Recording Process Documentation in the Presence of Failures

Zheng Chen and Luc Moreau

School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
zc05r@ecs.soton.ac.uk, L.Moreau@ecs.soton.ac.uk

Abstract. Scientific and business communities present unprecedented requirements on *provenance*, where the provenance of some data item is the process that led to that data item. Previous work has conceived a computer-based representation of past executions for determining provenance, termed *process documentation*, and has developed a protocol, PReP, to record process documentation in service oriented architectures. However, PReP assumes a failure free environment. Failures lead to process documentation unable to be recorded, losing the evidence that a process occurred. This is not acceptable in the applications relying on process documentation and would cause disastrous consequences. This paper describes our solution, F_PReP, a protocol for recording process documentation in the presence of failures. A complete formalisation of the protocol using Abstract State Machines is also presented.

1 Introduction

In scientific and business communities, a wide variety of applications have presented unprecedented requirements [20] for knowing the provenance of their data products, e.g., where they originated from and what has happened to them since creation. In chemistry experiments, provenance is used to detail the procedure by which a material is generated, allowing the material to be patented. In healthcare applications, in order to audit if proper decisions were made for a patient, there is a need to trace back the origins of these decisions. In engineering manufacturing, keeping track of the history of generated data in simulations is important for users to analyse the derivation of their data products. In finance business, the provenance of some data item establishes the origin and authenticity of the data item produced by financial transactions, enabling reviewers and auditors to verify if these transactions are compliant with specific financial regulations.

To meet these requirements, Groth et al. [15] have proposed an open architecture to record and access a computer-based representation of past executions, termed *process documentation*, which can be used for determining the provenance of data. A generic recording protocol, PReP [16], has been developed to provide interoperable means for recording process documentation in the context of service oriented architectures. In this architecture, process documentation consists

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 196–219, 2009.

of a set of assertions (termed *p-assertions*) made by *asserting actors* (i.e., either clients or services) involved in a process (i.e., the execution of a workflow). A dedicated repository, termed *provenance store*, is used to maintain p-assertions. For scalability reason, multiple provenance stores may be employed and process documentation may end up distributed, linked by pointers recorded along with p-assertions in each store. Using the pointer chain, distributed process documentation can be retrieved from one store to another.

Recording process documentation in the presence of failures is an issue that has been lacking attention so far. PReP assumes a system in which no failure occurs. However, large scale, open distributed systems are not failure-free [8,9]. For example, a service may not be available and network connection may be broken. The presence of failures may prevent process documentation from being recorded, losing the evidence that a process occurred. We now draw a parallel between the documentation of a process and a particular type of evidence in a legal setting, testimony. The absence of testimony from eyewitnesses to a crime scene would make it difficult for juries to make a judgement about whether to believe the set of claims provided by a suspect. Similarly, the unavailability of process documentation is not acceptable in the above domains that rely on process documentation to determine the provenance of their data products. It may also cause disastrous consequences as in the example of a provenance-based service billing system. In this system, users are charged according to their usage of services described by process documentation. If a user invoked a service, but documentation fails to describe this invocation, then the user will be charged too little, which must be avoided.

To address this problem, we have designed a recording protocol, F_PReP, which provides remedial actions and a novel component, Update Coordinator, to guarantee the recording of process documentation in the case of failures. The protocol has been formalised as an abstract state machine and its correctness has been proved. This paper details the protocol and presents its formalisation.

The rest of the paper is organised as follows: Section 2 introduces some terminology and identifies a set of requirements that the protocol should meet. Section 3 states our failure assumptions and defines protocol messages. In Section 4, we present a formalisation of the protocol and detail the protocol's behaviour. Then we outline the proof of the protocol's correctness in Section 5. Finally, Section 6 discusses related work, followed by a conclusion in Section 7.

2 Terminology and Requirements

2.1 Terminology

Process documentation describes a past process that led to a result. Such a process is modelled as a causally connected set of interactions between actors involved in that process [14]. An *interaction* is concerned with one application message exchanged between two actors, i.e., its sender and its receiver. An actor documents an interaction by making p-assertions to provide a sender or receiver's

view of the interaction. Process documentation therefore consists of a set of p-assertions.

A p-assertion can document the application message exchanged in an interaction (*interaction p-assertion*) or the internal state of an actor (*actor state p-assertion*), such as time and memory usage, in the context of an interaction. It can also be a *relationship p-assertion*, capturing the internal causal connections between interactions within the scope of an actor, i.e., the interaction where an output message is sent (*effect interaction*) and the interaction where an input message is received (*cause interaction*).

PReP specifies that *both* actors in an interaction must make p-assertions documenting the interaction for accountability or verification purposes. For scalability reason, an actor can use various stores to record p-assertions about different interactions, though p-assertions about the same interaction must be recorded in the same place. Besides, the p-assertions made by the two actors in an interaction are also allowed to be recorded in two different stores. A notion of link, i.e., a pointer to a provenance store, has been introduced to connect distributed documentation [14].

There are two types of links, *viewlink* and *causelink*. If the two actors in an interaction use two different stores, each actor records a *viewlink* that points to the provenance store where the opposite party recorded their p-assertions about that interaction. Therefore, both views of an interaction can be retrieved by navigating from one provenance store to the other. The causelink is used in relationship p-assertions. If the p-assertions that represent a cause interaction are recorded in a different provenance store, a *causelink* is embedded in the relationship p-assertion, indicating which provenance store the p-assertions representing the cause interaction are stored in. To facilitate the description of our protocol, we define a term *ownlink* as a pointer to the provenance store where an actor records its *own* p-assertions.

Figure 1 shows an example of how links are recorded. Actor A sends an application message M2 to actor B as a consequence of message M1. A uses provenance stores P_R and P_A to record p-assertions about the interactions in which M1 and M2 are exchanged, respectively. B records p-assertions about the receipt of M2 in provenance store P_B. In order to exchange a viewlink to B, A includes its ownlink to P_A in M2. B then extracts the link and records it as its viewlink in P_B. As a result, a viewlink from P_B to P_A is created (shown by the arc VL 1). We assume that A knows from its configuration that B always stores its p-assertions in P_B. Hence, A records a viewlink to P_B in P_A. Finally, A makes a relationship p-assertion between its effect interaction containing M2 and the previous cause interaction containing M1. In the relationship p-assertion, it adds a causelink to P_R, where the p-assertions related to the cause interaction are stored. A then records the relationship p-assertion in P_A, thus connecting P_A to P_R shown by the arc CL.

By recording links, a pointer chain can be formed connecting all the provenance stores hosting the documentation of a process. Using the pointer chain, distributed documentation can be retrieved from one store to another.

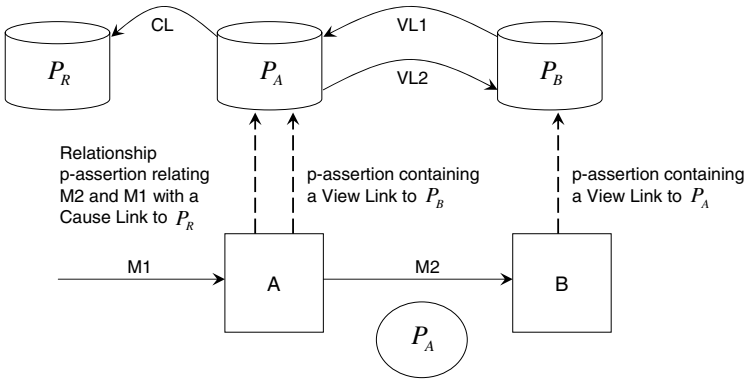

Fig. 1. A example of link [14]

2.2 Requirements

Miles et al. [20] have presented requirements that a provenance system should support, such as verifiability, accountability, reproducibility, preservation, scalability, generality, customisability, non-repudiation and distribution. They have been of particular importance in motivating the design of our protocol. We now identify several new requirements that are related to failures.

