Recording Process Documentation for Provenance

Paul T. Groth and Luc Moreau

***Abstract*— Scientific and business communities are adopting large scale distributed systems as a means to solve a wide range of resource intensive tasks. These communities also have requirements in terms of provenance. We define the provenance of a result produced by a distributed system as the process that led to that result. This paper describes a protocol for recording documentation of a distributed system's execution. The distributed protocol guarantees that documentation with characteristics suitable for accurately determining the provenance of results is recorded. These characteristics are confirmed through a number of proofs based on an abstract state machine formalisation.**

***Index Terms*— provenance, lineage, grids, distributed systems, data protocols**

I. Introduction

Scientific and business communities are adopting large scale distributed systems (Grids, Web Services) as a means to solve a wide range of resource intensive tasks. For example, bioinformaticians are using such systems to aid in drug discovery by modelling the folding of proteins [20]. Likewise, in aerospace engineering, practitioners are running simulations of aircraft using networked supercomputers to improve design and safety, while reducing cost [19]. Lastly, financial services firms are using the idle cycles of desktop computers to perform financial analytics with faster turn around times [22]. Beyond their use of large amounts of computational resources in distributed networks, these example applications share another common concern. In each application, the process by which the result was generated is as important as the result itself. For instance, in the aerospace example, if a plane has a malfunction, it is necessary to find where in the design and building process the failure could have arisen. If during the design process the simulation was at fault, it could be improved to prevent future malfunctions in planes that use similar simulation techniques. The process that led to the plane (including its design, construction, and operation) is termed the *provenance* of the plane. Thus, conceptually, we term the process that led to a result, the provenance of that result.

The necessity for provenance is apparent in a wide range of fields and mandated by a number of regulatory authorities. 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 in some cases). Likewise, the Federal Aviation Administration requires that simulation records, as well as other provenance data, 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). Beyond regulatory requirements, provenance is particularly important when there is no physical record as in the case of purely *in silico* distributed scientific processes since provenance provides the only means to validate a result.

Therefore, the ability to determine the provenance of results produced by a computational distributed systems is necessary. However, in most instances, the existence of a computer-derived result itself is not sufficient to determine its provenance. We need to know details of the actual execution (e.g. process) responsible for the result's generation. During the execution of a distributed system, it is possible to automatically create a description of such an execution, which we call *process documentation* and record it in a repository called a *provenance store*. A provenance store can then be queried to retrieve a *concrete representation* of the provenance of the result of interest. To reiterate, process documentation contains concrete representations of the provenance of results (e.g. data) produced by a distributed system. Creating, recording and querying process documentation are core steps of the provenance lifecycle [18]. The focus of this paper is on the creating and recording phase of such a lifecyle; we identify a protocol by which a distributed system can record process documentation in a provenance store. Furthermore, we define the expected behaviour of the distributed system, in the creation and recording phases of the provenance lifecycle, so that the recorded process documentation can help accurately determine the provenance of a result. Consequently, the contribution of this paper are threefold:

1) The definition of five characteristics that make process documentation high-quality.
2) A description and a formal definition of a distributed protocol that records process documentation.
3) Proofs that establish that the protocol records high-quality process documentation.

The rest of this paper is organised as follows. Section II presents a set of characteristics to ensure that process documentation is high-quality. In Section III, we describe a provenance-aware application. An intuitive description of a protocol for recording process documentation follows in Section IV. Section V then defines a formal model of the protocol using an abstract state machine. Based on this formal model, Section VI presents analyses the protocol to show that it supports to the characteristics presented in Section II. Finally, we discuss related work and conclude.

II. High-quality characteristics

Auditors, scientists, and reviewers, need to be confident that the process documentation which they rely upon for provenance is both accurate and comprehensive. We now define a set of five characteristics that help to ensure that process documentation is indeed accurate and comprehensive.

These characteristics were derived from an analysis of use cases from several domains [15]. We looked at both the technical requirements enumerated in the analysis as well as the use cases themselves. We found that in a majority of the use cases, process documentation provides *evidence* that a process occurred. Thus, these characteristics are justified by philosophical arguments that equate process documentation to evidence. Beyond these philosophical arguments, a number of the characteristics also directly support technical requirements.

Characteristic 1 (Immutable): In a legal setting, once evidence has been collected, it is criminal to tamper with it because it corrupts the view the court has on what occurred. Therefore, if we treat process documentation as evidence, once an application has created it for a particular execution, it should not be modified or deleted. Many times, it is not apparent that process documentation is useful until it is needed. Indeed, many users are caught off-guard when data they previously produced is deleted. Thus, it is important to maintain process documentation even when it is thought to be unimportant. Furthermore, in settings where the provenance of data might be used for scientific or legal verification, process documentation must not be tampered with. The requirement for immutable process documentation does not suggest that errors in the documentation can not be corrected. There are a number of ways, outside the scope of our protocol, to correct errors, for example annotation, that do not entail the deletion, replacement, or modification of existing documentation. The essential point is to prevent tampering with process documentation even under the guise of correcting errors.

Characteristic 2 (Attributable): In a court of law, evidence, particularly testimony, is judged by the person or institution who provides it. Furthermore, if it is found that the evidence given is false then remedial action can be taken against the provider. Similarly, if a user deems that process documentation is somehow erroneous, the user must know who is responsible for the creation of the documentation so that the party can be held accountable. By insuring users know the accountable party, they will have greater confidence in process documentation.

Characteristic 3 (Autonomously Creatable): In both criminal and scientific investigations, evidence is gathered at the most appropriate time and by the most appropriate person, device, or institution. By analogy, the components of a distributed systems should be able to create process documentation at a convenient point in time without having to synchronise with any other component.

Characteristic 4 (Finalizable): When process documentation is created within a distributed system, it is helpful to know when the components within the system have fully documented their processes so that the system can know when to query the documentation or relinquish components. Furthermore, it is helpful for users to know when full evidence (i.e. process documentation) has been provided for a particular process, otherwise, it is hard to know when a judgement can be made about the evidence. Thus, process documentation should be finalizable (i.e. markable as the final representation of a past process). This characteristic is particularly helpful in systems where, to achieve performance gains, creation and recording of documentation occur in an asynchronous fashion with respect to the process.

Characteristic 5 (Process Reflecting): Evidence is only useful in a court if it can be put together to make a convincing case that a particular crime occurred. Likewise, process documentation is only useful if it can be put together to show that a process occurred. Thus, for a process documentation to be useful, it should be connected in such a manner that it reflects how the distributed system executed. We term documentation that has this characteristic, process reflecting. This characteristic may be realized in different ways within different systems, for example in a workflow system connections might be conveyed through data dependencies.

We term process documentation that has the above five characteristics as *high quality*. When process documentation has these characteristics, the users of the provenance taken from this evidence can trust that it is both comprehensive and accurate. Later, in Section VI, these high-level characteristics are expressed in a set of formal properties. Using these properties, we show how our protocol ensures, through well-defined protocol messages (Section IV-A) and behavioral obligations (Section IV-B) on the entities within a distributed system, that the process documentation it records is high quality. Before defining the protocol, we define a provenance-aware system.

III. Provenance-aware Systems

To facilitate the presentation, we distinguish between the whole of process documentation and its constituent parts. We term an individual piece of process documentation, a *p-assertion*. Documentation of process, then, is a set of p-assertions. When a query is issued to a provenance store, it also retrieves a set of p-assertions. This terminology is adopted from an Architecture for Provenance Systems [12].

A distributed system can be described as a set of *actors* that communicate by message passing.[1] Each actor can have one or more roles that place specific *obligations* on actors to fulfil the particular roles. The roles used in this paper are defined below:

- An *application actor* is an actor that performs some application functionality irrespective of provenance.
- A *sender* is an application actor that sends messages to other actors.
- A *receiver* is an application actor that receives messages from other actors.
- An *asserter* is an actor that can create p-assertions.
- A *recorder* is an actor that can record p-assertions in a provenance store.
- A *provenance store* is an actor that can persistently store p-assertions.

The introduction of a specialized actor obligated to persistently store p-assertions is made to address the question of *where* process documentation should be stored.

The adoption of provenance stores avoids storing duplicate process documentation for multiple data items; second,

[1]This definition is similar to other definitions of distributed systems in the literature [4].

because they are specialised for the storage of p-assertions, provenance stores can be built to persistently store large amounts of process documentation. Systems that record p-assertions about their execution in provenance stores are referred to as *provenance-aware systems*.[2] Using the above role definitions, we now define a protocol for recording p-assertions into provenance stores.

IV. PReP: The P-assertion Recording Protocol

The P-assertion Recording Protocol (PReP) defines the communication between actors and the expected behaviour of those actors when recording p-assertions. In this case, PReP has the following benefits:

1) It ensures that the data residing in the provenance store is high-quality.
2) It provides a well-defined interface for actors to record p-assertions.

We note that the protocol is abstract and acts as a guideline for implementations. Thus, the abstract specification does not directly address implementation performance. However, the protocol has been designed to enable implementations that minimize impact on application performance. In particular, an extensive investigation of an implementation of PReP in both bioinformatics and controlled environments demonstrated a maximum overhead of between ten and fifteen percent [10].

We now present the protocol and corresponding definitions of actor behaviour that together ensure the recording of process documentation with high quality characteristics. We begin with a specification of the protocol.

PReP is an abstract asynchronous protocol for an actor to record p-assertions into a provenance store. An asynchronous protocol allows actors to send their messages at any time, which means that actors can choose when to record p-assertions and thus not delay their execution. Furthermore, this asynchronous approach enables implementations of the protocol to minimise performance perturbations by selecting when recording will have minimal impact on the application. We define PReP in terms of both its messages as well as the expected behaviour of the actors exchanging the protocol messages.

