A protocol for recording provenance in service-oriented Grids

Paul Groth, Michael Luck, and Luc Moreau

School of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom
{pg03r, mml, l.moreau}@ecs.soton.ac.uk

Abstract. Both the scientific and business communities, which are beginning to rely on Grids as problem-solving mechanisms, have requirements in terms of provenance. The provenance of some data is the documentation of process that led to the data; its necessity is apparent in fields ranging from medicine to aerospace. To support provenance capture in Grids, we have developed an implementation-independent protocol for the recording of provenance. We describe the protocol in the context of a service-oriented architecture and formalise the entities involved using an abstract state machine or a three-dimensional state transition diagram. Using these techniques we sketch a liveness property for the system.

Keywords: recording provenance, provenance, grids, web services, lineage

1 Introduction

A Grid is a system that coordinates computational resources not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service [4]. By coordinating diverse, distributed computational resources, Grids can be used to address large-scale problems that might otherwise be beyond the scope of local, homogenous systems. Grids are being developed to run a wide variety of applications for both the business and science communities. Scientific applications include the analysis of data from the Large Hadron Collider (lcg.web.cern.ch/LCG/), experiments in surface chemistry [3] and next generation climate research. Grids are used in the business community to support aircraft simulation, seismic studies in the petroleum industry, and to provide faster portfolio recommendations in financial services (www.ibm.com/grid).

These communities also have requirements in terms of *provenance*. We define the provenance of some data as the documentation of the *process* that led to the data. The necessity for provenance is apparent in a wide range of fields. For example, the American Food and Drug Administration requires that the provenance of a drug's discovery be kept as long as the drug is in use (up to 50 years sometimes). In chemistry, provenance is used to detail the procedure by which a material is generated, allowing the material to be patented. In aerospace, simulation records as well as other provenance data are required to be kept up to 99 years after the design of an aircraft. In financial auditing, the American Sarbanes-Oxley Act requires public accounting firms to maintain the provenance of an audit report for at least seven years after the issue of that report (United States Public Law No. 107-204). In medicine, the provenance of an organ is vital for its effective and safe transplantation. These are just some examples of the requirements for provenance in science and business. Provenance is particularly important when there is no physical record as in the case of a purely *in silico* scientific process.

Given the need for provenance information and the emergence of Grids as infrastructure for running major applications, a problem arises that has yet to be fully addressed by the Grid community, namely, how to record provenance in Grids? Some bespoke and ad-hoc solutions have been developed to address the lack of provenance recording capability within the context of specific Grid applications. Unfortunately, this means that such provenance systems cannot interoperate. Therefore, incompatibility of components prevents provenance from being shared.

Furthermore, the absence of components for recording provenance makes the development of applications requiring provenance recording more complicated and onerous.

Another drawback to current bespoke solutions is the inability for provenance to be shared by different parties. Even with the availability of provenance-related software components, the goal of sharing provenance information will not be achieved. To address this problem, standards should be developed for how provenance information is recorded, represented, and accessed. Such standards would allow provenance to be shared across applications, provenance components, and Grids, making provenance information more accessible and valuable. In summary, the paucity of standards, components, and techniques for recording provenance is a problem that needs to be addressed by the Grid community. The focus of this work, the development of a general architecture and protocol for recording provenance, is a first step towards addressing these problems.

The rest of the paper is organised as follows: Section 2 presents a set of requirements that a provenance system should address. Then, Section 3 outlines a design for a provenance recording system in the context of service-oriented architectures. The key element of our system is the Provenance Recording Protocol described in Section 4. In Section 5, the actors in the system are formalised, and the formalisations are then used, in Section 6, to derive some important properties of the system. Finally, Section 7 discusses related work, followed by a conclusion. Given the length of this paper, we assume the reader is familiar with Grids, Virtual Organisations (VO), Web Services, and service-oriented architectures (SOA).

2 Requirements

We have identified a number of requirements that a provenance system should support through an initial requirements gathering process. The following seven requirements have been of particular importance in motivating the development of our architecture and protocol.

1. Verifiability A provenance system should have the ability to verify a process in terms of the actors involved, their actions and their relationship with one another.

2. Accountability Closely related to verifiability is accountability. An actor should be accountable for its actions in a process. Therefore, a provenance system should record in a non-repudiable manner any provenance generated by an actor.

3. Reproducibility A provenance system should, at a minimum, be able to repeat a process and possibly reproduce a process from the provenance that it has stored.

4. Preservation A provenance system should have the ability to maintain provenance information for an extended period of time. This is vital for applications run in the VO context because even after a VO disbands, provenance will typically need to be maintained.

5. Scalability Given the large amounts of data that Grid applications handle, such as in the processing of data from the Large Hadron Collider, it is necessary that a provenance system be scalable. Another reason for scalability is that provenance information may be larger than the output data of an application.