PReP does not specify well-defined behaviour when recording documentation in the presence of failures. For example, it assumes an actor always obtains an acknowledgement from a provenance store for receiving a p-assertion and hence does not consider the situation where the acknowledgement is lost or provenance store crashes before storing that p-assertion. This may result in incomplete process documentation, which requires us to design a robust protocol to meet the following requirement:

Requirement 1 (Guaranteed Recording). *After a process finishes execution, the entire documentation of that process must eventually be recorded in provenance store(s).*

Distributed process documentation is connected by a chain of pointers (links) to enable retrievability. Accurate pointers must exist even in the presence of failures, leading to two requirements.

Requirement 2 (Viewlink Accuracy). *Viewlinks recorded for each interaction of a process must eventually be accurate in provenance stores. Each must point to the store where the other actor in the same interaction recorded p-assertions documenting that interaction.*

Requirement 3 (Causelink Accuracy). *Causelinks recorded during a process must eventually be accurate in provenance stores. Each must point to the store where p-assertions about the corresponding cause interaction were recorded.*

Creation and recording p-assertions have already introduced overhead into the application [13]. The remedial actions specified by the protocol may however take up computing resources and interfere with applications. In terms of recording performance, we identify another two requirements:

Requirement 4 (Efficient Recording). *Recording p-assertions and taking remedial actions should be efficient and introduce minimum overhead.*

Requirement 5 (Transparent Recording). *Recording p-assertions and taking remedial actions should be transparent to the application.*

Among the above requirements, requirements *Guaranteed Recording*, *Viewlink Accuracy* and *Causelink Accuracy* are concerned with the protocol's correctness, which are to be proved in Section 5.

3 Protocol Description

We firstly outline the design philosophy of F_PReP and state several assumptions, under which F_PReP meets the requirements identified in Section 2.2. Then, we define the protocol's messages and describe the protocol.

3.1 Design

The goal of our work is to design a *general* protocol, i.e., application and implementation independent, for recording process documentation in large, open distributed environments where a large number of provenance stores are present and failures may occur. Since PReP has provided an application independent solution to recording process documentation, we decided to derive PReP in order to inherit its generic nature.

There are several challenges in designing a distributed protocol that can cope with failures. Firstly, we need to state an appropriate failure model and systematically identify system behaviour in the case of failures. Failures are nondeterministic in nature and typically very hard to predict. Restricting our scope to particularly failures is hence necessary. Secondly, the protocol may involve the co-operation of several parties such as asserting actors, provenance stores, and if necessary, additional components. Designing such a distributed protocol is notoriously difficult, since we have to stay in control of not only the normal system behaviour when there is no failure but also of the complex situations which can occur when failures happen.

We restrict ourselves to certain failures that may occur during the recording of p-assertions into a provenance store.

Assumption 1. *Provenance stores may crash, i.e., they halt and stop any further execution, and can be restarted from their latest consistent state*[1].

[1] The provenance store has been implemented as a stateless web service with a database storage system. Hence the latest consistent state refers to the initial state of the service and the latest checkpointed state of the database.

Assumption 2. *Messages to/from provenance stores can be lost, reordered but not duplicated in communication channels.*

We do not consider the failures of asserting actors and the exchange of application messages since they are application dependant. Applications should provide fault tolerance mechanisms to ensure asserting actors' availability and reliable exchange of application messages.

Assumption 3. *An asserting actor has several provenance stores to use.*

Given that we are considering an open system where there are a large number of provenance stores, it is reasonable to make use of alternative stores to provide fault tolerance.

We now analyse several failure types in a recording scenario where an asserting actor sends a p-assertion (pa) to a provenance store (PS) and PS replies the actor with an acknowledgement (ack) after recording pa in its persistent storage.

- The message pa is lost;
- PS crashes before receiving pa;
- PS crashes after receiving pa and before recording pa;
- PS crashes after recording pa and before replying ack;
- The message ack is lost.

From an asserting actor's perspective, all these failure types may lead to the incapability of receiving an ack from a provenance store. An asserting actor can set a timeout when waiting for an ack message. If the actor does not receive a response within that time, it knows failures *may* have occurred; it can then send the p-assertion again. We note that a low speed network or a provenance store experiencing slowdown can also result in an expired timeout. Since a p-assertion may be recorded in a provenance store even in the case of timeout, a provenance store should be designed to handle duplicate p-assertions due to retransmission, and always return the same acknowledgement for a specific p-assertion.

We identify several remedial actions that the protocol needs to take in the presence of failures. The primary one is to resend a p-assertion to a provenance store due to a timeout. After several reattempts, if the p-assertion still fails to be acknowledged, an actor may use alternative stores to resubmit the p-assertion until it is acknowledged. A successful receipt of an acknowledgement tells the actor that the p-assertion being acknowledged has been recorded in a provenance store.

Since distributed process documentation is connected using links to enable retrievability, the use of alternative provenance stores causes a link to the original store incorrect. Hence, an asserting actor needs to take other remedial actions. To satisfy *Causelink Accuracy*, it can maintain history information of using alternative stores during its participation in a process. The protocol checks an actor's causelinks when recording relationship p-assertions and updates them according to the history information. To achieve *Viewlink Accuracy*, we introduce a novel component, Update Coordinator, to facilitate viewlink updating. An update coordinator is only involved when an alternative store is used, which means it does not participate in every interaction, hence introducing small overhead.

Assumption 4. *The update coordinator does not fail.*

We can use the traditional fault-tolerance mechanisms such as replication to ensure its availability. This is feasible since we can have only one coordinator per process and a coordinator maintains only a small amount of information, as illustrated later. However, it is infeasible to use the replication mechanism for provenance stores for two reasons. Firstly, we have assumed an open system where there are a great number of provenance stores, it is hard to assume each is facilitated with replicated backups. Secondly, though replication is sophisticated, it comes with a significant cost due to preserving the *one-copy equivalence* property [23]. Given that the documentation produced in a process can be on the order of terabytes [11], replication becomes very expensive and time consuming. Therefore, compared with replicating provenance stores, the use of alternative stores is a more general, simple and flexible approach.

To meet requirements *Efficient Recording* and *Transparent Recording*, F-PReP is designed to be an asynchronous protocol, allowing actors to send p-assertions at any time. This means that actors can choose when to record p-assertions without delaying their execution. Secondly, all p-assertions about one interaction are submitted in a single batch and hence can be acknowledged using one acknowledgement message, saving on the overhead of establishing network connections. Thirdly, remedial actions, e.g., selecting alternative stores, are taken by the protocol irrespective of the application.

3.2 Messages

F_PReP is a distributed protocol, specifying the behaviour of actors (i.e., asserting actors, provenance stores and update coordinator) and their communications. It is defined based on *interaction* i.e., the exchange of an application message between a sender and receiver.

There are six messages in the protocol: Application Message (app), Interaction Record Message (record), Record Ack Message (ack), Repair Message (repair), Update Message (update), and Update Ack Message (uack). We now define each message with Figure 2, which provides an example of actors exchanging these messages.

Application Message. The application message app is exchanged by all application actors. It contains application specific data needing to be transferred between actors. In the context of a provenance system, the application message is adapted to include interaction contextual information: an interaction key and the sender's ownlink.

An interaction key is generated by the sender in an interaction for uniquely identifying the interaction from all other interactions. The receiver can then use the same interaction key to record p-assertions about the same interaction.

In Figure 2, we assume that the key for the interaction where the sender, a, sends an application message to the receiver, b, is i. We also assume the default provenance stores that a and b use are $PS1$ and $PS2$, respectively. In Step 1, a

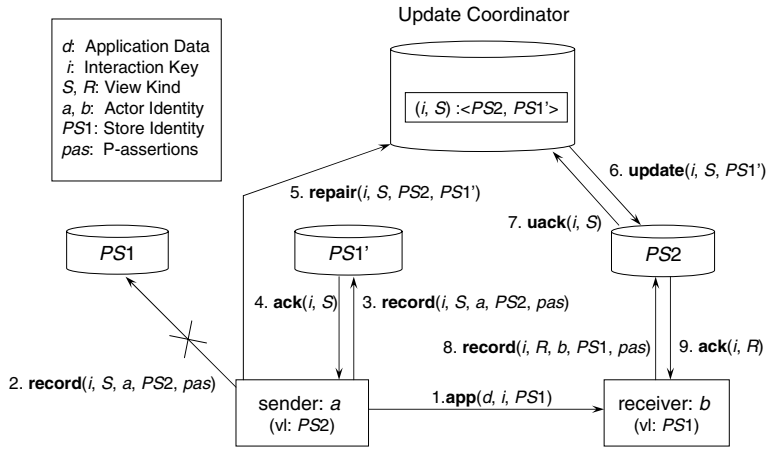

Fig. 2. Protocol Message Exchanges

sends an app to b containing application data d, interaction key i and a's ownlink to $PS1$ (Step 1). Upon receiving app, b becomes aware of its viewlink to $PS1$. We assume that a's viewlink to $PS2$ has been made available to a by means not explained in the figure; the viewlink can be built into a at deployment time or transferred to a in a response message or in an extra message from b.