With the previously listed characteristics in mind, we now proceed to define the protocol itself. As we defined earlier, an application can be described as actors communicating via message passing. In such a system, communication between actors gives a context to the individual execution of actors, hence, we base the protocol around the notion of an *interaction*. Actors record p-assertions in the context of a given interaction where they are either a sender or receiver. Note an actor may be involved in many different interactions and that its lifetime is application dependent.

[2]Examples of provenance-aware systems include workflow enactment engines that capture traces of workflow execution [24], databases that allow the retrieval of a tuple's derivation [5] and file systems that capture how programs manipulate files [14].

A. Messages

With these general notions, we now present the protocol's messages and those assumed by it. After describing the messages, we then present the dependencies between them.

Let us consider the example of a distributed application that is not provenance-aware, i.e. one in which there are no provenance stores and p-assertions are not recorded. In such a case, application actors communicate via application messages that contain some application specific data. Figure 1 shows a *basic application message* that has one parameter that consists of an element from the set DATA. The parameter refers to the data typically transmitted by senders and receivers irrespective of p-assertion recording. Each message named in Figure 1 is given a notation that is used later in a formalisation. Likewise, the parameters of each message, which are defined by sets, are used in the same formalisation.

Name	Notation	Parameters
basic application message		DATA
application message	app	IK, DATA
record p-assertion	rec	IK, RI, A, LPID, P-ASSERTION
view size	vs	IK, RI, A, LPID, $\mathbb{N}^+$
acknowledgement	ack	IK, RI, LPID, $\mathbb{B}$

Fig. 1. The messages of PReP (IK = Interaction Key, RI = Role Identifier, LPID = Local P-assertion Id, A = Actor Identity, $\mathbb{N}^+$ = number of p-assertions, $\mathbb{B}$ = whether the message was stored)

To transform such an application into a provenance-aware application, one or more provenance stores are introduced and actors record p-assertions into them. Additionally, basic application messages are extended with an identifier (similar to a message id) to be exchanged between actors, which is shown in Figure 1 as an *application message*. We label the identifier an interaction key.

An interaction key identifies an interaction uniquely from all other interactions. The sender in an interaction is required to generate this key and send it to the interaction receiver. We note that it may not always be possible to extend application messages to add an interaction key. Instead, out of band mechanisms would be required to propagate similar information, for example, by sending the interaction key in a separate provenance-specific message or relying on the message layer to provide unique message ids. During the following discussion, we assume that an interaction key can be added to basic application messages. The generation of interaction keys by senders supports a decentralised design where no centralised entity is necessary for senders or receivers to create or record p-assertions. The set of interaction keys is denoted by IK.

In a provenance-aware application, *application messages* define the messages exchanged by application actors. The rest of the messages defined in Figure 1 are exchanged between recorders and provenance stores.

The *record p-assertion* message is sent by a recording actor to a provenance store in order to record a p-assertion (the set of p-assertions is P-ASSERTION) about an interaction. Each p-assertion is given a key that allows for each p-assertion to be

uniquely identified. This key consists of the following parts: the interaction key the p-assertion is associated with, the role identifier (element of set RI), and the local p-assertion id (element of set LPID). The interaction key combined with the role identifier and local p-assertion id ensures that every p-assertion can be referenced uniquely. The role identifier specifies whether the actor recording the p-assertion was the sender or the receiver in the interaction. The local p-assertion id is a nonce generated by the actor through a pseudo-function that increments a local counter (LC) for every p-assertion created by the actor.

The *record p-assertion* message also includes an actor identity (element of set A) that is the asserter identity. The asserter identity is the creator of the p-assertion within the message and is essential for recording attributable process documentation. It connects the p-assertion in the message to the identity of the actor that creates, records, and is responsible for it.

To allow an actor to inform the provenance store about how many p-assertions should be recorded for a particular interaction, we introduce the *view size* message. It is similar to the *record p-assertion* message, except that the p-assertion parameter is replaced with an integer representing how many p-assertions a provenance store should receive in total from an actor for an interaction. By knowing how many p-assertions should be recorded, a provenance store can determine when an actor has finished recording and thus allow process documentation to be finalizable. The *view size* message, like any other message, can be sent at any time. We note that in most cases an actor will send this message *after* it has recorded all its p-assertions. Thus, the *view size* does not imply that an actor guesses how many p-assertions it will create and record for a particular interaction. Instead, the ability to send the message at any time preserves the asynchronicity of the protocol. For example, consider an actor that has already created all its p-assertions, because of PReP's design, the actor can record all the p-assertions and notify the provenance store about number of p-assertions contained within a view in parallel.

The last kind of message exchanged by recorders and provenance stores is the *acknowledgement* message. Each message received by a provenance store is acknowledged by an *acknowledgement* message, which contains the p-assertion key contained in the message being acknowledged as well as a boolean value denoting whether the message was stored. There is some computation time in processing *record* messages and storing their contents. Therefore, *acknowledgement* messages allow actors to track whether their p-assertions have been stored within the provenance store. This is useful when the actor wishes to notify other actors that it has completed recording. Furthermore, the provenance store can use the acknowledgement message to notify the actor that the store can be queried or to return error messages and other implementation specific information. We assume that flow control and reliable message delivery are handled by the underlying communication protocol. Thus, the acknowledgement message is not used for guaranteeing message delivery or flow control but, instead for the recorder to track the state of the provenance store.

We now describe the dependencies between the messages defined above. Due to the asynchronous nature of the protocol, the dependencies are minimal. They are as follows:

- For any *application message* in a given interaction, a *record p-assertion* or *view size* message about that interaction must contain the same interaction key as the *application message*.
- *Acknowledgement* messages must be sent after the receipt of the message that is being acknowledged.

B. Behaviour

The set of messages and their dependencies impose some behavioural constraints on the roles of sender, receiver, recorder, and provenance store. We now make such behaviour explicit. Our intent here is to give an intuitive description of the required behaviour of these actors and then use the formalisation that follows to give a precise definition of that behaviour. We enumerate these **behaviour constraints** below.

1) (Unique Interaction Key Rule) A sender must generate a globally unique interaction key for every new interaction and associate it to that interaction.
2) (Interaction Key Transmission Rule) A sender must send the interaction key to the receiver by including it within the application message being sent.
3) (Appropriate Interaction Rule) Both receiver and senders must use the interaction key associated with an interaction, I, when asserting p-assertions about I.
4) A recorder must keep track of the messages it has sent to a provenance store for a particular interaction until the acknowledgements are received for them. This behaviour models how actors can track the state of the provenance store.
5) The provenance store must be waiting to receive messages and when it receives a message, it must process its content, store it and return the appropriate acknowledgement message. The provenance store must prevent previously recorded p-assertions from being overwritten and not allow additional p-assertions to be added to complete Views.

We now present a formal model of PReP.

V. A Formal Model of PReP

To show that PReP records process documentation with the characteristics listed in Section II, we now present a formalisation of PReP in terms of the behaviour of the actors involved in the protocol and the messages used. We have chosen to model PReP as an abstract state machine (ASM) because it provides a precise, implementation-independent means of describing the protocol. The notation we use is a textual representation of the calculus of constructions adopted by Coq and was used to describe a distributed reference counting algorithm [16]. The abstract machine characterises the behaviour of actors with respect to the messages they send and receive. This behaviour is specified by the permissible transitions that the ASM is allowed to perform. We begin by describing the state space of the ASM, and we then proceed to discuss its transitions.

1) State Space: The state space of the ASM is shown in Figure 2. We model a distributed system as a set of actors communicating via asynchronous message passing over a set of communication channels, $\mathcal{K}$. We identify specific subsets of actors in the system, namely, senders, receivers and provenance stores. An actor may be a member of all these subsets. These subsets map to roles previously defined. Actors are referred to using an Actor Identity, A.

Communication channels are assumed to be reliable and secure, and not to duplicate messages. No assumption is made about message order in the channel (i.e. sending message A before sending message B does not guarantee that A will arrive at its destination before B). Hence, channels are represented as bags of messages between pairs of actors. We note that the dependencies between messages (e.g. that an acknowledgement is sent after the receipt of a message) are not enforced by channels but by actors following their specified behaviour. While PReP does not require message order, applications are free to add such a requirement and capture that fact in the process documentation that the application generates. The messages listed in Figure 1 are sent over these communication channels and are formally defined as an inductive type producing set $\mathcal{M}$ in Figure 2.

Having defined the state space for communication between actors, we now model the state space for the internal functionality of each actor role.

a) Provenance Store State Space: Informally, we can see a provenance store as an actor containing a table that maps interaction keys and a role identifier to a set of identified messages. This models the Views within a provenance store (V). An interaction key (κ) together with a role identifier (v) and set of identified messages is labelled a View. In Figure 2, the table ($store\_T$) is defined as a function that takes an interaction key and role identifier and returns a triple containing a view size message, several record messages and some local p-assertion ids. We use the power set notation ($\mathbb{P}$) to denote that there can be more than one of a given element. We define the set of provenance stores, PSS, as a mapping from an actor identity to a set of Views. Since each set of Views can be located at a different actor, our model allows for multiple provenance stores.

b) Sender and Receiver State Space: Now, we define the state space of sending and receiving actors in Figure 2. This state space describes the various tables that these actors use to keep track of the messages they need to send to the provenance store (TO_SEND), the messages they have sent to it (SENT) and the acknowledgements received from it (ACK). Furthermore, the state space describes the p-assertions that a sender or receiver need to record in a provenance store (ASSERT) and how an actor keeps track of the local p-assertion ids it has already used (LPID_MAP). Finally, each sending actor has a local counter (LC) used to create interaction keys.