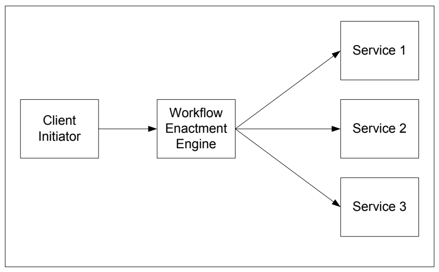
6. Generality Grids are designed to support a wide variety of applications, therefore, a provenance system should be general enough to record provenance from these varying applications.

7. Customisability To allow for more application specific use of provenance information, a provenance system should allow for customisation. Aspects of customisability could include constraints on the type of provenance recorded, time constraints on when recording can take place, and the granularity of provenance to be recorded.

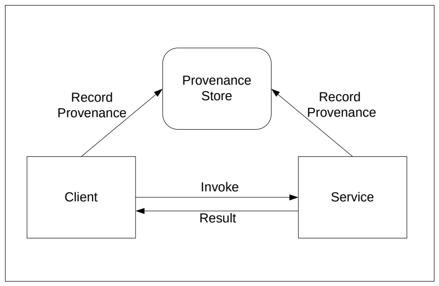
With these requirements in mind, we now detail our conceptual architecture for recording provenance in a SOA.

3 Conceptual Architecture

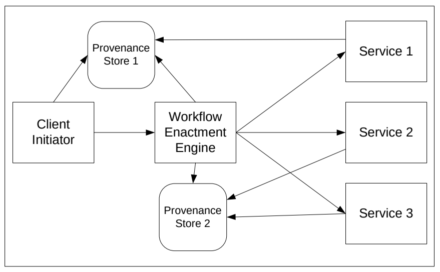
Figure 1(a) shows a typical workflow based service-oriented architecture. A client initiator invokes a workflow enactment engine which, in turn, invokes various services based on the workflow specified by the initiator, finally, a result is returned to the initiator. In essence, the architecture can be broken down into two types of actors: clients who invoke services and services that receive invocations and return results.



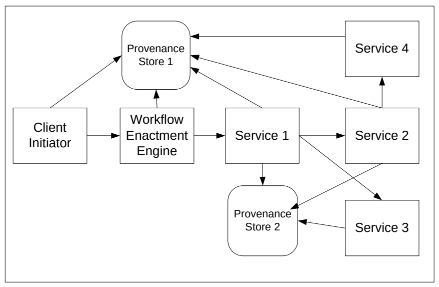
(a) Typical workflow based architecture



(b) The interaction between a client service and provenance store



(c) Workflow based architecture with provenance recording



(d) Architecture with provenance recording and services invoking other services

Fig. 1. Architecture diagrams

Given these types of actors and their method of communication, we have identified two kinds of provenance that exist in a service-oriented architecture. The first kind of provenance is interaction provenance. For some data, interaction provenance is the documentation of interactions between actors that led to the data. In a SOA, interactions are, fundamentally, a client invoking a service. Therefore, interaction provenance can be obtained by recording the inputs and outputs of the various services involved in generating a result. The second type of provenance we have identified is actor provenance. For some data, actor provenance is documentation that can only be provided by a particular actor pertaining to the process that led to the data.

Within the context of these kinds of provenance, our architecture introduces a third type of actor, the provenance store.

Third Party Provenance Stores We see third party provenance stores as key to fulfilling the requirements outlined above. In terms of preservation, placing the burden of maintaining provenance on third party stores means that neither clients nor services must maintain provenance information beyond the scope of any given application run. An additional benefit of third party provenance stores is that they provide a method for legacy applications to maintain provenance. In order to better understand how provenance stores help to address the other requirements, we now explain the recording process for interaction provenance.

A Triangle of Interaction Our architecture records interaction provenance in the following manner. For each interaction between a client and service, consisting of an invocation and a result, each party is required to submit their view of the interaction to a common provenance store. Even though our architecture considers multiple actors, the interaction between all these actors can be reduced down to a common 'triangular' pattern of interaction described above and shown in Figure 1(b). This reduction is possible because our system contains only three types of actors, the client, service and provenance store, where the store exists in order to record a copy of the simple one-to-one interaction of the client and service. The interaction of these three actors is governed by Provenance Recording Protocol, which we will detail later in the paper.

Uses of Interaction Provenance The client-service interactions that our architecture records make up the interaction provenance for some data. This interaction provenance can be used to repeat or even reproduce the process that led to the data. For example, if the services involved in a process have not changed, the inputs to these services, stored in the provenance, can be used to reinvoke the corresponding services reproducing the process. Other uses of interaction provenance include holding actors to account for their inputs and outputs and for the verification of processes.

The case for recording two views However, if interaction provenance is to be used for reproduction, accountability or verification purposes, the interactions recorded must be agreed upon by the actors involved. Each actor has its own view of an interaction, which, at its most basic, is the input and output of the actor in an interaction. Therefore, in our architecture a client and service must submit their view of an interaction to a common provenance store, which can then check that the actors agree on their interaction. Without verification by the provenance store, several problems could arise, particularly in open environments.