Interaction Record Message. For each interaction, both actors document the interaction by asserting p-assertions and sending them in an interaction record message, record, to their respective provenance stores. The message contains: (1) an interaction key, identifying the interaction being documented; (2) a view kind, indicating the role of the asserting actor in the interaction, i.e., a sender or a receiver; (3) an actor identity, representing the asserting actor that documents the interaction, which is essential for recording attributable process documentation; (4) a viewlink of the asserting actor for that interaction; (5) a set of p-assertions that describe the interaction.

In Figure 2, both a and b create a set of p-assertions, pas, about the interaction, i, and send them in record messages with their viewlink to PS2 and PS1, respectively (Steps 3, 8). We note that the two record messages can be sent in any order, not restricted by the step numbers in the figure.

The set of p-assertions must contain an interaction p-assertion to document the exchange of an app message. If app is the consequence of receiving other messages, then the sender of app must make a relationship p-assertion to capture the causal connections between these messages.

Due to the asynchronous nature of the protocol, an asserting actor accumulates record messages in a local queue and submits them to a provenance store at its most convenient time. Before delivering a record message to a provenance store, an actor checks all the relationship p-assertions in the message and updates incorrect causelinks in order to meet *Causelink Accuracy* requirement. These actions are detailed in Section 4.3.

Record Ack Message. A provenance store acknowledges record message by means of an acknowledgement message ack, only *after* it has successfully recorded the content of record in its persistent storage. An ack message includes an interaction key and a view kind, indicating from which view of an interaction, a record is being acknowledged. Therefore, one ack can acknowledge a set of p-assertions in a record, which reduces communication overhead.

An asserting actor sets a timeout when waiting for an ack immediately after it sends a record to a provenance store. This helps the actor take remedial actions without waiting too long. If an ack is not received before the timeout, then the actor resends the same record to the actor's default store or an alternative store. Only after receiving an ack acknowledging a record can the actor eliminate the record from its local queue. An ack means that the acknowledged record message has been processed and recorded in a provenance store persistently.

In Figure 2, a sends a record to its default store $PS1$ (Step 2) but does not receive an ack before a timeout. Then it selects another store $PS1'$ to use (Step 3) and finally receives an ack (Step 4).

Repair Message. An asserting actor sends a repair message, repair, to an update coordinator to request an update of the other actor's viewlink. It consists of four elements: (1) an interaction key, indicating in which interaction the opposite actor's viewlink is to be updated; (2) the asserting actor's view kind in the interaction; (3) a pointer ($DestPS$) to the provenance store recording the opposite actor's viewlink in the interaction; (4) the actor's ownlink, pointing to the provenance store from which the actor received an ack for that interaction.

An actor issues a repair request only if it used an alternative store in an interaction, which results in the other actor's viewlink incorrect. In Figure 2, the sender sends its record to the alternative store $PS1'$ (Step 3) and receives an ack (Step 4). As a consequence, the receiver's viewlink to $PS1$ becomes incorrect, hence requiring an update. In order not to interfere with applications to support *Transparent Recording* requirement, the protocol does not allow the sender to directly inform the receiver with its new ownlink, which is now pointing to $PS1'$. Instead, the sender requests an update coordinator (Step 5) to help update the receiver's provenance store (Step 6).

An update coordinator is necessary since both sender and receiver may issue a repair request in an interaction. This cannot be achieved by direct update of the other actor's provenance store, because at that moment, one does not know which store the opposite actor is actually using. In Figure 2, if the receiver uses an alternative store to record its p-assertions, then the sender's viewlink to $PS2$ becomes incorrect as well. In that case, the receiver needs to issue another repair request to the coordinator.

Since an update coordinator is not involved in every interaction, we recommend that all the application actors participating in a process employ one coordinator. If using more than one, then any two actors exchanging an application message must share the same one in order to ensure Viewlink Accuracy requirement. The identifier of a coordinator can be built in actors or exchanged to other actors in the application message app. Figure 2 employs the former approach.

We do not consider the loss of repair messages in channel, which can be solved by using an extra acknowledgement message and retransmission actions. Assumption 4 implies that a repair request can always be processed by an update coordinator.

Update Message. The update coordinator sends an update message, update, to a provenance store in order to update a viewlink in that store. The message contains: (1) an interaction key, indicating for which interaction, the opposite actor's viewlink needs to be updated; (2) the view kind of the asserting actor that issued a repair request for that interaction; (3) the ownlink of the requesting actor. The $DestPS$ field in the repair message tells the update coordinator where to send the update message.

In order to deal with the case where both actors in an interaction each issue a repair request, which can be in any order, the update coordinator maintains request information: the identity of the destination store, specified by the $DestPS$ field in the repair message, and the requesting actor's ownlink. This request information is indexed by the pair of interaction key and view kind. In Figure 2, after receiving a repair request from the sender, the coordinator records a tuple $(PS2, PS1')$ indexed by the pair (i, S). Then the coordinator sends to store $PS2$ an update message containing the sender's ownlink to $PS1'$ (Step 6). Therefore, the receiver's viewlink stored in $PS2$ is replaced with $PS1'$ and hence becomes correct.

If the update coordinator receives two repair messages each from one asserting actor in an interaction, then it sends out two update messages after performing operations using the stored request information to ensure that both update messages are delivered to correct destination stores. We detail the coordinator's internal behaviour in Section 4.5.

We note that a provenance store may receive an update and a record message in any order (Steps 6, 8). The protocol specifies that the viewlink obtained from update is NOT overwritten by the one from record in order to achieve *Viewlink Accuracy* requirement.

Update Ack Message. After updating a viewlink in a provenance store, the store returns an acknowledgement message uack, containing an interaction key and a view kind, to the update coordinator acknowledging the respective update message. Since update or uack messages may be lost in channel according to Assumption 2, the coordinator sets a timeout when waiting for a uack and an expired timeout leads to resending the same update message.

4 Protocol Formalisation

F_PReP has been formalised through the use of an abstract state machine (ASM). The ASM notation we adopt has been used previously to describe a distributed reference counting algorithm [22] and a fault-tolerant directory service for mobile agents [21]. 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. Such a formalisation provides a precise, implementation-independent means of describing the system.

We begin by describing the state space of the ASM, and then proceed to discuss its transitions. Finally, we detail the behaviour of each kind of actors.

4.1 System State Space

Figure 3 shows the system state space. We identify specific subsets of actors in the system, namely, the senders, the receivers, provenance stores, and update coordinators. The set of each of protocol messages is defined formally as an inductive type. For example, the set of Application Messages is defined by an inductive type whose constructor is app and whose parameters are from the set of DATA, IK and OL. The notation DATA refers to the set of application related data. The set of all protocol messages ($\mathcal{M}$) is defined as the union of these message sets. Messages are exchanged over a set of communication channels, $\mathcal{K}$. Since no assumption is made about message order in communication channels, $\mathcal{K}$ is represented as bags of messages between pairs of actors. The power set notation ($\mathbb{P}$) denotes that there can be more than one of a given element.

We define the set of relationship p-assertions as an inductive type whose constructor is rel-pa. The names of relationships are given in the set REL. Since a relationship p-assertion captures causal connections between effect interaction and cause interaction(s), we use the set EID and CID to index the respective interactions, each containing the interaction's key (IK) and the role (VK) that the actor plays in that interaction. With EID or CID, the p-assertions about a related interaction can be found in a local provenance store or a remote store (indicated by a causelink from the set CL). The set of interaction p-assertions can be constructed by i-pa whose parameter is from the set IK and application data set DATA. Since actor state p-assertions are not used in the formalisation, we do not model them to simplify the state space. The set of all kinds of p-assertions (PA) is defined as the union of these p-assertion sets.