The state space that we have described may appear to be global in Figure 2. However, each table for a sender or receiver is indexed by an actor identity (A) and can be implemented with updates that are local to actors. Hence, the protocol does not require any global knowledge by actors of other actors' state.

For convenience, we define two accessor functions. The accessor function to access the state of the View is defined as follows:

If $store\_T(a)(\kappa, v) = \langle vs, recs, lpids \rangle$ then
$store\_T(a)(\kappa, v).vs = vs$,
$store\_T(a)(\kappa, v).recs = recs$,
$store\_T(a)(\kappa, v).lpids = lpids$

We also define a function for accessing the state of a view size message. The function is defined as follows:

If $vs = \mathsf{vs}(\kappa, v, a, \ell, na)$ then
$vs.\kappa = \kappa$, $vs.v = v$, $vs.a = a$,
$vs.\ell = \ell$,$vs.na = na$

Having described the state space of our ASM, a state (or configuration) of the machine is described in Figure 2. The machine's initial state, shown in the same figure, can be summarised as: empty interaction record stores, empty communication channels, all sending and receiving actors having empty p-assertion and message tables, and local counters being initialized to zero. We use λ-calculus notation to represent tables as functions in the initial state.

The machine proceeds from this initial state through its execution by going through *transitions* that lead to new states. These transitions are defined by the rules of the state machine discussed in the next section.

When describing the execution of a state machine, we use the following notation and definitions.

- A *transition* is the application of a rule to one configuration to achieve another configuration.
- A *reachable configuration* is a configuration of the ASM that can be reached by transitions from the initial configuration.
- $\longmapsto$ denotes a transition.
- $c \longmapsto^* c'$ denotes any number of transitions from a configuration c to another configuration c'.

We now discuss the specific rules of the ASM.

2) State Machine Rules: The permissible transitions in the ASM are described through rules, which are represented using the following notation.

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

Rules are identified by their name and the parameters that they operate over. Rules are configuration transformers; converting any configuration satisfying all its preconditions into a new configuration by applying all the rule's pseudostatements. There is a nondeterministic selection of rules when several can be fired at the same time. This does not impact the eventual recording of p-assertions in the provenance store as shown later in Section VI-E.

We use *send*, *receive* and 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. Likewise, the table update operation

$$
\begin{array}{rcll}
\text{A} & = & \{a_1, a_2, \ldots, a_n\} & \text{(Set of Actor Identities)} \\
\text{SENDERS} & \subseteq & \text{A} & \text{(Set of Sender Identities)} \\
\text{RECEIVERS} & \subseteq & \text{A} & \text{(Set of Receivers Identities)} \\
\text{PS} & \subseteq & \text{A} & \text{(Set of Provenance Store Identities)} \\
\text{REL} & = & \{r_1, r_2, \ldots, r_n\} & \text{(Set of Business Logic Descriptions)} \\
 & & & \\
\text{P-ASSERTION} & = & \{\alpha_1, \alpha_2, \ldots\} & \text{(Set of P-Assertions)} \\
 & & & \\
\mathcal{M} & = & \mathsf{app} : \text{IK} \times \text{DATA} \rightarrow \mathcal{M} & \text{(Set of Messages)} \\
 & & \quad | \ \mathsf{rec} : \text{IK} \times \text{RI} \times \text{A} \times \text{LPID} \times \text{P-ASSERTION} \rightarrow \mathcal{M} & \\
 & & \quad | \ \mathsf{vs} : \text{IK} \times \text{RI} \times \text{A} \times \text{LPID} \times \mathbb{N}^+ \rightarrow \mathcal{M} & \\
 & & \quad | \ \mathsf{ack} : \text{IK} \times \text{RI} \times \text{LPID} \times \mathbb{B} \rightarrow \mathcal{M} & \\
 & & & \\
\text{VS} & = & \{m \in \mathcal{M} \mid m = \mathsf{vs}(\kappa, v, a, \ell, na)\} & \text{(Set of View Size Messages)} \\
\text{R} & = & \{m \in \mathcal{M} \mid m = \mathsf{rec}(\kappa, v, a, \ell, \alpha)\} & \text{(Set of Record Messages)} \\
 & & & \\
\text{IK} & = & \text{SENDERS} \times \text{RECEIVERS} \times \mathbb{N} & \text{(Set of Interaction Keys)} \\
\text{RI} & = & \{\mathsf{S}, \mathsf{R}\} & \text{(Set of Role Identifiers)} \\
 & & & \\
\text{V} & = & \text{IK} \times \text{RI} \rightarrow \text{VS}_\perp \times \mathbb{P}(\text{R}) \times \mathbb{P}(\text{LPID}) & \text{(Set of Views)} \\
\text{PSS} & = & \text{A} \rightarrow \mathbb{P}(\text{V}) & \text{(Set of Provenance Stores)} \\
 & & & \\
\text{TO\_SEND} & = & \text{A} \rightarrow \text{IK} \rightarrow Bag(\mathcal{M}) & \text{(Set of Messages To Send Tables)} \\
\text{SENT} & = & \text{A} \rightarrow \text{IK} \rightarrow Bag(\mathcal{M}) & \text{(Set of Sent Messages Tables)} \\
\text{ACK} & = & \text{A} \rightarrow \text{IK} \rightarrow Bag(\mathcal{M}) & \text{(Set of Acknowledged Messages Tables)} \\
\text{ASSERT} & = & \text{A} \rightarrow \text{IK} \times \text{RI} \rightarrow Bag(\text{P-ASSERTIONS}) & \text{(Set of p-assertions to be recorded)} \\
\text{LPID\_MAP} & = & A \rightarrow \text{IK} \times \text{RI} \rightarrow \mathbb{P}(\text{LPID}) & \text{(Map from actor to local p-assertion ids)} \\
\text{LC} & = & \text{SENDERS} \rightarrow \mathbb{N} & \text{(Set of Local Counters)} \\
 & & & \\
\mathcal{K} & = & \text{A} \times \text{A} \rightarrow Bag(\mathcal{M}) & \text{(Set of Channels)} \\
 & & & \\
\text{C} & = & \text{PSS} \times \mathcal{K} \times \text{TO\_SEND} \times \text{SENT} \times & \text{(Set of Configurations)} \\
 & & \text{ACK} \times \text{ASSERT} \times \text{LPID\_MAP} \times \text{LC} &
\end{array}
$$

Characteristic Variables:

$$
\begin{array}{rcl}
a & \in & A \\
a_s & \in & \text{SENDER} \\
a_r & \in & \text{RECEIVER} \\
a_{ps} & \in & \text{PS} \\
r & \in & \text{REL} \\
m & \in & \mathcal{M} \\
d & \in & \text{DATA} \\
\alpha & \in & \text{P-ASSERTION} \\
\kappa & \in & \text{IK} \\
v & \in & \text{RI} \\
na & \in & \mathbb{N}^+ \\
\ell & \in & \text{LPID}
\end{array}
\qquad
\begin{array}{rcl}
k & \in & \mathcal{K} \\
lpids & \in & \mathbb{P}(\text{LPID}) \\
recs & \in & \mathbb{P}(R) \\
store\_T & \in & \text{PSS} \\
to\_send\_T & \in & \text{TO\_SEND} \\
sent\_T & \in & \text{SENT} \\
ack\_T & \in & \text{ACK} \\
assert\_T & \in & \text{ASSERT} \\
lpid\_T & \in & \text{LPID\_MAP} \\
lc & \in & \text{LC} \\
c & \in & \text{C} \\
b & \in & \mathbb{B}
\end{array}
$$

Initial State / Configuration:

$c_i = \langle store\_T_i, k_i, to\_send\_T_i, sent\_T_i, ack\_T_i, assert\_T_i, lpid\_T_i, lc_i \rangle$
where:

$$
\begin{array}{rcl@{\qquad}rcl}
store\_T_i & = & \lambda a_i \lambda \kappa_i v_i . \langle \perp, \emptyset, \emptyset \rangle, & k_i & = & \lambda a_i a_j . \emptyset, \\
to\_send\_T_i & = & \lambda a_i \kappa_i . \emptyset, & sent\_T_i & = & \lambda a_i i k_i . \emptyset, \\
ack\_T_i & = & \lambda a_i i k_i . \emptyset, & assert\_T_i & = & \lambda a_i \kappa_i v_i . \emptyset, \\
lpid\_T_i & = & \lambda a_i \kappa_i v_i . \emptyset, & lc_i & = & \lambda a_i . 0
\end{array}
$$

Fig. 2. State Space

puts a message into a table. The notation $table\_T$ is used to refer to any table in the state space. Formally, these pseudo-statements act as state transformers and are defined as follows.

- We use the operators $\oplus$ and $\ominus$ to denote union and difference on bags. If k is the set of message channels of a state $\langle \ldots, k, \ldots \rangle$, then the expression $send(a_1, a_2, m)$ and $receive(a_1, a_2, m)$ respectively denote the state $\langle \ldots, k', \ldots \rangle$, where $k'(a_1, a_2) = k(a_1, a_2) \oplus \{m\}$ and $k(a_1, a_2) \ominus \{m\}$, 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 $table\_T.y := V$ denotes the state $\langle \ldots, table\_T', \ldots \rangle$, where $table\_T.x = table\_T'.x$ if $x \neq y$, and $table\_T'.y = V$.