For example, should the client be the only party recording the interaction in the provenance store, the service would be reliant on the client to submit provenance. In fact, without the submission of provenance from the service, there would be no evidence that the client invoked the service should the client not record the interaction. Given that a service can be held accountable for its actions recorded in the provenance store, this is unacceptable. In our system, the provenance store would know that a service was invoked because the service submits that information. The same problem would also exist in the case where the service was the only party submitting to the provenance store. We note that the requirement that both parties submit their views does not prevent collusion between parties, but it does allow the provenance store to detect when the two parties disagree about the record of an interaction.

Multiple provenance stores Although a client and service are required to share a common provenance store for an interaction, different provenance stores can be used for different interactions even between the same client and service. Figure 1(c) shows a typical workflow based architecture with multiple provenance stores. This architecture is assembled from the 'triangle' pattern pictured in Figure 1(b). One benefit of multiple provenance stores is the elimination of a central point of failure. Another benefit is that demand is spread across multiple services increasing the architecture's robustness. These benefits help to address the scalability requirement.

Advanced Architecture Support As well as supporting typical workflow enactment based architectures, our system supports more advanced architectures like the one shown in Figure 1(d). In this architecture, services invoke other services to produce a result, in contrast to the previous architecture where the workflow enactment engine was the only actor invoking services. In order to maintain provenance across provenance stores, a client needs to inform the original

provenance store when it uses a new provenance store. For example, in Figure 1(d), Service 1 must inform Provenance Store 1 that it has used Provenance Store 2 when invoking Service 3. This creates a link between provenance records stored in different stores that can be followed in order to provide the entire provenance trace for an application started by a client initiator.
Actor Provenance We have mainly discussed how our system supports the recording of information about the interaction between actors in a service-oriented architecture. Our system also supports actor provenance, which could include anything from the workflow that an enactment engine runs to the disk and processing power a service used in a computation. This information can only be provided by the actor itself, so it cannot be verified like interaction provenance. We use a simple mechanism to store actor provenance by tying it to interaction provenance. The basis for our provenance recording system is the interaction between one client, one service and one provenance store. This interaction is specified by the Provenance Recording Protocol, which is presented next.

4 Recording Protocol

PReP is a four phase protocol consisting of negotiation, invocation, provenance recording and termination phases. The negotiation phase allows a client and service to agree on a provenance store to store a trace of their interaction. After this phase, the protocol enters the invocation phase, during which a client invokes a service and receives a result. Asynchronously, in the provenance recording phase, both the client and service submit their input and output data to the provenance store. When all data has been received by the provenance store, the termination phase occurs.

Name	Notation	Fields
propose	pro	ActivityId, PSAllowedList, Extra
reply	reply	ActivityId, PSAccepted, Extra
invoke	inv	ActivityId, Data, Extra
result	res	ActivityId, Data, Extra
record negotiation	rec_neg	ActivityId, PSAllowedList, PSAccepted, Extra
record negotiation acknowledgement	rec_neg_ack	ActivityId
record invocation	rec_inv	ActivityId, Extra, Data
record invocation acknowledgement	rec_inv_ack	ActivityId
record result	rec_res	ActivityId, Data
record result acknowledgement	rec_res_ack	ActivityId
submission finished	sf	ActivityId, NumOfMessages
submission finished acknowledgement	sf_ack	ActivityId
additional provenance	ap	ActivityId, Extra
additional provenance acknowledgement	ap_ack	ActivityId

Fig. 2. Protocol messages, their formal notation and message parameters.

First, we discuss the messages and their parameters used by PReP, then we consider the four phases in detail. We model the protocol as an asynchronous message-passing system, in which all communication is expressed by an outbound message followed by a return message. The return message is either a result of the service invocation, a reply from the service during negotiation, or an acknowledgement that the provenance store has received a particular message. Figure 2 lists the fourteen messages in our protocol. The usage of each message is described in more detail when we present the phases of the protocol. The message parameters shown in Figure 2 are detailed below.

The ActivityId parameter identifies one exchange between a client and server. It contains: NonceId, an identifier generated by the client to distinguish between other exchanges with the called service; SessionId, comprising all invocations that pertain to one result (the client originator of Figure 1(c) generates this identifier, which must be unique); ThreadId, which allows clients to parse multiple interactions with the same service; Client, which identifies the client; and Service, which identifies the service.

Other parameters are: Data, which contains data exchanged between a client and service; Extra, which is an envelope that can contain other messages related or not to the protocol allowing it to be extended; NumOfMessages, which indicates the total number of messages an entity sends to the provenance store; PSAllowedList, which is a list of approved provenance stores; and PSAccepted, which contains a reference to a provenance store that an entity accepts, or a rejection token.

PReP is divided into four phases: negotiation, invocation, provenance recording, and termination, which we now discuss in detail.