The internal functionality of each kind of actors is modelled as follows.

Sender and Receiver State Space. An asserting actor (indexed by an asserter identity) uses various tables ($in\_T \in$ IN, $asserter\_T \in$ ASSERTER, $log\_T \in$ LOG, $queue\_T \in$ QUEUE, $lc \in$ LC and $timer\_T \in$ TIMER) to record p-assertions into a provenance store. A table maps a key to a tuple. For example, the table ($asserter\_T$) maps an interaction key ($\kappa \in$ IK) and the actor's view kind ($v \in$ VK) to a tuple of four elements: the state of an interaction record message during recording ($str \in$ STR), the actor's ownlink ($ol \in$ OL), viewlink ($vl \in$ VL) and the p-assertions created in the interaction ($\mathbb{P}$(PA)).

As all the data that an asserting actor works upon is located in received messages, these incoming messages and interaction keys identifying these messages are stored in a table (in_T), which is used when creating p-assertions.

$$
\begin{array}{rll}
\mathrm{A} & = & \{a_1, \ldots, a_n\} & \text{(Set of Actor Identities)} \\
\mathrm{SID} & \subseteq & A & \text{(Sender Identities)} \\
\mathrm{RID} & \subseteq & A & \text{(Receiver Identities)} \\
\mathrm{PID} & \subseteq & A & \text{(Provenance Store Identities)} \\
\mathrm{CID} & \subseteq & A & \text{(Coordinator Identities)} \\
 & & & \\
\mathcal{M} & = & \mathsf{app} : \mathrm{DATA} \times \mathrm{IK} \times \mathrm{OL} \to \mathcal{M} & \text{(Set of Protocol Messages)} \\
 & & \quad | \ \mathsf{record} : \mathrm{IK} \times \mathrm{VK} \times \mathrm{A} \times \mathrm{VL} \times \mathbb{P}(\mathrm{PA}) \to \mathcal{M} & \\
 & & \quad | \ \mathsf{ack} : \mathrm{IK} \times \mathrm{VK} \to \mathcal{M} & \\
 & & \quad | \ \mathsf{repair} : \mathrm{IK} \times \mathrm{VK} \times \mathrm{DESTPS} \times \mathrm{OL} \to \mathcal{M} & \\
 & & \quad | \ \mathsf{update} : \mathrm{IK} \times \mathrm{VK} \times \mathrm{OL} \to \mathcal{M} & \\
 & & \quad | \ \mathsf{uack} : \mathrm{IK} \times \mathrm{VK} \to \mathcal{M} & \\
\mathcal{K} & = & A \times A \to Bag(\mathcal{M}) & \text{(Set of Channels)} \\
 & & & \\
\mathrm{R} & = & \{m \in \mathcal{M} | m = \mathsf{record}(\kappa, v, a, vl, pas)\} & \text{(Set of Interaction Records)} \\
\mathrm{IK} & = & \mathrm{SID} \times \mathrm{RID} \times N & \text{(Set of Interaction Keys)} \\
\mathrm{VK} & = & \{\mathrm{S}, \mathrm{R}\} & \text{(Set of ViewKinds)} \\
\mathrm{OL} & = & \mathrm{PID} & \text{(Set of Ownlinks)} \\
\mathrm{VL} & = & \mathrm{PID} & \text{(Set of Viewlinks)} \\
\mathrm{DESTPS} & = & \mathrm{PID} & \text{(Set of Destination Stores)} \\
 & & & \\
\mathrm{PA} & = & \mathsf{rel\text{-}pa} : \mathrm{REL} \times \mathrm{EID} \times \mathbb{P}(\mathrm{CID}) \to \mathrm{PA} & \text{(Set of P-Assertions)} \\
 & & \quad | \ \mathsf{i\text{-}pa} : \mathrm{IK} \times \mathrm{DATA} \to \mathrm{PA} & \\
\mathrm{REL} & = & \{r_1, \ldots, r_n\} & \text{(Set of Business Logic Descriptions)} \\
\mathrm{EID} & = & \mathrm{IK} \times \mathrm{VK} & \text{(Set of EffectIDs)} \\
\mathrm{CID} & = & \mathrm{CL} \times \mathrm{IK} \times \mathrm{VK} & \text{(Set of CauseIDs)} \\
\mathrm{CL} & = & \mathrm{PID} & \text{(Set of CauseLinks)} \\
 & & & \\
\mathrm{IN} & = & \mathrm{A} \to \mathbb{P}(\mathrm{IK} \times \mathrm{DATA}) & \text{(Set of In Tables)} \\
\mathrm{ASSERTER} & = & \mathrm{A} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{STR}_\perp \times \mathrm{OL}_\perp \times \mathrm{VL}_\perp \times \mathbb{P}(\mathrm{PA}) & \text{(Set of Asserting Actors)} \\
\mathrm{LOG} & = & \mathrm{A} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{CHANGED}_\perp \times \mathrm{APS}_\perp & \text{(Set of Log Tables)} \\
\mathrm{QUEUE} & = & \mathrm{A} \to Queue(\mathrm{R}) & \text{(Set of Record Queues)} \\
\mathrm{LC} & = & \mathrm{A} \to \mathbb{N} & \text{(Sender's Local Counts)} \\
\mathrm{PSLIST} & = & \mathrm{A} \to \mathbb{P}(\mathrm{PID}) & \text{(Set of Alternative Store Lists)} \\
\mathrm{STR} & = & \{\mathsf{READY}, \mathsf{SEND}, \mathsf{SENT}, \mathsf{ACKED}, \mathsf{OK}\} & \text{(States of Interaction Record)} \\
\mathrm{CHANGED} & = & \{\mathsf{TRUE}, \mathsf{FALSE}\} & \text{(Flags of using alternative PS))} \\
\mathrm{APS} & = & \mathrm{PID} & \text{(Set of Alternative Stores Used)} \\
 & & & \\
\mathrm{TIMER} & = & \mathrm{A} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{STATUS}_\perp \times \mathrm{TIMEOUT} & \text{(Set of Timers)} \\
\mathrm{STATUS} & = & \{\mathsf{ENABLED}, \mathsf{DISABLED}\} & \text{(Set of Timer Statuses)} \\
\mathrm{TIMEOUT} & = & \mathbb{N} & \text{(Set of Timeouts)} \\
 & & & \\
\mathrm{PS} & = & \mathrm{PID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{A}_\perp \times \mathrm{VL}_\perp \times \mathbb{P}(\mathrm{PA}) & \text{(Set of Provenance Stores)} \\
 & & & \\
\mathrm{C} & = & \mathrm{CID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{DESTPS}_\perp \times \mathrm{OL}_\perp & \text{(Set of Coordinators)} \\
\mathrm{UPDATE} & = & \mathrm{CID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{STATE}_\perp & \text{(Set of Update Tables)} \\
\mathrm{STATE} & = & \{\mathsf{UPDATE}, \mathsf{SENT}, \mathsf{UPDATED}, \mathsf{F}\} & \text{(Set of Update States)} \\
 & & & \\
\mathrm{SC} & = & \mathrm{IN} \times \mathrm{ASSERTER} \times \mathrm{QUEUE} \times \mathrm{LOG} \times \mathrm{LC} \times & \\
 & & \mathrm{TIMER} \times \mathrm{PS} \times \mathrm{C} \times \mathrm{UPDATE} \times \mathcal{K} & \text{(Set of Configurations)}
\end{array}
$$