To ease the readability of the rules, we also define the following functions. The function *complete* determines if a View is complete:

$complete : \text{A} \times \text{IK} \times \text{RI} \rightarrow \{true, false\}$
$complete(a, \kappa, v) :=$
  If $store\_T(a)(\kappa, v).vs \neq \perp$,
  then return
    $store\_T(a)(\kappa, v).vs.na =$
    $|store\_T(a)(\kappa, v).recs|$
  else return *false*.

The pseudo function *newIdentifier* creates new interactions keys, updating an actor's local counter table:

$newIdentifier : \text{SENDERS} \times \text{RECEIVERS} \rightarrow \text{IK}$
$newIdentifier(a_s, a_r) :=$
  $lc(a_s) := lc(a_s) + 1$
  return $\langle a_s, a_r, lc(a_s) \rangle$.

Figures 3 and 4 show the ASM's transition rules, which formally define the behaviour of the actors in PReP that meet the constraints described in Section IV-B. We now discuss the two rules that govern the provenance store's behaviour shown in Figure 3.

a) Provenance Store Rules: The $receive\_record\_passertion$ rule takes an incoming record message from a particular actor and places it in the correct View in the provenance store as defined by $store\_T$, and thereby also stores the p-assertion enclosed in the message. The View is looked up via the interaction key and role identifier located in the rec message. An acknowledgement message (ack) is then sent to the actor who sent the rec message. During its execution, the rule checks to see whether or not the local p-assertion id (ℓ) of the p-assertion contained within the message has already been used in the interaction record. If ℓ has not been used, the message is stored,

```
1: receive_record_passertion(a, a_ps, κ, v, ℓ, α) :
2:     rec(κ, v, a, ℓ, α) ∈ K(a, a_ps)
   → {
3:       receive(rec(κ, v, a, ℓ, α), a, a_ps);
4:       if (ℓ ∉ store_T(a_ps)(κ, v).lpids∧
           ¬complete(a_ps, κ, v)), then
5:         store_T(a_ps)(κ, v).lpids :=
               store_T(a_ps)(κ, v).lpids ∪ {ℓ};
6:         store_T(a_ps)(κ, v).recs :=
               store_T(a_ps)(κ, v).recs ∪ {rec(κ, v, a, ℓ, α)};
7:         send(ack(κ, v, ℓ, true), a_ps, a);
8:       else send(ack(κ, v, ℓ, false), a_ps, a);
       }

9: receive_view_size(a, a_ps, κ, v, ℓ, na) :
10:    vs(κ, v, a, ℓ, na) ∈ K(a, a_ps)
   → {
11:      receive(vs(κ, v, a, ℓ, na), a, a_ps);
12:      if(ℓ ∉ store_T(a_ps)(κ, v).lpids ∧
           store_T(a_ps)(κ, v).vs = ⊥), then
13:        store_T(a_ps)(κ, v).lpids :=
               store_T(a_ps)(κ, v).lpids ∪ {ℓ};
14:        store_T(a_ps)(κ, v).vs := vs(κ, v, a, ℓ, na);
15:        send(ack(κ, v, ℓ, true), a_ps, a);
16:      else send(ack(κ, v, ℓ, false), a_ps, a);
       }
```

Fig. 3. Provenance Store rules

otherwise it is not. Likewise, if the View is complete (i.e. it already contains the requisite number of p-assertions) the message is not stored. In all cases, an acknowledgement is sent informing the actor whether the p-assertion was stored (true if it was, false otherwise). This rule satisfies part of Behaviour Constraint 5 (See Section IV-B).

The *view_size* rule operates in a similar manner and satisfies the rest of Behaviour Constraint 5. However, only one view size message (vs) is allowed to be recorded. If subsequent vs messages are received by the provenance store, the message is discarded and an ack message is sent to the actor.

b) Sender and Receiver rules: Figure 3 defined the behaviour of a provenance store. However, in order to demonstrate the protocols support for high-quality characteristics, we need to prescribe some behaviour of the actors that use provenance stores. Our aim is to show that if an actor behaves in the manner defined, then certain guarantees can be given. This behaviour is formalised in Figure 4.

The functionality of these rules can be summarised as follows. When involved in an interaction (i.e. either sending or receiving a message), an actor makes p-assertions related to that interaction. This behaviour is shown in the *send_app_msg* and *receive_app_msg* rules. We highlight the *makeAssertions* function which takes an actor identity (a), an interaction key (κ), some data (d), and a business logic description (r) and produces a set of p-assertions, $\{\alpha_1, \ldots, \alpha_n\}$. This function is defined by each protocol implementation, which is responsible for generating p-assertions describing the contents of messages exchanged in interactions as well as the business logic (i.e., application functionality) operating on data within those messages. Thus, by using business logic descriptions, the *makeAssertions* function not only allows message contents to be recorded but also the dependencies between those messages to be captured. In other work [11], we describe how documentation produced by the *makeAssertions* function can be organized to effectively capture such dependencies and thus

```
1: send_app_msg(a_s, a_r, d, r) :
   // triggered when d, produced by a function described by r,
   // is to be sent by a_s to a_r
   → {
2:       let κ = newIdentifier(a_s, a_r)
3:       in  let {α_1, ..., α_n} = makeAssertions(a_s, κ, d, r)
4:           in  send(app(κ, d), a_s, a_r);
5:               assert_T(a_s, κ, S) :=
                     assert_T(a_s, κ, S) ∪ {α_1, ..., α_n};
     }

6: receive_app_msg(a_s, a_r, κ, d) :
7:     app(κ, d) ∈ K(a_s, a_r)
   → {
8:       receive(app(κ, d), a_s, a_r);
         // receive business logic
9:       let {α_1, ..., α_n} := makeAssertions(a_r, κ, d, ⊥)
10:      in  assert_T(a_r, κ, R) :=
                 assert_T(a_r, κ, R) ∪ {α_1, ..., α_n};
     }

12: make_passertion_msg(a, α, κ, v) :
13:    α ∈ assert_T(a, κ, v)
   → {
14:      let ℓ ∉ lpid_T(a, κ, v)
15:      in  to_send_T(a, κ) :=
                 to_send_T(a, κ) ∪ {rec(κ, v, a, ℓ, α)};
16:          assert_T(a, κ, v) := assert_T(a, κ, v) ⊖ α;
17:          lpid_T(a, κ, v) := lpid_T(a, κ, v) ⊕ ℓ;
     }

18: make_view_size_msg(a, κ, v) :
19:    assert_T(a, κ, v) = ∅
   → {
20:      let ℓ ∉ lpid_T(a, κ, v)
21:      in  to_send_T(a, κ) :=
             to_send_T(a, κ) ∪ {vs(κ, v, a, ℓ, |lpid_T(a, κ, v)|)};
22:          lpid_T(a, κ, v) := lpid_T(a, κ, v) ⊕ ℓ;
     }

23: record_message(a, a_ps, m) :
24:    m ∈ to_send_T(a, κ)
   → {
25:      to_send_T(a, κ) := to_send_T(a, κ) ⊖ m;
26:      sent_T(a, κ) := sent_T(a, κ) ⊕ m;
27:      send(m, a, a_ps);
     }

28: receive_ack(a, a_ps, κ, v, ℓ, b) :
29:    ack(κ, v, ℓ, b) ∈ K(a_ps, a)
   → {
30:      receive(ack(κ, v, ℓ, b), a_ps, a);
31:      ack_T(a, κ) := ack_T(a, κ) ⊕ ack(κ, v, ℓ, b);
     }
```

Fig. 4. The rules of the ASM used by sending and receiving actors

aid in the determination of provenance.

Once the set of p-assertions has been created, a rec message for each p-assertion is then constructed and added to the table of messages to be sent to the provenance store (to_send_T), which is shown in *make_passertion_msg*. This rule also obtains the asserter identity. When all the rec messages have been added to to_send_T, a vs message is constructed . The vs message could be constructed at any time, but we have chosen to do so at this stage to make the rules simpler. Creating vs messages in this manner is a lazy strategy, they could alternatively be generated using an eager strategy. These steps are formalised in *make_view_size_msg*. These two rules both keep track of the local p-assertion ids that the actor has used.

Once a message has been sent, it is added to a table of messages that have been sent to the provenance store, $sent\_T$, and removed from to_send_T. Finally, when an acknowledgement has been received, it is stored in the table ack_T. This models how an actor keeps track of its communication with a provenance store for a particular interaction identified by

κ. The rules that correspond to this tracking functionality are *record_msg* and *receive_ack* and satisfy Behaviour Constraint 4.

VI. Protocol Analysis

Based on the ASM above, we now analyse PReP according to the high-quality characteristics presented in Section II. The analysis shows how these broad characteristics are reflected directly by concrete properties supported by PReP. Several properties are *invariants* i.e. as the state machine proceeds these properties always stay the same. We rely on case analysis either on its own or in the context of a proof by induction to establish these properties. Essentially, the rules of the ASM are analysed to show that after any number of transitions the particular property still holds for the resulting configuration of the state machine.

When using proof by induction, induction is performed upon the length of the transition sequence from one configuration to another. In the base case, we establish that the invariant holds in some initial configuration. In the inductive case, the invariant is assumed to hold after a series of transitions from the initial configuration to another configuration. We then prove, using case analysis, that the property holds after an additional transition. We now begin the analysis.

A. Immutable

We demonstrate that PReP supports the storage of immutable process documentation by showing that p-assertions that have been previously recorded will not be overwritten, deleted or modified. We begin by defining a function that retrieves the record messages from a View in a particular configuration.