Negotiation is the process by which a client and service agree on a provenance store to use. Typically, a client presents a list of provenance stores to the service via a *propose message*. The service then extracts the PSAllowedList from the propose message and selects a provenance store from the list. The service then replies with a *response message* containing the selected provenance store or a rejection in the PSAccepted parameter. Although the negotiation modelled here is simple, with only one request-response, the protocol is extensible through the use of the Extra parameter. Entities can encode more complicated messages into this envelope, providing a means for complex negotiations to take place. A client and service that have already negotiated and agreed on a provenance store might like to skip the negotiation phase of the protocol. Therefore, a message informing the service of the use of a previously agreed provenance store can be enclosed in the Extra envelope of the *invoke message*. However, the provenance store still needs to be informed of the agreement between the service and client via the *record negotiation message*.

Invocation If a client has successfully negotiated with a service, it can then invoke the service and receive a result via the *invoke message* and *result message*. We have tried to limit the impact of PReP on normal invocation, the only extra parameters required to be sent are the ActivityId and the Extra envelope. The ActivityId is necessary to identify the exchange in relation to the provenance stored in the service, while the Extra envelope allows the protocol to be used without a negotiation phase and for later protocol extension.

Provenance Recording is the key phase of the protocol. As discussed previously, the client and service are required to submit copies of all their sent and received messages to the provenance store. Submission is done through the various record messages with both the client and service sending *record negotiation*, *record invocation* and *record result* messages. Acknowledgement messages then inform the sender that each message has been received by the provenance store. The *record negotiation message* contains the list of provenance stores (PSAllowedList), the client proposed, and the provenance store accepted (PSAccepted) by the service. The *record invocation* and *record result* messages together contain the entire data transmitted between the client and service from the perspective of both entities. The requirement that all data be submitted allows the provenance store to have a complete view of the exchange. In order not to delay service invocation, the submission process can be done in a totally asynchronous fashion; for example, the client could send a *record invocation message* to the provenance store before or after receiving a *result message* from the service.

We cater for actor provenance instead of interaction provenance by the *additional provenance message*. With this message, an actor can record provenance about itself or other actors in the architecture by enclosing in the Extra envelope whatever information is pertinent. An important use of this capability is the linking of provenance records across provenance stores as described in Section 3. We note that there are no constraints on the data that can be submitted to the provenance store, allowing a wide variety of applications to be supported.

Termination The final phase of the protocol is termination. The protocol terminates when the provenance store has received all expected messages from both the client and the service. The client and service are notified of termination through the acknowledgement to the *submission finished message*, which is returned after all expected messages are received from the client and service. The number of expected messages is determined by the NumOfMessages parameter in the *submission finished message*. Because of the asynchronous nature of the protocol, the *submission finished message* can be sent any time after the negotiation phase.

5 Actors

We now consider how the provenance store, service and client act in response to the messages they send and receive. To understand the actions of these actors, we use complementary formalisation techniques, chosen because of the nature of the actors involved. First, we represent the provenance store as an abstract state machine (ASM). Second, we use a 3D state diagram to show the possible responses of the client and service. Both techniques assume asynchronous message passing. The importance of the internal functionality of the provenance store lends itself to an ASM formalisation whereas, given the importance of the external interactions of the client and service, a state transition diagram formalisation is more appropriate. We begin with the provenance store.

The Provenance Store plays the central role in PReP. As far as recording is concerned, its interaction with the outside world is simple: it receives messages and sends acknowledgements. It does not initiate any communication and its purpose is to simply store messages. By formalising the provenance store, we can explain how the accumulation of messages dictates its actions.

To detail these actions, we model the provenance store as an ASM whose behaviour is governed by a set of transitions it is allowed to perform. The notation allows for any form of transition with no limits on complexity or granularity and has been used previously to describe a distributed reference counting algorithm [6].

The ASM State Space The state space of the provenance store's ASM is shown in Figures 3 and 4. The System State Space models the space of messages and message channels that actors in the system use to communicate, whereas the Provenance Store State Space models the internal state space of provenance stores. We first describe the System State Space.

The System State Space considers a finite number of actors, A, which exchange messages. The set of messages is defined as the union of the sets RN, RI, RR, SF, and AP. All of these sets, excluding AP, are in turn defined by inductive types, whose constructors are named according to the messages in Figure 2. Communication between actors is modelled as a set of communication channels represented as bags of messages between pairs of actors.

An instance of a provenance store actor, p, is a tuple that consists of an element from the Client Message Store, CS, an element from the Service Message Store, SS, and an element from the set of communication channels, $\mathcal{K}$. The two tables are defined as functions whose argument is of type ActivityId and consist of sets of messages that are from either the client or the service. On the other hand, AP is a set that contains all of the *additional provenance messages*. Note that SS and CS are not defined using AP but with APL, the power set of AP. Informally, this shows that any number of *additional provenance messages* can be stored per ActivityId.

Given the state space, the ASM is described by an initial state and a set of transitions. Figure 4 contains the initial state space, which can be summarised as empty client stores, empty service message stores, and empty communication channels. We use an arrow notation for a function taking an argument and returning a result. Therefore, $client\_T_i$ and $service\_T_i$ take an ActivityId as an argument and return an empty state.