Characteristic Variables:
$a \in A$, $a_s \in \mathrm{SID}$, $a_r \in \mathrm{RID}$, $a_{ps} \in \mathrm{PID}$, $a_c \in \mathrm{CID}$, $m \in \mathcal{M}$, $k \in \mathcal{K}$, $d \in \mathrm{DATA}$, $\kappa \in \mathrm{IK}$, $v \in \mathrm{VK}$, $ol \in \mathrm{OL}$, $vl \in \mathrm{VL}$, $a_{dps} \in \mathrm{DESTPS}$, $pa \in \mathrm{PA}$, $pas \in \mathcal{P}(\mathrm{PA})$, $content \in \mathrm{CONTENT}$, $r \in \mathrm{REL}$, $cids \in \mathcal{P}(\mathrm{CID})$, $cl \in \mathrm{CL}$, $in\_T \in \mathrm{IN}$, $asserter\_T \in \mathrm{ASSERTER}$, $log\_T \in \mathrm{LOG}$, $queue\_T \in \mathrm{QUEUE}$, $lc \in \mathrm{LC}$, $psList \in \mathrm{PSLIST}$, $str \in \mathrm{STR}$, $changed \in \mathrm{CHANGED}$, $aps \in \mathrm{APS}$, $timer\_T \in \mathrm{TIMER}$, $status \in \mathrm{STATUS}$, $to \in \mathrm{TIMEOUT}$, $store\_T \in \mathrm{PS}$, $coord\_T \in \mathrm{C}$, $update\_T \in \mathrm{UPDATE}$, $c \in \mathrm{SC}$

Initial State of Configuration:
$c_i = \langle in\_T_i, asserter\_T_i, log\_T_i, queue\_T_i, lc_i, timer\_T_i, store\_T_i, coord\_T_i, update\_T_i, k_i \rangle$
where:
$asserter\_T_i = \lambda a \kappa v \cdot \langle \perp, \perp, \perp, \emptyset \rangle$, $log\_T_i = \lambda a \kappa v \cdot \langle \perp, \perp \rangle$, $queue\_T_i = \lambda a \cdot \emptyset$,
$lc_i = \lambda a \cdot 0$, $timer\_T_i = \lambda a \kappa \cdot \langle \perp, 0 \rangle$, $store\_T_i = \lambda a \kappa v \cdot \langle \perp, \perp, \emptyset \rangle$,
$coord\_T_i = \lambda a \kappa v \cdot \langle \perp, \perp \rangle$, $update\_T_i = \lambda a \kappa v \cdot \langle \perp \rangle$, $k_i = \lambda a_i a_j \cdot \emptyset$
$in\_T_i = \lambda a \cdot \emptyset$

Fig. 3. System State Space

The log table (log_T) maintains history information of using alternative stores in an asserting actor, used for updating causelinks. A flag ($changed \in$ CHANGED) is set to TRUE if an alternative store was used. The identifier of the final store from which an actor received acknowledgment is remembered in a field ($aps \in$ APS). So that an asserting actor knows which store recorded its p-assertions about an interaction.

After creating interaction records, an actor accumulates them in a local queue, modelled by the table ($queue\_T$), before shipping them to a provenance store. The FIFO property of the queue guarantees successful update of causelinks, detailed later. The notation (LC) defines a function mapping a sender identifier to a natural number so as to distinguish interactions between the sender and receiver. The sender needs to ensure that the natural number is locally unique on the sender side in each interaction. The list of alternative provenance stores are modelled by the set PSLIST, mapping an actor's identity to a set of store identities.

The timer table ($timer\_T$) models the timer used by asserting actors and update coordinators when waiting for acknowledgement messages. The timer's state ($status \in$ STATUS) indicates if the timer is enabled or disabled. A timeout ($to \in$ TIMEOUT) is a natural number, which counts down to zero after the timer is enabled.

PS and Coordinator State Space. The set PS models provenance stores, each containing a table ($store\_T$) indexed by a provenance store's identity. The table maps an interaction key and the view kind of the asserter that created and recorded p-assertions in the interaction to a tuple: the identity of the asserter, a viewlink and the set of p-assertions documenting the interaction. The set C models update coordinators. A update coordinator maintains repair request information in a table ($coord\_T$) and the states of updating a viewlink in another table ($update\_T \in$ UPDATE). We will further detail these tables when we describe the rules of a provenance store and update coordinator.

Given the state space, the ASM is described by an initial state and a set of transitions. A transition is the application of a rule to one configuration to achieve another configuration. Figure 3 contains the initial state ($c_i \in$ SC), which can be summarised as empty channels, empty tables and any local counters being initialised to zero in all actors. The ASM proceeds from this initial state through its execution by going through transitions that lead to new states. These transitions are defined below by the rules of the state machine.

State Machine Rules. The state machine rules are represented using the following notation.

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

...
$pseudo\_statement_n$;
}

Rules are identified by their name and a number of parameters that the rule operates over. Any number of conditions must be met for a rule to fire. Once a rule's conditions are met, the rule fires. 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 new state is achieved after applying all the rule's pseudo-statements to the state that met the conditions of the rule.

We use *send* and *receive* and table update pseudo-statements. Informally, $send(m, a_1, a_2)$ inserts a message m into the communication channel from actor a_1 to actor a_2, and $receive(m, a_1, a_2)$ removes m from the channel. The table update operation puts a message into a table or changes content state in a table. We use the notation $table\_T$ to refer to any table in the system state space. Formally, *send*, *receive* and table update pseudo-statements act as state transformers and are defined as follows.

- If k is the set of message channels of a state $\langle \ldots, k \rangle$, then the expression $send(m, a_1, a_2)$ denotes the state $\langle \ldots, k' \rangle$, where[2] $k'(a_1, a_2) = k(a_1, a_2) \oplus m$, 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 $receive(m, a_1, a_2)$ 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 $table\_T(\ldots).y := V$ denotes the state $\langle \ldots, table\_T', \ldots \rangle$, where $table\_T'(\ldots).x = table\_T(\ldots).x$ if $x \neq y$, and $table\_T'(\ldots).y := V$.

To manipulate an asserting actor's queue, which is used for accumulating interaction records, we define the following operations: $head(q)$, $enqueue(m, q)$ and $dequeue(q)$.

- The expression $head(q)$ returns the head element of queue q.
- The expression $enqueue(m, q)$ denotes $q := q \,\|\, m$, which means m is added at the tail of queue q.
- The expression $dequeue(q)$ denotes $q := tail(q)$, which means the head of queue q is removed.

For convenience, we use notation $a \leftarrow b$ to bind a local variable a to a value b. We then define an assignment operator := for tables. It can assign a value to a field of a table, or assign a tuple to a table as in the following example. In this example, the second field of $asserter\_T(a, \kappa, v)$, i.e., the ownlink ol, is not assigned when $*$ is present.

$$asserter\_T(a, \kappa, v) := \langle \mathsf{OK}, *, PS_2, pas \rangle \equiv \begin{cases} asserter\_T(a, \kappa, v).str := \mathsf{OK} \\ asserter\_T(a, \kappa, v).vl := PS_2 \\ asserter\_T(a, \kappa, v).pas := pas \end{cases}$$

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

Having defined the system state space and ASM rules, we now introduce the rules for asserters (the senders and receivers), provenance stores and update coordinators. These rules precisely define these actors' internal behaviour.

4.2 Asserter Rules in Exchanging Phase

An asserting actor's behaviour can be summarised as two phases: Exchanging and Recording. We firstly describe the Exchanging phase and then introduce the rules of the Recording phase in Section 4.3.

The sender and receiver in an interaction have different rules in the Exchanging phase (Figure 4 and Figure 5). The sender exchanges to the receiver an application message app including application data (d), an interaction key (κ) and the sender's ownlink (i.e., the receiver's viewlink, vl). After receiving an app message, the receiver adds κ and d into table (in_T). Both actor document the exchange of app and an interaction record message is then produced and accumulated in a queue ($queue\_T$). This buffering of interaction records is designed to meet *Transparent Recording* and *Efficient Recording* requirements. It reduces the performance penalty upon the application by allowing the actor to send interaction records when convenient. An asserting actor also initialises several tables, used in the Recording phase.