Definition 1 (View Function): For all c, a_{ps}, κ, v, $View(c, a_{ps}, \kappa, v)$ is defined as $store\_T(a_{ps})(\kappa, v).recs$ where $store\_T$ is in the configuration c.

Using this function, we define the following invariant, stating that a view's contents are monotonically increasing.

Invariant 1 (Monotonically Increasing): For any configurations c_1 and c_2, where $c_1 \longmapsto^* c_2$,

$$View(c_1, a_{ps}, \kappa, v) \subseteq View(c_2, a_{ps}, \kappa, v)$$

for any a_{ps}, κ, v.

Proof: We prove this invariant by induction on the length of the transition sequence $c_1 \longmapsto^* c_2$. In the base case, where the length is zero, c_1 equals c_2 and the invariant holds.

In the inductive case, we assume that for $c_1 \longmapsto^* c_n$ the invariant holds: therefore, $View(c_1, a_{ps}, \kappa, v) \subseteq View(c_n, a_{ps}, \kappa, v)$. We then consider the possible transitions from $c_n \longmapsto c_2$. There are only two rules of the state machine that govern the actions of the provenance store: $receive\_record\_passertion$ and $receive\_view\_size$ shown in Figure 3.

In the case of the $receive\_record\_passertion$, $store\_T$ is only added to as shown in Figure 3 lines 5 and 6. Moreover, if the same local p-assertion id is used by a record message, the message is discarded and an acknowledgement is sent. This test is shown in Figure 3 line 4. Hence, $View(c_n, a_{ps}, \kappa, v) \subseteq View(c_2, a_{ps}, \kappa, v)$.

Likewise, in the case of the $receive\_view\_size$ rule, if a vs message has already been recorded for a given View, then no other vs message is allowed to be recorded as seen in Figure 3 line 12. Furthermore, there are no rules that operate on $store\_T$ that delete or modify already recorded p-assertions. Hence,

$$View(c_n, a_{ps}, \kappa, v) \subseteq View(c_2, a_{ps}, \kappa, v).$$

By transitivity, we conclude that

$$View(c_1, a_{ps}, \kappa, v) \subseteq View(c_2, a_{ps}, \kappa, v).$$

For all other rules, the provenance store is not updated; hence, $c_n = c_2$, which completes the proof. ■

Because the contents of the provenance store are monotonically increasing, PReP ensures that the evidence of a process is immutable, which gives confidence to both users and creators of process documentation. Users know that process documentation is the same as it was when it was originally recorded by the source. Likewise, creators know that process documentation they have entrusted to the provenance store will not disappear or be corrupted. Finally, immutable process documentation ensures that users will not be caught off-guard by its accidental or non-malicious deletion.

B. Attributable

PReP supports attributable process documentation by ensuring that for every p-assertion in a provenance store there is an actor identity for that p-assertion that identifies the p-assertion's asserter, which means that every p-assertion can be "tracked back" to its creator. Thus, actors can be held accountable for the p-assertions they create, by a means outside PReP. Again, we note that, within PReP, the creator and recorder of a p-assertion are the same actor. We begin by defining the following invariant, which states that the actor identity within a message does not change.

Invariant 2 (Identity Preserving): For all configurations, c_1, c_2, where $c_1 \longmapsto^* c_2$, for any message m in c_1, the actor identity in m is the same in c_2, for all $m \in R \cup VS$.

The proof can be found in Appendix A, which follows the same proof procedure as Invariant 1. Based on the Identity Preserving invariant, the following lemma can be shown. Its proof can also be found in the same appendix.

Lemma 1: An actor identity contained within a given rec message is always the identity of the actor that caused the generation of the rec message.

Based on Lemma 1, Invariant 2, and Invariant 1, we conclude that if a p-assertion is in a provenance store then the identity of the actor who created and recorded the p-assertion can be determined. Note, that p-assertions only appear within rec messages inside the provenance store. Hence, every p-assertion is attributed to a particular actor.

C. Autonomously Creatable

To ensure that actors are able to create and record p-assertions at their convenience in an autonomous fashion

without having to synchronise with any other actors, PReP specifies three behaviour constraints specific to sender and receivers. When these behaviour constraints are followed, the sender and receiver are able to create and record p-assertions independently from one another while still ensuring consistency within the provenance store. These behaviour constraints are identified in the following definition:

Definition 2 (Actor Behaviour Compliant): Two actors passing a single message are actor behaviour compliant when they follow the Unique Interaction Key Rule, the Interaction Key Transmission Rule, and the Appropriate Interaction Rule.

If actors conform to the state machine, they will be actor behaviour compliant as shown by the case analysis given in Appendix B.

D. Finalizable

We now show that process documentation recorded by PReP is finalizable. Using the view size message, an actor can state that its account of an interaction is fully documented. This notion is captured in by a complete View. Once a View is complete, it can no longer be changed. This is stated formally in the following invariant, which is proven in Appendix C.

Invariant 3 (Complete View): Consider a configuration, c_1, where a View is complete. For any configuration c_2, such that $c_1 \longmapsto^* c_2$, then

$$View(c_1, a_{ps}, \kappa, v) = View(c_2, a_{ps}, \kappa, v),$$

for all a_{ps}, κ, v.

From Invariant 3, process documentation recorded by PReP in a View can be marked as complete and is then immutable. Hence, the process documentation recorded by PReP has the high-quality characteristic of being finalizable, which seals a View preventing future information from being added, which gives users a firm basis for making a judgement about the interaction the View documents; they know that no new information will suddenly arise. Furthermore, it allows a system to determine when an actor is finished recording process documentation for an interaction and thus when the provenance store can be queried about that interaction.

E. Process Reflecting

We now show that a process (i.e. the execution of an application) can be reflected by process documentation recorded in a provenance store.[3] To do this, we extend the ASM to consider the execution of actors. The actors execute in accordance with the following definition of process:

Definition 3 (Interaction Process Definition): A process is a connected set of interactions and transformations.

Thus, the execution of actors is described by the exchange of messages between actors and the transformations they perform on received messages. Transformations are defined formally later in Definition 4. To describe this execution formally, we first define the state space for the execution of the set of actors.

As all the data that an actor works upon is located in received messages, we model the state of actor (AS) as data paired with the interaction key from the message in which the data was received. We assume that a garbage collector will collect any unused data, but, for simplicity, we do not model garbage collection here. We also define a table, APPS, that maps from actor identities (A) to actor states. The same accessor notation that we have used for other tables applies to APPS as well. So that $as(a)$ will access the state of the actor identified by a. The configuration of the application is defined by the combination of the state of all the actors in the system combined with the configuration of PReP. This is modelled by PAPPS.

[3]By reflection, we mean common sense reflection. We do *not* mean introspection as offered by some programming language runtime environments.

32: $consume\_msg(a_s, a_r, \kappa, d)$:
33: $\mathsf{app}(\kappa, d) \in \mathcal{K}(a_s, a_r)$
$\rightarrow$ {
34: $receive(\mathsf{app}(\kappa, d), a_s, a_r)$;
35: $as(a_r) := as(a_r) \cup \{\langle d, \kappa\rangle\}$;
// receive business logic
}

36: $produce\_msg(a_s, a_r, d, f)$:
// triggered when d, produced by function f
// that is described by r,
// is to be sent by a_s to a_r
$\rightarrow$ {
37: let $\kappa = newIdentifier(a_s, a_r)$
38: in $send(\mathsf{app}(\kappa, d), a_s, a_r)$;
}

AS	=	$\mathbb{P}(\text{DATA} \times \text{IK})$	(Actor States)
APPS	=	$\text{A} \rightarrow \text{AS}$	(Application State)
PAPPS	=	$\text{C} \times \text{APPS}$	(Provenance-aware) Application State)

Characteristic Variables:

$\langle d, \kappa\rangle \in$ AS, $as \in$ APPS, $pas \in$ PAPPS, $\langle c, as\rangle = pas$

Fig. 5. Application Rules and State Space

The execution of actors following Definition 3 can be modelled by rules shown in Figure 5 that express the transition of states when sending and receiving messages.

Thus, after the receipt of a message, the state of the application is updated, whereas, after sending a message, the state stays the same. Sending a message is triggered by the execution of some business logic or a transformation on the actor state, which we define as follows.

Definition 4 (Transformation): A transformation is the execution of a function f on an actor state, $as(a)$, to achieve a tuple $\langle d, a_r\rangle$. This is represented as $\langle d, a_r\rangle = f(as(a))$ where a_r specifies the actor to which the data item should be sent.

The functions applied to actor states are modelled by business logic definitions, which are specified by the set REL defined in Figure 2.

The execution of some application is modeled by the transitions between actor states denoted by $as_1 \longmapsto^*_{app} as_2$, where the transitions are defined in Figure 5. In our modeling, an application is allowed to execute independently of recording process documentation. We now show how the application can be integrated with PReP. First, we denote three different categories of transitions and provide a notation for each:

1) Provenance Aware Application Transitions are all the transitions that occur in both PReP and an application. These are denoted by $\longmapsto_{paa}$.
2) Application Transitions are the transitions defined in Figure 5. These are denoted by $\longmapsto_{app}$.

3) PReP Transitions are the transitions defined in Figures 3 and 4. These are denoted by $\longmapsto_{prep}$.

The execution of such a provenance-aware application is denoted by $pas_1 \longmapsto^*_{paa} pas_2$. In such an execution, PReP Transitions and Application Transitions are coupled together following Definition 5.