$A = \{a_1, a_2, \ldots, a_n\}$ (Set of Actors)
$\textsc{Client} \subset A$ (Set of Clients is a subset of Actors)
$\textsc{Service} \subset A$ (Set of Services is a subset of Actors)
$\textsc{ActivityId} = \textsc{SessionId} \times \textsc{NonceId} \times \textsc{ThreadId} \times \textsc{Client} \times \textsc{Service}$ (Activity Identification)

$\mathsf{rec\_neg}{:}\textsc{ActivityId} \times \textsc{PSAllowedList} \times \textsc{PSAccepted} \times \textsc{Extra} \rightarrow RN$ (Negotiation Messages)
$\mathsf{rec\_inv}{:}\textsc{ActivityId} \times \textsc{Extra} \times \textsc{Data} \rightarrow RI$ (Invocation Messages)
$\mathsf{rec\_res}{:}\textsc{ActivityId} \times \textsc{Extra} \times \textsc{Data} \rightarrow RR$ (Result Messages)
$\mathsf{sf}{:}\textsc{ActivityId} \times \textsc{NumOfMessages} \rightarrow SF$ (Submission Finished Messages)
$\mathsf{ap}{:}\textsc{ActivityId} \times \textsc{Extra} \rightarrow AP$ (Additional Provenance Messages)

$\mathcal{M} = RN \cup RI \cup RR \cup SF \cup AP$ (Messages)
Each message has a corresponding acknowledgement message, which is also a part of $\mathcal{M}$.

$\mathcal{K} = A \times A \rightarrow Bag(\mathcal{M})$ (Set of Message Bags)

Charateristic Variables:
$a \in A, k \in \mathcal{K}$, $ai \in \textsc{ActivityId}$, $\mathit{rec\_neg} \in RN, \mathit{rec\_inv} \in RI$, $\mathit{rec\_res} \in RR, sf \in SF, ap \in AP$, $e \in \textsc{Extra}$, $psal \in \textsc{PSAllowedList}$, $psa \in \textsc{PSAccepted}$, $d \in \textsc{Data}$, $nid \in \textsc{NonceId}$, $tid \in \textsc{ThreadId}$, $client \in \textsc{Client}$, $service \in \textsc{Service}$, $nm \in \textsc{NumOfMessages}$

If $ai = \langle sid, nid, tid, ts, client, service\rangle$ then
$ai.sid = sid, ai.nid = nid, ai.tid = tid, ai.ts = ts, ai.client = client, ai.service = service$
If $sf = \langle ai, nm\rangle$ then $sf.ai = ai, sf.nm = nm$

Fig. 3. System State Space

$APL = \mathbb{P}(AP)$ (Set of Sets of Additional Provenance Messages)
$CN = RN$ (Client Negotiation Messages)
$CI = RI$ (Client Invocation Messages)
$CR = RR$ (Client Result Messages)
$CSF = SF$ (Client Submission Finished Messages)
$SN = RN$ (Service Negotiation Messages)
$SI = RI$ (Service Invocation Messages)
$SR = RR$ (Service Result Messages)
$SSF = SF$ (Service Submission Finished Messages)
$CS = \textsc{ActivityId} \rightarrow CN \times CI \times CR \times CSF \times APL$ (Client Records, a Client Message Store)
$SS = \textsc{ActivityId} \rightarrow SN \times SI \times SR \times SSF \times APL$ (Service Records, Service Message Store)
$PS = CS \times SS$ (Set of Provenance Stores)

Characteristic variables:
$p = \langle \mathit{client\_T}, \mathit{service\_T}, k\rangle, p \in A$, $apl \in APL, \mathit{client\_T} \in CS, \mathit{service\_T} \in SS, ps \in PS$
If $\mathit{service\_T}[ai] = \langle \mathit{rec\_neg}, \mathit{rec\_inv}, \mathit{rec\_res}, sf, apl\rangle$ then
$\mathit{service\_T}[ai].\mathit{rec\_neg} = \mathit{rec\_neg}, \mathit{service\_T}[ai].\mathit{rec\_inv} = \mathit{rec\_inv}$,
$\mathit{service\_T}[ai].\mathit{rec\_res} = \mathit{rec\_res}, \mathit{service\_T}[ai].sf = sf, \mathit{service\_T}[ai].apl = apl$
The same notation applies for $\mathit{client\_T}[ai]$.
Initial State:
$p_i = \langle \mathit{client\_T}_i, \mathit{service\_T}_i, k_i\rangle$, $\mathit{client\_T}_i = ai \rightarrow \emptyset$, $\mathit{service\_T}_i = ai \rightarrow \emptyset, k_i = \emptyset$

Fig. 4. Provenance Store State Space

The ASM Rules The transitions of the ASM are described through rules, which follow the format presented in Figure 5. Rules are identified by their name and a number of parameters that the rule operates over. Any number of conditions must be met in order for a rule to be fireable. A new state is achieved after applying all the pseudo-statements and functions to the state that met the conditions of the rule. The execution of a rule is atomic, so that no other rule may interrupt or interleave with an executing rule. This maintains the consistency of the ASM. A rule may contain *send*, *receive* or table update pseudo-statements. Informally, $send(a_1, a_2, m)$ inserts a message m into the channel from actor a_1 to actor a_2, and $receive(a_1, a_2, m)$ removes the message. A rule may also contain the *complete* function, which checks that none of the fields accessed by an ActivityId are null. Formally, the pseudo-statements are defined as follows.