$send\_app(a_s, a_r, a_{ps}, vl, d, r)$:
//*triggered when d, produced by a function*
//*described by r, is to be sent by* a_s *to* a_r,
//*and when the viewlink, vl, is available.*
→ {
$\kappa \leftarrow newIdentifier(a_s, a_r)$;
$send(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
$pas \leftarrow createPA(a_s, \kappa, d, r)$;
$enqueue(\mathsf{record}(\kappa, S, a_s, vl, pas), queue\_T(a_s))$;
$asserter\_T(a_s, \kappa, S) := \langle \mathsf{READY}, a_{ps}, vl, pas \rangle$;
$log\_T(a_s, \kappa, S) := \langle \mathsf{FALSE}, \perp \rangle$;
}

Fig. 4. The Sender's Rules (Exchanging Phase)

$receive\_app(a_s, a_r, a_{ps}, d, \kappa, vl)$:
$\mathsf{app}(d, \kappa, vl) \in k(a_s, a_r)$
→ {
$receive(\mathsf{app}(d, \kappa, vl), a_s, a_r)$;
$in\_T(a_r) := in\_T(a_r) \oplus \langle \kappa, d \rangle$;
$pas \leftarrow createPA(a_s, \kappa, d, \perp)$;
$enqueue(\mathsf{record}(\kappa, R, a_r, vl, pas), queue\_T(a_r))$;
$asserter\_T(a_r, \kappa, R) := \langle \mathsf{READY}, a_{ps}, vl, pas \rangle$;
$log\_T(a_r, \kappa, R) := \langle \mathsf{FALSE}, \perp \rangle$;
// business logic
}

Fig. 5. The Receiver's Rules (Exchanging Phase)

The function $newIdentifier(a_s, a_r)$ creates a globally unique interaction key, as defined by the following pseudo function. This function requires that senders are responsible for creating interaction keys. This function takes the identities of the sender and the receiver as inputs. It then obtains the local counter of the sender and increases it by one. Finally, a new interaction key using the two actor identities and the local counter is constructed and returned.

Definition
$newIdentifier : \text{SID} \times \text{RID} \rightarrow \text{IK}$
$newIdentifier(a_s, a_r)$:
$lc(a_s) := lc(a_s) + 1$;
return $\langle a_s, a_r, lc(a_s) \rangle$;

In order to define function $createPA(a, \kappa, d, r)$, we firstly define a function $cause(a, d, r)$. It takes an actor identity (a), application data (d), and a business description (r) as input and finds interaction keys of all causes that are related to the production of d.

Definition
$cause : \mathrm{A} \times \mathrm{DATA} \times \mathrm{REL} \to \mathbb{P}(\mathrm{IK})$
$cause(a, d, r)$:
let f_r *be a function described by* r, *such that* $f_r(args) = d$,
where $args = \{\langle \kappa, d' \rangle \mid \langle \kappa, d' \rangle \in in\_T(a)\}$;
return $\{\kappa \mid \langle \kappa, d' \rangle \in args\}$;

Recall that in Figure 5, the data received from application messages is stored in table in_T. In $cause(a, d, r)$, we assume there exists a function f_r that takes some data d' from in_T as input and produces a result data d. Then $cause(a, d, r)$ returns the keys of interactions where all the input data is received.

The $createPA(a, \kappa, d, r)$ function is defined as follows. It takes an actor identity (a), an interaction key (κ), application data (d), and a business logic description (r) to create a set of p-assertions documenting the interaction (indexed by κ) in which d is transferred.

Definition
$createPA : \mathrm{A} \times \mathrm{IK} \times \mathrm{DATA} \times \mathrm{REL} \to \mathbb{P}(\mathrm{PA})$
$createPA(a, \kappa, d, r)$:
$pas \leftarrow$ *if* $r = \bot$
$\{\mathsf{i\text{-}pa}(\kappa, d)\}$;
$\{\mathsf{i\text{-}pa}(\kappa, d), \mathsf{rel\text{-}pa}(r, \langle \kappa, S \rangle, cids)\}$,
where $cids = \{\langle cl, \kappa', R \rangle \mid \kappa' \in cause(a, d, r)$ *and* $cl = asserter\_T(a, \kappa', R).ol\}$;
return pas;

In $createPA(a, \kappa, d, r)$, the created p-assertions must at least include an interaction p-assertion documenting the exchange of an application message that contains κ and d. If d is the consequence of receiving other messages, i.e., $r \neq \bot$, then the sender must make a relationship p-assertion[3] to capture the causal connections between these messages. Function $cause(a, d, r)$ is used here to find keys of cause interactions when creating a relationship p-assertion. An asserting actor may create other application dependent p-assertions, which are not shown in the definition.

4.3 Asserter Rules in Recording Phase

In Recording phase, an asserting actor sends queued record messages to a provenance store and takes remedial actions in response to timeouts. To facilitate presentation, we assume each asserting actor employs a Recording Manager (RM), which monitors the actor's queue and submits record messages to a provenance store. The behaviour of RM is specified in Figure 6 and summarised now.

[3] A relationship p-assertion is always created and recorded in the context of its effect interaction.

$pre\_check(a, \kappa, v, vl, pas)$:
  $queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge asserter\_T(a, \kappa, v).str = \mathsf{READY}$
→ {
  for *each* $pa \in pas$, *such that* $pa = \mathsf{rel\text{-}pa}(r', \langle \kappa, v \rangle, cids')$
    do for *each* $cid \in cids'$
      do $\langle cl', \kappa', v' \rangle \leftarrow cid$;
        if $(log\_T(a, \kappa', v').changed)$, *then*
          $cid' \leftarrow \langle log\_T(a, \kappa', v').aps, \kappa', v' \rangle$;
          $cids'' \leftarrow cids' \ominus cid \oplus cid'$;
      $pa' \leftarrow \mathsf{rel\text{-}pa}(r', \langle \kappa, v \rangle, cids'')$;
      $pas' \leftarrow pas \ominus pa \oplus pa'$;
  if $pas' \neq \bot$
    $asserter\_T(a, \kappa, v) := \langle *, *, *, pas' \rangle$;
  $asserter\_T(a, \kappa, v).str := \mathsf{SEND}$;
}

$send\_record(a, \kappa, v, vl, pas, to)$:
  $queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge asserter\_T(a, \kappa, v).str = \mathsf{SEND}$
→ {
  $a_{ps} \leftarrow asserter\_T(a, \kappa, v).ol$;
  $send(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$;
  $timer\_T(a, \kappa, v) := \langle \mathsf{ENABLED}, to \rangle$;
  $asserter\_T(a, \kappa, v).str := \mathsf{SENT}$;
}

$timer\_click(a, \kappa, v)$:
  $timer\_T(a, \kappa, v).status = \mathsf{ENABLED}$
→ {
  $timer\_T(a, \kappa, v).to := timer\_T(a, \kappa, v).to - 1$;
}

$timeout\_ack(a, \kappa, v)$:
  $timer\_T(a, \kappa, v).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa).to \leq 0$
→ {
  $a'_{ps} \leftarrow random(psList(a))$;
  $log\_T(a, \kappa, v).changed := \mathsf{TRUE}$;
  $timer\_T(a, \kappa, v) := \langle \mathsf{DISABLED}, 0 \rangle$;
  $asserter\_T(a, \kappa, v) := \langle \mathsf{SEND}, a'_{ps}, *, * \rangle$;
}

$receive\_ack(a, a_{ps}, \kappa, v)$:
  $\mathsf{ack}(\kappa, v) \in k(a, a_{ps})$
→ {
  $receive(\mathsf{ack}(\kappa, v), a, a_{ps})$;
  $ol \leftarrow asserter\_T(a, \kappa, v).ol$;
  if $(timer\_T(a, \kappa, v).to > 0 \wedge a_{ps} = ol \wedge asserter\_T(a, \kappa, v).str = \mathsf{SENT})$, *then*
    $dequeue(queue\_T(a))$;
    $timer\_T(a, \kappa, v) := \langle \mathsf{DISABLED}, 0 \rangle$;
    $asserter\_T(a, \kappa, v).str := \mathsf{ACKED}$;
}

$post\_check(a, a_c, \kappa, v)$:
  $asserter\_T(a, \kappa, v).str = \mathsf{ACKED}$
→ {
  if $(log\_T(a, \kappa, v).changed)$, *then*
    $a_{ps} \leftarrow asserter\_T(a, \kappa, v).ol$;
    $a_{dps} \leftarrow asserter\_T(a, \kappa, v).vl$;
    $send(\mathsf{repair}(\kappa, v, a_{dps}, a_{ps}), a, a_c)$;
    $log\_T(a, \kappa, v).aps := a_{ps}$;
  $asserter\_T(a, \kappa, v).str := \mathsf{OK}$;
}

Fig. 6. Asserter rules in Recording phase

- *Updating causelinks.* Given a record message from the queue ($queue\_T(a)$), RM checks and updates causelinks in all relationship p-assertions included in the message (rule *pre_check*). A log table (log_T) maintains a history of the use of alternative provenance stores for each interaction. If the log table shows that an alternative store was used to record p-assertions about a cause interaction, then the corresponding causelink is updated.
- *Submitting a record message.* RM sends a record message to a provenance store and sets timeout when waiting for an ack message (rule *send_record*).
- *Resubmitting a record message.* If RM does not receive an ack when the timeout expires (rule *timeout_ack*), then it infers that failures may have occurred. In this case, RM may resend the record to the same store or use an alternative store if retry attempts to the old store also failed. In order to simplify rules, we do not formalise resending messages to a same provenance store; instead, a new store is selected once a timeout expires. The function $random(psList(a))$ returns an alternative store's identity, selected from a list of candidates. There can be various ways of selecting a store from a list of stores. Here we randomly select one to use.