Definition 5 (System Coupling): For any application configurations, $\langle c_1, as_1\rangle$, $\langle c_2, as_2\rangle$, the following transition is allowed:

$$\langle c_1, as_1\rangle \longmapsto_{paa} \langle c_2, as_2\rangle$$

if one of the following conditions hold:

1) $as_1 \longmapsto_{app} as_2$ with $produce\_msg$ and $c_1 \longmapsto_{prep} c_2$ $send\_app\_msg$.
2) $as_1 \longmapsto_{app} as_2$ with $consume\_msg$ and $c_1 \longmapsto_{prep} c_2$ with $receive\_app\_msg$.
3) With transitions other than $send\_app\_msg$ or $receive\_app\_msg$ then $c_1 \longmapsto_{prep} c_2$ and $as_1 = as_2$.

What is occurring is that when the $produce\_msg$ or $consume\_msg$ rules are fired in the application, the corresponding $send\_app\_msg$ or $receive\_app\_msg$ rule are fired as well. Essentially, the application and PReP rules are merged together. Therefore, we note that the pseudo-statements shared by the rules only execute once.

Using this system coupling definition, we can show that process documentation reflecting the application's execution will end up in the provenance store. First, we define the function that creates documentation when an application rule fires. The function is called with $r = \perp$ when $receive\_app\_msg$ fires and with $r \neq \perp$ when $send\_app\_msg$ fires.

Definition 6 (Make Assertions Function):

$makeAssertions(a, \kappa, d, r)$:
PA $= \emptyset$; *// set of p-assertions*
if $r \neq \perp$
  $PA = \{\langle as(a), r, \langle d, \kappa\rangle\rangle\}$
else
  $PA = \{\langle d, \kappa\rangle\}$
return PA;

This $makeAssertions$ function creates p-assertions describing either a transformation executed resulting in a message being sent or the receipt of a message by an actor. In the case where a transformation has been executed, $makeAssertions$ constructs a p-assertion which contains the inputs to the transformation ($ac(a)$ e.g. the state of the actor), the output of the transformation ($\langle d, \kappa\rangle$) and the business logic description of the function, r. When describing the receipt of a message (specified as the argument r begin $\perp$), a p-assertion documenting the incoming data, d, within the interaction identified by κ (i.e. the state being added to the actor state). Process documentation generated by this function reflects the application's actual execution as defined in Definition 3.

Using the above definitions, we provide a proof outline for the following lemma, which is also depicted in Figure 6.

Lemma 2 (Process Reflection): For any application state, as, reachable from as_i, where $as_i \longmapsto^*_{app} as$; for all pas reachable from pas_i: $pas_i = \langle c_i, as_i\rangle \longmapsto^*_{paa} pas$ with $pas = \langle c, as\rangle$; there exists a final configuration $pas_2 = \langle c_2, as\rangle$ for some c_2, where there are no messages in transit and no messages to send, such that $pas \longmapsto^*_{paa} pas_2$ without application transitions, such that the provenance stores in pas_2 contain the description of $as_i \longmapsto^*_{app} as$.

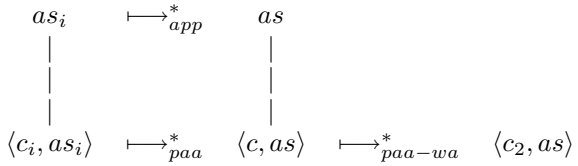

Fig. 6. State transition diagram depicting Lemma 2

Intuitively, the application proceeds from an initial state as_i to some final state as where the application has finished executing. Because of system coupling (shown by the vertical hash lines), the provenance-aware application also proceeds from an initial state $\langle c_i, as_i\rangle$ to some state $\langle c, as\rangle$. However, there may be p-assertions remaining to be recorded that describe the application's execution, thus the provenance-aware application finishes recording those p-assertions without using application transitions (denoted by $paa\text{-}wa$ in the figure). Our proof outline for this lemma again uses induction and is described in Appendix E. The proof outline relies on the guarantee that once a p-assertion is created it will recorded into a provenance store and acknowledge. This guarantee is discussed and proven in Appendix D. If an application executes according to Definition 3 and the $makeAssertions$ function is defined according to Definition 6, then from Lemma 2 documentation reflecting the application's process will eventually be found in provenance stores.

VII. Related Work

The motivation for this paper stems from the need for the provenance of results produced by computational systems. This need has not gone without notice in the literature. Under the heading of lineage, Bose and Frew present a comprehensive overview of provenance related systems [2]. Likewise, Moreau and Foster provide a survey of current provenance research [17]. From an analysis of these works, we assert that the focus of provenance research has been on the implementation of concrete systems for provenance in the context of either specific domains (i.e. geographic information systems, chemistry, biology) or technologies (i.e. databases). In contrast, this paper describes a domain and implementation independent means for recording process documentation for provenance. Based on these surveys, we believe there are no other abstract protocols designed specifically for recording process documentation.[4] Furthermore, our protocol supports the tracing of provenance through general transformations (i.e. actors) that was highlighted as an area that needs to be addressed by Tan [21].

[4]This version of PReP extends previous work on the protocol ([8], [9]) by increasing its generality, clarifying the formalisation, and grounding the work more clearly in use case analyses.

Like PReP, work in the database community has taken a formal approach to provenance. Specifically, the community has focused on the data lineage problem, which can be summarised as: given a data item, determine the source data used to produce that item. Cui *et al.* formalise the problem and present a number of algorithms for determining the lineage of data in relational databases [5]. Buneman *et al.* also develop a formal model of provenance for database systems that applies to both hierarchical and relational databases [3]. PReP differs from these approaches because it considers provenance in open systems where data is distributed and represented using several different kinds of structures and processing happens using a wide variety of algorithms.

PReP is closely related to data synchronization protocols such as rsync [23] and Harmony [7]. Like these protocols, PReP copies data from one actor to another (i.e. a recorder to a provenance store). However, unlike these protocols, PReP is not designed for duplication of data; instead; it is designed for recording data as documentation of a process. Therefore, it provides a context (interactions) to all pieces of data recorded. Furthermore, PReP ensures that all data recorded using is attributable to some actor. The protocols cited do not enforce such attributability.

PReP is also similar to distributed debugging [1], [13] and failure recovery systems [4], [6]. In distributed debugging, the execution of a distributed system is recorded by capturing events in the system. Events are defined by actions of the various actors in the system. These traces of execution can then be used to debug, visualise, or monitor execution [13]. Similarly, failure recovery systems take periodic snapshots of the distributed system state so that execution can be restarted if failure occurs. Unlike these systems, PReP ensures that the documentation recorded using it conforms to five high-quality characteristics. Additionally, PReP specifically enables the capture of business logic descriptions of the functions used within the distributed system.

VIII. Conclusion

The provenance of results produced by distributed systems is important in fields such as aerospace, pharmaceuticals, and finance. We have identified that the provenance of results can be retrieved from documentation of a distributed system's execution. Focusing on the recording of process documentation, we have shown that our recording protocol helps ensure that the process documentation recorded using it complies with the five characteristics necessary for high-quality process documentation. We follow a systematic proof procedure based on mathematical induction. We believe that this systematic approach, while done by hand, is sufficient to provide confidence that the protocol does support the enumerated high-quality characteristics. Furthermore, because of this systematic approach and the extensive use of invariants, we hypothesise that the proofs are at a stage where they could be translated into a format suitable for mechanical proof.

References

[1] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, 1995.

[2] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys*, 37(1):1–28, 2005.

[3] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *Int. Conf. on Databases Theory (ICDT)*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2001.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[6] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[7] J. N. Foster, M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. In *Symposium on Database Programming Languages (DBPL), Trondheim, Norway*, volume 3774 of *Lecture Notes in Computer Science*, pages 42 – 57. Springer-Verlag, 2005.

[8] P. Groth, M. Luck, and L. Moreau. Formalising a protocol for recording provenance in grids. In *Proc. of the UK OST e-Science second All Hands Meeting 2004 (AHM'04)*, Nottingham, UK, September 2004.

[9] P. Groth, M. Luck, and L. Moreau. A protocol for recording provenance in service-oriented grids. In T. Higashino, editor, *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS'04)*, volume 3544 of *Lecture Notes in Computer Science*, pages 124–139, Grenoble, France, December 2004. Springer-Verlag.

[10] P. Groth, S. Miles, W. Fang, S. C. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05)*, 2005.

[11] P. Groth, S. Miles, and L. Moreau. A Model of Process Documentation to Determine Provenance in Mash-ups. *Transactions on Internet Technology (TOIT)*, 2008.

[12] P. Groth, S. Miles, V. Tan, and L. Moreau. Architecture for provenance systems. Technical report, University of Southampton, October 2005. http://eprints.ecs.soton.ac.uk/11310/.

[13] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, 1987.

[14] J. Ledlie, C. Ng, D. A. Holland, K.-K. Muniswamy-Reddy, U. Braun, and M. Seltzer. Provenance-aware sensor data storage. In *NetDB 2005*, April 2005.

[15] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of using provenance in e-science experiments. *Journal of Grid Computing*, 5(1):1–25, 2007.

[16] L. Moreau, P. Dickman, and R. Jones. Birrell's Distributed Reference Listing Revisited. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1344 – 1395, Nov. 2005.

[17] L. Moreau and I. Foster, editors. *Provenance and Annotation of Data — International Provenance and Annotation Workshop, IPAW 2006*, volume 4145 of *Lecture Notes in Computer Science*. Springer-Verlag, May 2006.

[18] L. Moreau, P. Groth, S. Miles, J. Vazquez, J. 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.