- If k is the set of message channels of a state $\langle \ldots, k\rangle$, then the expression $send(a_1, a_2, m)$ denotes the state $\langle \ldots, k'\rangle$, where [1] $k'(a_1, a_2) = k(a_1, a_2) \oplus \{m\}$, and $k'(a_i, a_j) = k(a_i, a_j), \forall (a_i, a_j) \neq (a_1, a_2)$.
- If k is the set of message channels of a state $\langle \ldots, k\rangle$, then the expression $receive(a_1, a_2, m)$ denotes the state $\langle \ldots, k'\rangle$, where $k'(a_1, a_2) = k(a_1, a_2) \ominus \{m\}$, and $k'(a_i, a_j) = k(a_i, a_j), \forall (a_i, a_j) \neq (a_1, a_2)$.
- If $table\_T$ is a component of state $\langle \ldots, table\_T, \ldots\rangle$, then the expression $table\_T[ai].y := V$ denotes the state $\langle \ldots, table\_T', \ldots\rangle$, where $table\_T[ai].x = table\_T'[ai].x$ if $x \neq y$, and $table\_T'[ai].y = V$.

Likewise, the function *complete* is defined as follows:

- If $client\_T$ and $service\_T$ are components of a state $\langle client\_T, service\_T, \ldots\rangle$, then the expression $complete[ai]$ evaluates to true if $client\_T[ai].rec\_neg \neq \bot$, $client\_T[ai].rec\_inv \neq \bot$, $client\_T[ai].rec\_res \neq \bot$, $client\_T[ai].sf \neq \bot$, $client\_T[ai].sf.nm - 4 = |client\_T[ai].apl|$ and $service\_T[ai].rec\_neg \neq \bot$, $service\_T[ai].rec\_inv \neq \bot$, $service\_T[ai].rec\_res \neq \bot$, $service\_T[ai].sf \neq \bot$, $service\_T[ai].sf.nm - 4 = |service\_T[ai].apl|$.

Figure 6 shows one of the ASM's transition rules. *receive_neg* is the transition rule for the receipt of a record negotiation message. It specifies the behaviour of a provenance store actor when receiving, from actor a, a rec_neg message containing: an ActivityId, a PSAllowedList, a PSAccepted parameter and an Extra envelope. The condition placed on the rule states that for the rule to fire there must be a rec_neg message, which is part of the communication channel ($\mathcal{K}$) between a provenance store actor, p, and a. If this condition is satisfied, the message is consumed using the *receive* pseudo-statement. The rule then determines whether a is a client or service and puts the rec_neg message in the correct field of the appropriate table. After this table update, an rec_neg_ack is sent using the *send* pseudo-statement, which places the given message onto the communication channel between the specified entities. Finally, the *complete* functions tests to see if all messages have been received from both the client and the service. If all messages have been received, the *submission finished acknowledgement message* can be sent. The other four transitions listed follow the same pattern as the *receive_neg* rule, consuming a message and placing it into the the correct field of the appropriate table. The entire set of rules can be found at http://www.pasoa.org/protocol/rules.htm.

```
$rule\_name(v_1, v_2, \cdots)$ :
    $condition_1(v_1, v_2, \cdots)$
    $\wedge condition_2(v_1, v_2, \cdots) \wedge \cdots$
$\rightarrow$ {
    $pseudo\_statement_1$;
    $\cdots$
    $pseudo\_statement_n$;
  }
```

Fig. 5. Rule format

```
$receive\_neg(p, a, ai, psal, psa, e)$ :
    $\mathsf{rec\_neg}(ai, psal, psa, e) \in \mathcal{K}(ps, a)$
$\rightarrow$ {
    $receive(p, a, \mathsf{rec\_neg}(ai, psal, psa, e))$;
    $if$ $(a = ai.client)$, then
        $client\_T[ai].rec\_neg := \mathsf{rec\_neg}(ai, psal, psa, e)$;
    $elif$ $(a = ai.service)$, then
        $service\_T[ai].rec\_neg := \mathsf{rec\_neg}(ai, psal, psa, e)$;
    $send(p, a, \mathrm{rec\_neg\_ack}(ai))$;
    $if$ $complete[ai]$, then
        $send(p, a, \mathsf{sf\_ack}(ai))$;
  }
```

Fig. 6. Receive negotiation rule

[1] We use the operators $\oplus$ and $\ominus$ to denote union and difference on bags.