  Only after an ack is received, can RM eliminate the acknowledged record from the queue (rule *receive_ack*). Checking the state *str* as well as the identifier of the provenance store from which an ack is received help detect duplicate acknowledgements, preserving the correcness of the protocol (rule *receive_ack*).
- *Requesting to update viewlinks.* If an alternative store was used to record a record message, the actor's ownlink known to the opposite actor in an interaction becomes invalid, since the actor's store has changed. Therefore, RM requests a update coordinator to update the opposite actor's viewlink by sending a repair message (rule *post_check*). We note that for a given interaction, an asserting actor at most sends one repair request, which minimises the overhead of taking remedial actions.
- *Updating log table.* If an alternative store was used to record a record message, RM sets $log\_T(a, \kappa, v).changed$ to TRUE (rule *timeout_ack*). After a record message is successfully recorded in a provenance store, RM remembers the provenance store's identity in the log table if $log\_T(a, \kappa, v).changed$ is TRUE (rule *post_check*). This information is to be used for updating causelinks as described above.

We note that the FIFO property of the queue guarantees successful update of causelinks. This is because rule *send_app* and rule *receive_app* enforce that an actor always makes p-assertions about a cause interaction, i.e., where it receives a message, before an effect interaction, i.e., where it sends another message as consequence of received messages. This implies that the record messages about cause interactions are always placed into the queue before that about the effect interaction. Therefore, by monitoring the use of alternative stores when sending record messages, causelinks can be updated successfully. Although current modelling indicates that there is only one queue per asserting actor, which is highly sequential, it can be relaxed by adding a process identifier to $queue\_T(a)$. Then,

each process that an actor participates in can utilise a queue, which enables parallel recording.

4.4 Provenance Store Rules

Figure 7 gives provenance store's rules. A provenance store replies an ack message only *after* it has processed a record message (rule *receive_record*). A store checks if p-assertions about a given interaction exists before processing a record message. This prevents resubmitted record messages from being recorded multiple times.

$receive\_record(a, a_{ps}, \kappa, v, vl, pas)$:
  $\mathsf{record}(\kappa, v, a, vl, pas) \in k(a, a_{ps})$
→ {
  $receive(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$;
  if $(store\_T(a_{ps}, \kappa, v).pas = \emptyset)$, *then*
    $store\_T(a_{ps}, \kappa, v) := \langle a, *, pas \rangle$;
  if $(store\_T(a_{ps}, \kappa, v).vl = \bot)$, *then*
    $store\_T(a_{ps}, \kappa, v).vl := vl$;
  $send(\mathsf{ack}(\kappa, v), a_{ps}, a)$;
}

$receive\_update(a_{ps}, a_c, \kappa, v, \overline{v}, ol)$:
  $\mathsf{update}(\kappa, \overline{v}, ol) \in k(a_c, a_{ps})$
→ {
  $receive(\mathsf{update}(\kappa, \overline{v}, ol), a_c, a_{ps})$;
  $store\_T(a_{ps}, \kappa, v).vl := ol$;
  $send(\mathsf{uack}(\kappa, \overline{v}), a_{ps}, a_c)$;
}

Fig. 7. Provenance Store rules

Since a provenance store may receive an update and a record message related to a same interaction in any order, to achieve requirement *Viewlink Accuracy*, the viewlink obtained from record must NOT overwrite any existing one which may come from an update.

The notation $\overline{v}$ in rule *receive_update* stands for the opposite view in an interaction. For example, if v is the view of the sender, then $\overline{v}$ represents the view of the receiver.

An actor selects an alternative store to record p-assertions if an ack is not received within a timeout. However, it may be the case that the original store still receives and records those p-assertions. This may lead to duplicate information in several stores though, it does not affect the correctness of the protocol, since only the p-assertions successfully acknowledged by an ack can be retrieved using the links updated by the protocol.

4.5 Coordinator Rules

The update coordinator's rules are shown in Figure 8. Upon receiving a repair request (rule *receive_repair*), if there exists request information from the opposite view with regard to the same interaction, which means the coordinator has received a repair message from the other actor, then the coordinator replaces one actor's destination store with the other's ownlink, thus making each actor's destination store correct. Then the update coordinator dispatches two update messages to their respective new destination stores by setting update status to UPDATE (rule *send_update*).

Since a crashing provenance store can be restarted, resending update messages to a same provenance store can be eventually successful (rule *timeout_uack*

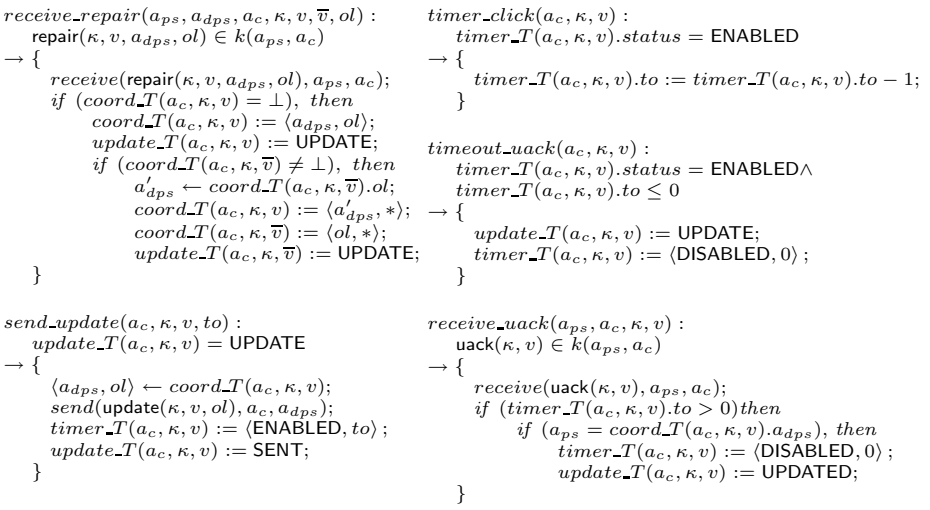

Fig. 8. Coordinator rules

sets $update\_T(a_c, \kappa, v)$ to UPDATE, which will resend update message in rule *send_update*.). This ensures that all requested viewlinks in provenace stores can be updated.

In the current design, we do not specify removing request information maintained in an update coordinator. Request information with regard to an interaction can only be eliminated *after* the coordinator *successfully* updates the provenance store in *each view* of the interaction. If there exists request information for only one view, then the coordinator cannot delete it since it may receive another repair request from the other view. Given that an actor sends out a repair message for an interaction within finite time (due to the use of timeouts in Figure 6), the coordinator can remove any request information with corresponding update status being UPDATED after a reasonably long period of time since the information is recorded. As illustrated above, request information with an update status F cannot be removed.

5 Protocol Analysis

Based on the ASM above, we now analyse F_PReP. The requirements *Guaranteed Recording*, *Viewlink Accuracy* and *CauseLink Accuracy*, identified in Section 2.2, are concerned with the protocol's correctness. We prove that the three requirements are satisfied when the protocol terminates in each interaction. Given that a process consists of a set of interactions, if the protocol can ensure that for each interaction, the three requirements are supported, then the documentation of the whole process is guaranteed to be recorded and retrievable. We have proved the protocol terminates under the assumptions stated in Section 3.1. We now formalise the three requirements as properties and outline the proof of these properties.

