture in an application independent manner and attempted to provide a registry based architecture which is able to satisfy them. As future work, we plan to optimise our prototype to produce a more scalable solution and investigate other sources of monitoring information which could be used within our architecture. Of particular interest is the monitoring of per-actor metrics and patterns of access, especially when multiple actors co-exist on the same platform.

References

1. Groth, P., Jiang, S., Miles, S., Munroe, S., Tan, V., Tsasakou, S., Moreau, L.: An Architecture for Provenance Systems. Technical Report (v0.6), University of Southampton (2006) [Online]. Available: http://eprints.ecs.soton.ac.uk/12023/.
2. Miles, S., Groth, P., Branco, M., Moreau, L.: The requirements of recording and using provenance in e-Science experiments. Technical report, University of Southampton (2005) [Online]. Available: http://eprints.ecs.soton.ac.uk/11189/.
3. Groth, P., Luck, M., Moreau, L.: A protocol for recording provenance in service-oriented Grids. In: Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS'04), Grenoble, France (2004) [Online]. Available: http://eprints.ecs.soton.ac.uk/11914/.
4. Ali, A.S., Rana, O.F., Parmee, I.C., Abraham, J., Shackelford, M.: Web-Services Based Modelling/Optimisation for Engineering Design. In: OTM Workshops. (2005) 244–253
5. Kloss, G.K., Schreiber, A.: Provenance Implementation in a Scientific Simulation Environment. In: Proceedings of the International Provenance and Annotation Workshop (IPAW'06), Chicago, USA, Springer-Verlag (2006)
6. Andreozzi, S., Burke, S., Field, L., Fisher, S., Konya, B., Mambelli, M., Schopf, J.M., Viljoen, M., Wilson, A.: Glue Schema Specification. Technical Report (v1.2) (2005) [Online]. Available: http://infnforge.cnaf.infn.it/glueinfomodel/index.php/Spec/V12.
7. Greenwood, M., Goble, C., Stevens, R., Zhao, J., Addis, M., Marvin, D., Moreau, L., Oinn, T.: Provenance of e-Science Experiments - experience from Bioinformatics. In: Proceedings of the UK OST e-Science second All Hands Meeting 2003 (AHM'03), Nottingham, UK (2003) 223–226
8. Taylor, V.E., Wu, X., Stevens, R.L.: Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. SIGMETRICS Performance Evaluation Review **30**(4) (2003) 13–18
9. Reilly, C.F., Naughton, J.F.: Exploring Provenance in a Distributed Job Execution System. In: Proceedings of the International Provenance and Annotation Workshop (IPAW'06), Chicago, USA, Springer-Verlag (2006)