The Client and Service We now formalise the actions of the client and the service. In this case, we have chosen not to use the ASM formalism because we have no knowledge of the decision algorithm a service would use when selecting a provenance store from the list proposed by the client. Furthermore, we want developers to be free to experiment with any sort of algorithm they deem best. However, we still want to formally investigate the actions of the client and service in response to PReP, so we represent the two entities with a 3D state transition diagram, which offers an intuitive yet rigorous means to describe the actions of the client and service based on sent and received messages.

Figure 7 shows the state transition diagram for both the client and service. It contains all the possible states of a client or service with regard to the PReP. Transitions between states are only permitted when messages are sent or received by the actor. These transitions are identified by the transition keys in the diagram. For example, transition (4) is the receipt of a *result message* and transition (5) is the sending of an *invoke message* in the case of the client. The diagram shows all possible ways that a client or service could send and receive messages.

We believe that these formalisations provide a firm basis for developers to implement the protocol. The ASM and 3D state transition diagram allow developers to understand the interaction of the client, service, and provenance store without prescribing a particular implementation technique. This gives developers the opportunity to choose the implementation mechanisms that fit their needs.

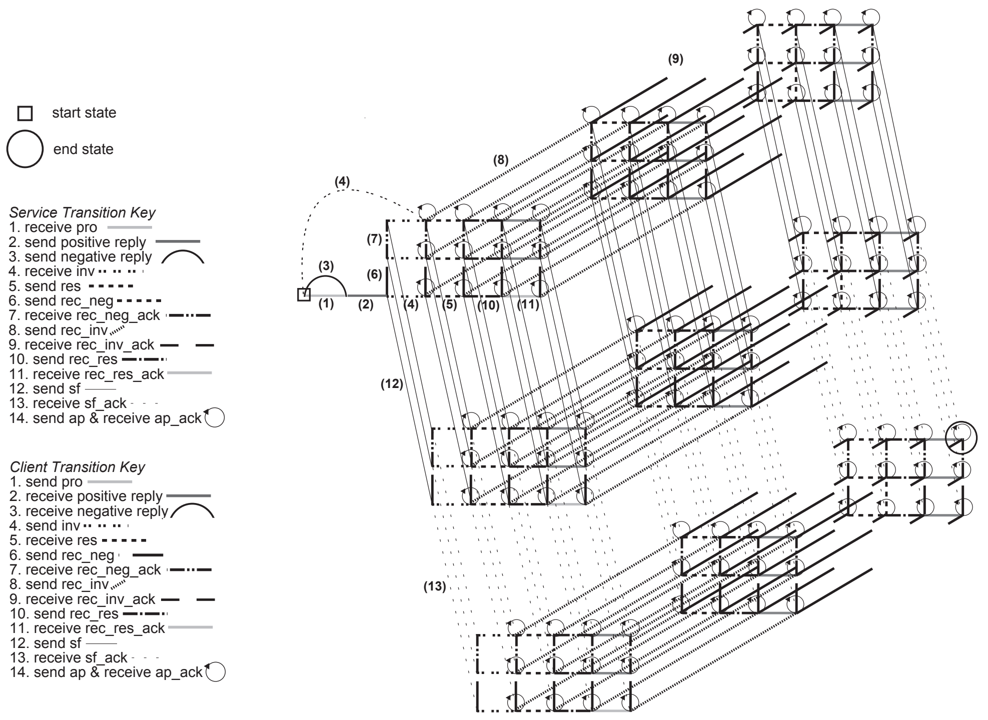


Fig. 7. State transition diagram for both the client and service

6 Properties

Given the above formal representations of the client, service and provenance store, we now can show an important property of PReP, namely, liveness. In distributed systems, it is common to refer to safety and liveness properties, to denote, respectively, that nothing bad will happen and that something good will eventually happen. In the case of PReP, liveness is that, ultimately, the *submission finished acknowledgement* message will be sent to both the client and the service.

To show that the protocol is indeed live, we first make some assumptions about the system implementing PReP. We assume that the client and service are live i.e. that they will eventually send and receive all the messages designated in the protocol. This entails that for any given invocation a service will always respond. Finally, we assume that all communication channels are live. Therefore, all sent messages will be delivered to the addressed party.

Given these assumptions, we now show that both the client and service will eventually end their interaction with the provenance store for one invocation of the service.

Lemma 1 (Termination) Given a finite number of exchanged messages, the actions of the client and service in relation to PReP will terminate for one invocation of a service. *Proof* Figure 7 shows, by definition, the actions of the client and service in relation to PReP for one invocation of a service. We then derive the assumption that there are a finite number of *additional provenance messages*, because the *submission finished message* requires that a finite number of messages be specified. Next, we can determine a bound on the number of messages a client or service will exchange. Excluding *additional provenance messages*, we calculate this bound by enumerating all paths from the start state to the end state in the graph and selecting the longest, which is twelve transitions. This means that a client or service will exchange a maximum of twelve messages. Given this fixed bound and a finite number of *additional provenance messages*, the client and service will reach the end state shown in the graph and terminate.