[19] A. Schreiber. The integrated simulation environment TENT. *Concurrency and Computation: Practice and Experience*, 14(13-15), January 2003.

[20] C. D. Snow, H. Ngyen, V. S. Pande, and M. Gruebele. Absolute comparison of simulated and experimental protein-folding dynamics. *Nature*, 420:102–106, 2002.

[21] W.-C. Tan. Research problems in data provenance. *IEEE Data Engineering Bulletin*, 27(4):45–52, 2004.

[22] S. Tezuka, H. Murata, S. Tanaka, and S. Yumae. Monte carlo grid for financial risk management. *Future Generation Computer Systems*, 21(5):811–821, 2005.

[23] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, February 1999.

[24] J. Zhao, C. Goble, R. Stevens, and D. Turi. Mining taverna's semantic web of provenance. *Journal of Concurrency and Computation:Practice and Experience*, 2007.

Appendix

In this section, we present proofs of the properties and invariants presented in Section VI.

A. Proofs for Attributable Invariant and Lemma

Invariant 2 (Identity Preserving): For all configurations, c_1, c_2, where $c_1 \longmapsto^* c_2$, for any message m in c_1, the actor identity in m is the same in c_2, for all $m \in R \cup VS$.

Proof: This invariant can be established by induction on the length of invariant by induction on the length of the transition sequence $c_1 \longmapsto^* c_2$.

In the base case, where the transition length is zero, c_1 is equal to c_2 and thus the invariant holds trivially.

In the inductive case, for transition lengths greater than zero, we assume that the invariant holds for $c_1 \longmapsto^* c_n$ and we consider all the possible transitions from $c_n \longmapsto c_2$. There are five rules that deal with messages from the set $R \cup VS$: *receive_record_passertion*, *receive_submission_finished*, *record_message*, *make_passertion_msg*, and *make_view_size_msg*. None of the first three rules create record or view size messages neither do the rules modify them. Thus, the actor identity in c_2 remains the same as in c_n and the invariant holds. In the case where $c_n \longmapsto c_2$ using *make_passertion_msg* or *make_view_size_msg*, the newly created message does not belong to c_n and the invariant holds. ■

Lemma 1: An actor identity contained within a given rec message is always the identity of the actor that caused the generation of the rec message.

Proof: In Figure 4, once a p-assertion is in the $assert\_T$ table, the *make_passertion_msg* rule can fire. In the first pseudo-statement in the rule, a is used to generate the rec message. Furthermore, the rule can only fire if α belongs to $assert\_T$ by a. Therefore, a in the rec message is the identity of the actor that caused the rec message to be generated. A similar argument also applies to the creation of vs messages. Finally, from Invariant 2, we know that these identities remain the same. ■

B. Proof for Actor Behaviour Compliant

Lemma 3 (Compliance Enforcement): Actors following PReP are actor behaviour compliant.

This lemma can be established through a case analysis of the $send\_app\_msg$ and $receive\_app\_msg$ that govern the interactions between senders and receivers.

Proof: To establish this theorem, we perform a case analysis. Rules $send\_app\_msg$ and $receive\_app\_msg$ govern the interactions between senders and receivers. $send\_app\_msg$ sends the application message to a receiver. In Figure 4 line 2, the rule creates a new interaction key using the $newIdentifier(a_s, a_r)$ pseudo-function, which guarantees the creation of a unique interaction key (under the assumption that actor identities are unique). Furthermore, in line 3, the rule assigns the interaction key to the application message and sends the application message to the receiver. Therefore, the ASM, through the $send\_app\_msg$, rule complies to the Unique Interaction Key Rule and the Interaction Key Transmission Rule.

In $send\_app\_msg$ rule, the $makeAssertions(a_s, \kappa, d, r)$ function uses the generated interaction key to create p-assertions about the interaction (see Figure 4 line 3). Likewise, in the $receive\_app\_msg$ rule, the $makeAssertions$ $(a_r, \kappa, d, \perp)$ function uses the interaction key in the application message, $\mathsf{app}(\kappa, d)$, to create p-assertions about the interaction (see Figure 4 line 9). Therefore, the ASM, through these rules, complies with the Appropriate Interaction Rule.

Because the ASM complies to the Unique Interaction Key Rule, the Interaction Key Transmission Rule, and the Appropriate Interaction Rule, actors following PReP are actor behaviour compliant as defined by Definition 2. ■

C. Proof for the Complete View Invariant

Invariant 3 (Complete View): Consider a configuration, c_1, where a View is complete. For any configuration c_2, such that $c_1 \longmapsto^* c_2$, then

$$View(c_1, a_{ps}, \kappa, v) = View(c_2, a_{ps}, \kappa, v),$$

for all a_{ps}, κ, v.

Proof: We prove this lemma by induction on the length of transition sequence $c_1 \longmapsto^* c_2$. In the base case, where the length is 0, c_2 is the same as c_1 and the lemma trivially holds.

In the inductive case, a length greater than zero is considered. We assume that the invariant holds for $c_1 \longmapsto^* c_n$ and show that the property holds for all possible transitions from $c_n \longmapsto c_2$.

An inspection of the ASM rules shows that only two rules modify views. They are $receive\_submission\_finished$ and $receive\_record\_passertion$. By hypothesis, the view is complete. Hence, by definition of complete, *vs* is not null. From Invariant 1, if *vs* is not $\perp$, *vs* cannot be modified. Likewise, the rules prevent messages from being recorded if *vs* is not $\perp$ (see Figure 3 line 4 and the complete function definition). Therefore, $View(c_2, a_{ps}, \kappa, v) = View(c_n, a_{ps}, \kappa, v)$ and the invariant holds. ■

D. Guaranteed Recording

Once p-assertions have been created, we show that they will be recorded in the provenance store and acknowledged. Before establishing this formally, we first define two invariants. The first invariant shows that every message sent to the provenance store will be recorded by it. The second invariant shows that once a message has been recorded the recorder will have a corresponding acknowledgement message. Both invariants can be proved by induction on the length of the transition sequence between two configurations. We make the following assumption:

- There is implicit conversion from sets to bags.

We also define a function to select a group of messages from a table that contain a particular interaction key and view kind combination.

Definition 7 (Messages Selector): For all κ, v,

$$Bag(\mathcal{M}) \downarrow (\kappa, v) = \Big\{ \mathsf{rec}(\kappa, v, -, -, -) \in Bag(\mathcal{M}), \mathsf{vs}(\kappa, v, -, -, -) \in Bag(\mathcal{M}) \Big\}$$

Invariant 4 (Always Recorded): For any configuration c_1 reachable from c_i, for all κ, a, a_{ps},

$$sent\_T(a, \kappa) \downarrow (\kappa, v) =$$

$$k(a, a_{ps}) \downarrow (\kappa, v)$$
$$\oplus store\_T(a_{ps})(\kappa, v).recs$$
$$\oplus store\_T(a_{ps})(\kappa, v).vs$$

where v is the role identifier of a.

Proof: We prove this invariant by induction on the length of the transition sequence, $c_i \longmapsto^* c_1$, where c_i is the initial configuration of the state machine.

In the base case, where the length is zero, c_1 equals c_i. In c_i all tables and channels are empty and thus the invariant holds.

In the inductive case, we assume that for $c_i \longmapsto^* c_n$ the invariant holds. We then consider the possible transitions from $c_n \longmapsto c_1$. There are three rules that deal with the tables identified in the invariant. We address each individually.

- In the $record\_message$ rule, a message is added to the $sent\_T$ table and is also placed on the channel between a and a_{ps} through the $send$ pseudo-statement. Hence, $sent\_T(a, \kappa)$ in c_1 is $sent\_T(a, \kappa) \oplus m$ in c_n and $k(a, a_{ps})$ in c_1 is $k(a, a_{ps}) \oplus m$ in c_n. Thus, the invariant holds for c_1 since it holds in c_n.
- In the $receive\_record\_passertion$ rule, a message on the channel from a to a_{ps} is received and is therefore removed from the channel. The same message is then added to $store\_T$ in Figure 3 line 6. Hence, $k(a, a_{ps})$ in c_1 is $k(a, a_{ps}) \ominus m$ in c_n and $store\_T(a_{ps})(\kappa, v).recs$ in c_1 is $store\_T(a_{ps})(\kappa, v).recs \cup \{m\}$ in c_n. Therefore, the invariant holds in c_1 because it holds in c_n.
- In the $receive\_view\_size$ rule, a message on the channel from a to a_{ps} is received and is therefore removed from the channel. The same message is then added to $store\_T$ in Figure 3 line 14. Hence, $k(a, a_{ps})$ in c_1 is $k(a, a_{ps}) \ominus m$ in c_n and $store\_T(a_{ps})(\kappa, v).vs$ in c_1 is $store\_T(a_{ps})(\kappa, v).vs \cup \{m\}$ in c_n. Therefore, the invariant holds in c_1 because it holds in c_n.

Therefore, the property holds in the inductive case and the invariant is established. ■

We note that the actor sending a message to a provenance store will receive a corresponding acknowledgement message. This notion relies on the following function that converts the messages in a View to a set of acknowledgement messages.

Definition 8 (Message to Ack Conversion): For all a, κ, v, $viewToAck(store\_T(a)(\kappa, v))$ is defined as

$$Bag\Big(\mathsf{ack}(\kappa, v, \ell, b) \mid \mathsf{rec}(\kappa, v, a, \ell, \alpha) \in store\_T(a)(\kappa, v).recs \text{ or } \mathsf{vs}(\kappa, v, a, \ell, na) \in store\_T(a)(\kappa, v).vs\Big)$$

By using this function, the messages stored in the provenance store can be converted to acknowledgement messages and thus can be matched to the acknowledgement messages received by the recording actor. This matching is explained by the following invariant.