Lemma 2 (Completeness) A provenance store can determine when it has a complete record of a service invocation. *Proof* We define a complete record as the function *complete* evaluating to true. An invocation is identified by an ActivityId. Therefore, by definition, the provenance store can determine when it has a complete record for a service invocation.

Lemma 3 (PReP satisfies the liveness property) The *submission finished acknowledgement* message will be sent to both the client and the service. *Proof* Given that both the client and service will terminate (Lemma 1), both actors will send all their messages to the provenance store, which, as represented by the state machine, will fire the appropriate rule corresponding to the receipt of each message. These rules in turn update the state of the record referenced by an ActivityId, *ai* and check for a complete record (Lemma 2) and, if it exists for *ai*, the *submission finished acknowledgement* is sent.

7 Related Work

Provenance recording also been investigated in the myGrid (www.mygrid.org.uk) project, whose goal is to provide a personalised "workbench" for bioinformaticians to perform *in-silico* experiments [7]. Although myGrid allows users to capture provenance data [10], it does not not address general architectures or protocols for recording provenance.

Ruth *et al.* present a system for recording provenance in the context of data sharing by scientists [8]. Each scientist has an e-notebook which records and digitally signs any input data or manipulations of data. When the data is shared via peer-to-peer communication, a scientist cannot refute the provenance of the data because of the digital signature process. The goal of the system is to generate a virtual community where scientists are accountable for their data. [8] focuses mainly on the trust aspect of the e-notebook system, rather than the protocols for distributing and storing provenance data.

Some work has focused on data provenance in databases. Buneman et al. [2] make the distinction between *why* (which tuples in a database contribute to a result) provenance and *where* (the location(s) of the source database that contributed to a result) provenance. In [1], a precise definition of provenance is given for both XML hierarchy and relational databases.

Szomszor and Moreau [9] argued for infrastructure support for recording provenance in Grids and presented a trial implementation of an architecture that was used to demonstrate several mechanisms for handling provenance data after it had been recorded. Our work extends [9] in several important ways. First, we consider an architecture that allows for provenance stores as well as composite services. Secondly, we model an implementation -independent protocol for recording provenance within the context of a service-oriented architecture, whereas, Szomszor and Moreau present an implementation specific service-oriented architecture.

The Chimera Virtual Data System [5] provides a data catalog along with a representation of derivation procedures in order to document data provenance. Chimera focuses on representing and querying data derivation information. We imagine that PReP could be used as the underlying protocol to store provenance information in a Chimera like system.

Conclusion There are several avenues of future work we intend to pursue in the development of a provenance system. These avenues include, the further specification of PReP in terms of security, the implementation of PReP using Web Services and the integration of PReP into real world scenarios.

The necessity for storing, maintaining and tracking provenance is evident in fields ranging from biology to aerospace. As science and business embrace Grids as a mechanism to achieve their goals, recording provenance will become an ever more important factor in the construction of Grids. The development of common components, protocols, and standards will make this construction process faster, easier, and more interoperable. In this paper, we presented a stepping stone to the development of a common provenance recording system, namely, an implementation-independent protocol for recording provenance, PReP.

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

References

1. P. Buneman, S. Khanna, and W.-C. Tan. Data provenance: Some basic issues. In *Foundations of Software Technology and Theoretical Computer Science*, 2000.
2. P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *Int. Conf. on Databases Theory (ICDT)*, 2001.
3. M. Ford, D. Livingstone, J. Dearden, and H. V. der Waterbeemd, editors. *Comb-e-Chem: an e-science research project*. Blackwell, March 2002.
4. I. Foster. What is the grid? a three point checklist., July 2002.
5. I. Foster, J. Voeckler, M. Wilde, and Y.Zhao. Chimera: A virtual data system for representing, querying and automating data derivation. In *Proc. of the 14th Conf. on Scientific and Statistical Database Management*, July 2002.
6. L. Moreau and J. Duprat. A construction of distributed reference counting. *Acta Informatica*, 37:563–595, 2001.
7. L. Moreau and et. al. On the use of agents in a bioinformatics grid. In S. Lee, S. Sekguchi, S. Matsuoka, and M. Sato, editors, *Proc. of the 3rd IEEE/ACM CCGRID'2003 Workshop on Agent Based Cluster and Grid Computing*, pages 653–661, Tokyo, Japan, 2003.
8. P. Ruth, D. Xu, B. K. Bhargava, and F. Regnier. E-notebook middleware for acccountability and reputation based trust in distributed data sharing communities. In *Proc. 2nd Int. Conf. on Trust Management, Oxford, UK*, volume 2995 of *LNCS*. Springer, 2004.
9. M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *Int. Conf. on Ontologies, Databases and Applications of Semantics*, volume 2888 of *LNCS*, 2003.
10. J. Zhao, C. Goble, M. Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science. In *Proc. of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, October 2003.