Invariant 5 (Always Acknowledged): For all configurations c_1 reachable from c_i, for all κ, a, a_{ps}, v,

$$viewToAck(store\_T(a)(\kappa, v)) =$$
$$k(a_{ps}, a) \downarrow (\kappa, v) \oplus ack\_T(a, \kappa) \downarrow (\kappa, v).$$

Proof: We prove this invariant by induction on the length of the transition sequence $c_i \longmapsto^* c_i$, where c_i is the initial configuration of the state machine.

In the base case, where the length is zero, all tables and channels are empty and thus

$$viewToAck(store\_T(a)(\kappa, v)) = k(a_{ps}, a)$$
$$= ack\_T(a, \kappa) = \emptyset$$

and the invariant holds.

In the inductive case, we assume that for $c_i \longmapsto^* c_n$ the invariant holds. We then consider the possible transitions from $c_n \longmapsto c_1$. There are three rules that deal with the tables identified in the invariant. We address each individually.

- In the $receive\_record\_passertion$ rule, a message is added to the $store\_T$ table. A corresponding acknowledgement message is also generated and added to the the channel $k(a_{ps}, a)$. Hence,

  $$viewToAck(store\_T(a)(\kappa, v)) \text{ in } c_1 \text{ is}$$
  $$\text{then } viewToAck(store\_T(a)(\kappa, v)) \oplus m \text{ in } c_n,$$

  and $k(a_{ps}, a)$ in c_1 is $k(a_{ps}, a) \oplus m$ in c_n. Thus, the invariant holds in c_1 since it holds in c_n.
- In the $receive\_view\_size$ rule, a message is added to the $store\_T$ table. A corresponding acknowledgement message is also generated and added to the the channel $k(a_{ps}, a)$. Hence,

  $$viewToAck(store\_T(a)(\kappa, v)) \text{ in } c_1 \text{ is}$$
  $$\text{then } viewToAck(store\_T(a)(\kappa, v)) \oplus \quad m \text{ in } c_n$$

  and $k(a_{ps}, a)$ in c_1 is $k(a_{ps}, a) \oplus m$ in c_n. Thus, the invariant holds in c_1 since it holds in c_n.
- In the $receive\_ack$ rule, an acknowledgement message is removed from the channel $k(a_{ps}, a)$ and then added to the $ack\_T(a, \kappa)$ table. Hence, $k(a_{ps}, a)$ in c_1 is $k(a_{ps}, a) \ominus m$ in c_n and $ack\_T(a, \kappa)$ in c_2 is $ack\_T(a, \kappa) \oplus m$ in c_n. Therefore, the invariant holds in c_1 because it holds in c_n.

Therefore, $viewToAck(store\_T(a)(\kappa, v)) = k(a_{ps}, a) \downarrow (\kappa, v) \oplus ack\_T(a, \kappa) \downarrow (\kappa, v)$ holds in the inductive case and the invariant is established. ■

Using Invariants 4 and 5, we now establish that the messages and thus the p-assertions within those messages sent to the provenance store will be stored and an acknowledgement will be received by the sender. Thus, the acknowledgements received from the provenance store by the sender will be equal to what was originally sent by the sender (after a simple conversion step).

Lemma 4 (Always Recorded and Acknowledged): For all a, a_{ps}, in any reachable configuration, c_f where all channels between a and a_{ps} are empty, all messages from the set $R \cup VS$ that have been sent by a to a_{ps} have been stored in

a_{ps} and a has received an acknowledgement.

Proof: From Invariant 4, if $k(a, a_{ps}) = \emptyset$ then for all κ, v,

$$sent\_T(a, \kappa) \downarrow (\kappa, v) =$$

$$store\_T(a_{ps})(\kappa, v).recs \cup store\_T(a_{ps})(\kappa, v).vs.$$

By definition of $store\_T(a)(\kappa, v)$, we can collapse this equation to be

$$sent\_T(a, \kappa) \downarrow (\kappa, v) = store\_T(a_{ps})(\kappa, v).$$

Furthermore, from Invariant 5, if $k(a_{ps}, a) = \emptyset$ then for all κ, v,

$$viewToAck(store\_T(a)(\kappa, v)) =$$

$$ack\_T(a, \kappa) \downarrow (\kappa, v).$$

Hence,

$$viewToAck(sent\_T(a, \kappa) \downarrow (\kappa, v)) =$$

$$ack\_T(a, \kappa) \downarrow (\kappa, v).$$

Essentially, what an actor sent to a provenance store has been acknowledged. Given that no rule removes messages from the acknowledgement table and from Invariant 1 the provenance store is immutable, all record and view size messages that have been sent to a provenance store are stored in the provenance store and acknowledgements have been received by the recorder. ■

We now introduce two more invariants that state that after p-assertions are created, they end up in the $sent\_T$. The proofs are omitted for brevity.

Invariant 6 (Always Sent): For all configurations c_1, c_2, where $c_1 \longmapsto^* c_2$, for all a, κ, v,

$$assert\_T(a, \kappa, v) \oplus to\_send\_T(\kappa, v) \oplus sent\_T(\kappa, v)$$

is constant for all transitions excluding $send\_app\_msg$ and $receive\_app\_msg$.

Invariant 7 (P-assertion Accumulation): For all configurations c_1, c_2, where $c_1 \longmapsto^* c_2$, for all a, κ, v,

$$assert\_T(a, \kappa, v) \oplus to\_send\_T(\kappa, v)$$

$$\oplus sent\_T(\kappa, v) \quad at c_1$$

$$\subseteq assert\_T(a, \kappa, v) \oplus to\_send\_T(\kappa, v)$$

$$\oplus sent\_T(\kappa, v) \quad at \quad c_2$$

for $send\_app\_msg$ and $receive\_app\_msg$ transitions.

The above lemma and invariants show that once p-assertions are created, they will end up in a provenance store and the creator of the p-assertions will have received acknowledgements that they have been stored.

E. Proof of Process Reflection

Lemma 2 (Process Reflection): For any application state, as, reachable from as_i, where $as_i \longmapsto^*_{app} as$; for all pas reachable from pas_i: $pas_i = \langle c_i, as_i \rangle \longmapsto^*_{paa} pas$ with $pas = \langle c, as \rangle$; there exists a final configuration $pas_2 = \langle c_2, as \rangle$ for some c_2, where there are no messages in transit and no messages to send, such that $pas \longmapsto^*_{paa} pas_2$ without application transitions, such that the provenance stores in pas_2 contain the description of $as_i \longmapsto^*_{app} as$.

Intuitively, the application proceeds from an initial state as_i to some final state as where the application has finished executing. Because of system coupling (shown by the vertical hash lines), the provenance-aware application also proceeds from an initial state $\langle c_i, as_i \rangle$ to some state $\langle c, as \rangle$. However, there may be p-assertions remaining to be recorded that describe the application's execution, thus the provenance-aware application finishes recording those p-assertions without using application transitions (denoted by $paa\text{-}wa$ in the figure).

Proof: Our proof outline proceeds by induction on the length of the transition sequence from $as_i \longmapsto^*_{app} as$.

In the base case, as_i equals as, hence no execution has taken place and process documentation is empty and the lemma holds trivially.

In the inductive case, we assume if $as_i \longmapsto^*_{app} as_n$ then $pas_i \longmapsto^*_{paa} pas_n$ and process documentation describing $as_i \longmapsto^*_{app} as_n$ will be recorded in a set of provenance stores at some later configuration, $\langle c_x, as_n \rangle$. This is the inductive hypothesis and is shown in Figure 6.

We now show that for all possible transitions from $as_n \longmapsto_{app} as$ that process documentation describing that transition will be in a provenance store after $pas_n \longmapsto^*_{paa-wa} pas_2$, where pas_2 = $\langle c_2, as \rangle$ and one application transition.

In this inductive step, the application proceeds from as_i after any number of transitions to the state as_n. We assume that p-assertions describing this execution are recorded in the provenance store when the provenance-aware application state space reaches $\langle c_x, as_n \rangle$. Once the application has reached the state as_n, one more application transition occurs to the state as. Through system coupling, the provenance-aware application state will also proceed from $\langle c_x, as_n \rangle$ to $\langle c_x, as \rangle$ by one application transition. We note that this application transition can occur at any time after configuration $\langle c_n, as_n \rangle$. It does not have to wait for the recording of p-assertions to finish.

We now consider the two possible application transitions from $as_n \longmapsto_{app} as$, $produce\_msg$ and $consume\_msg$. When one of these rules executes, from Definition 5, the corresponding rule ($send\_app\_msg$ and $receive\_app\_msg$) defined in Figure 4 will eventually execute. This eventual execution is based on our hypothesis that in the ASM any number of other rules can fire between the execution of the application rule and the corresponding PReP rule but that rule will fire.

Once the corresponding rule is fired, the $makeAssertion$ function is called and a set of p-assertions, $\alpha_1, \ldots, \alpha_n$, is produced. These p-assertions are then placed in a table, $assert\_T$, with a key for the particular interaction, κ. Once α is in the $assert\_T$ table, from Invariants 6 and 7, it will end up in a message in $sent\_T$. From Lemma 4, the message will be recorded and acknowledged in a finite number transitions and the system will reach a final configuration, pas_2. Thus, in the inductive case the lemma holds. ■