Theorem 1 (Guaranteed Recording). *When the protocol terminates, the documentation produced by each asserting actor about the interaction is recorded in provenance stores.*
For any reachable configuration c and for any a, κ, v, the following implication holds when the ASM terminates:

If $asserter\_T(a, \kappa, v).str \neq \bot$, *then*

$$store\_T(a_{ps}, \kappa, v) = \langle a, vl, asserter\_T(a, \kappa, v).pas \rangle,$$

such that $vl \neq \bot$ *and* $a_{ps} = asserter\_T(a, \kappa, v).ol$. □

Theorem 2 (Viewlink Accuracy). *When the protocol terminates, each asserter's viewlink of an interaction is accurate in its provenance store. The viewlink points to the store where the other actor in the interaction recorded p-assertions about the same interaction.*
For any a, a′, κ, v, then the following implication holds when the ASM terminates:
if $asserter\_T(a, \kappa, v).str \neq \bot$, *then*

$$store\_T(a_{ps}, \kappa, v).vl = asserter\_T(a', \kappa, \overline{v}).ol,$$

such that $a_{ps} = asserter\_T(a, \kappa, S).ol$. □

Theorem 3 (Causelink Accuracy). *When the protocol terminates, an asserter's causelinks are accurate in its provenance store. Each points to the store where p-assertions about the corresponding cause interaction are recorded.*
For any a, κ, v, then the following must hold when the protocol terminates:

if $asserter\_T(a, \kappa, v).str \neq \bot$, *then*

$$\begin{array}{l} \textit{for any } pa \in store\_T(a_{ps}, \kappa, v).pas, \textit{ such that } pa = \textsf{rel-pa}(rel, \langle \kappa, v \rangle, cids), \\ \quad \textit{for any } c \in cids, \textit{ let } \langle cl', \kappa', v' \rangle = c, \\ \quad\quad cl' = asserter\_T(a, \kappa', v').ol. \end{array}$$

such that $a_{ps} = asserter\_T(a, \kappa, v).ol$. □

Due to space restriction, we now outline our proof of these properties. Given an arbitrary valid configuration of the ASM, our proofs typically proceed by induction on the length of the transitions that lead to the configuration, and by a case analysis on the kind of transitions. We show that a property is true in the initial configuration of the machine and remains true for every possible transition. This kind of proof is systematic, less error prone and avoids the complications of temporal reasoning.

6 Related Work

Much research has been seen to support recording process documentation, such as Chimera [10], myGrid [26], Karma [24], Kepler [3]. All these systems rely on

their execution environment or specific technologies. The drawback is that the recorded documentation lacks interoperability and hence cannot be shared by different organisations. To promote interoperability, Groth et al. [15] proposed an application and technology independent approach to modelling process documentation in the context of SOAs and developed a generic recording protocol, PReP. All the surveyed systems however do not deal with failures. F_PReP preserves the application and technology independent nature and provides well-defined behaviour while recording documentation in the case of failures.

Redundancy has been widely used to provide fault-tolerance for distributed systems [4]. It involves replicating data or system functionalities, and repeating messages or operations. We adopt the redundancy mechanism in our work, e.g., replicating update coordinators and retransmitting messages. Provenance stores are not suitable to be replicated due to the complexity and significant cost of replicating documentation as explained before.

Atomic transactions typically requires all-or-nothing property to maintain system consistency [12]. It can be an alternative solution to our work. We now discuss a scenario where atomic transaction is applied. We assume that an asserting actor and its provenance store are the two participants in a transaction of recording p-assertions. If the provenance store fails, the actor is notified that the transaction is aborted. Then the actor can select another store to use until the transaction is complete. In this case, the use of atomic transaction provides similar functionality as the remedial actions in our protocol, i.e., selecting an alternative store upon an expired timeout. This approach however is too complicated to be adopted by the fact that each interaction leads to two transactions (sender/receiver).

Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems. There are three rigorous methods, Abstract State Machines (ASM) [17], B[2] and Z[1], that share a common conceptual foundation and are widely used in both academia and industry for the design and analysis of hardware and software systems.

Applying formal methods to the design and reasoning of fault-tolerance has been studied in distributed systems, e.g., distributed database systems[25], control systems[18], and mobile agent systems[19]. The ASM notation we adopt has been used previously to describe a fault-tolerant directory service for mobile agents[21] and PReP. Our proof follows a systematic procedure based on mathematical induction. While done by hand, we believe it is sufficient to provide confidence that the protocol does conform to the properties in Section 5. Previous experience has shown that the ASM formalism is suitable for mechanical proof derivations, and several algorithms[21] have been carried out using a Coq theorem prover[5].

7 Conclusion

In this paper, we have presented a generic protocol, F_PReP, for recording process documentation in the presence of failures. By deriving PReP, F_PReP not

only keeps the generic nature, but also guarantees that process documentation is recorded in the presence of failures. Also, it enables the retrievability of distributed documentation in large scale distributed environments where failures may occur. The protocol is systematically designed and meets the requirements, identified in Section 2, under the assumptions we state on failures.

Our ASM-based formalisation provides a precise and implementation independent means of specifying the protocol. Firstly, it sketches the essence of the protocol and accurately defines required actor's behaviour with unnecessary message fields or messages removed. Secondly, it promotes a rigorous design of the protocol and helps us better understanding the complex behaviour of actors in the presence of failures. With such a formal description, we have successfully identified several deficiencies in the early design of the protocol. Thirdly, the code-like specification is independent of any given programming language or implementation. This enables our protocol to be implemented using different languages and technologies. In summary, the use of a formal notation has significantly improved the design of F_PReP.

F_PReP has been implemented in Java and integrated into a client side p-assertion recording library developed by the University of Southampton. Its performance has been evaluated and the result reveals that it introduces acceptable overhead [7]. We are currently investigating how to create process documentation when an application has its own fault tolerance schemes to tolerate application level failures. In future work, we plan to make use of the process documentation recorded in the presence of failures to diagnose failures.

References

1. Abrial, J., Schuman, S., Meyer, B.: A specification language. On the Construction of Programs, 343–410 (1980)
2. Abrial, J.R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance collection support in the kepler scientific workflow system. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 118–132. Springer, Heidelberg (2006)
4. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1), 11–33 (2004)
5. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saibi, A., Werner, B.: The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203 (1997)
6. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 241–260. Springer, Heidelberg (2006)
7. Chen, Z., Moreau, L.: Implementation and evaluation of a protocol for recording process documentation in the presence of failures. In: Proceedings of Second International Provenance and Annotation Workshop (IPAW 2008), Salt Lake City, USA, June 17-18. Springer, Heidelberg (2008)
8. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design, 4th edn. Addison-Wesley, Reading (2005)

9. Deelman, E., et al.: Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In: e-Science, p. 14. IEEE Computer Society, Los Alamitos (2006)
10. Foster, I.T., Vöckler, J., Wilde, M., Zhao, Y.: The virtual data grid: A new model and architecture for data-intensive collaboration. In: CIDR (2003)
11. Gagliardi, F., Jones, B., Grey, F., Bgin, M.E., Heikkurinen, M.: Building an infrastructure for scientific grid computing: Status and goals of the egee project. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363(1833), 1729–1742 (2005)
12. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco (1993)
13. Groth, P., Miles, S., Fang, W., Wong, S.C., Zauner, K.-P., Moreau, L.: Recording and using provenance in a protein compressibility experiment. In: 14th IEEE International Symposium on HPDC 2005: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings, pp. 201–208. IEEE Computer Society, Washington (2005)
14. Groth, P.: The origin of data: Enabling the determination of provenance in multi-institutional scientific systems through the documentation of processes. Phd thesis, University of Southampton (2007)
15. Groth, P., Jiang, S., Miles, S., Munroe, S., Tan, V., Tsasakou, S., Moreau, L.: An architecture for provenance systems. Technical Report D3.1.1, University of Southampton (February 2006)
16. Groth, P., Luck, M., Moreau, L.: A protocol for recording provenance in service-oriented grids. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 124–139. Springer, Heidelberg (2005)
17. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.): ASM 2000. LNCS, vol. 1912. Springer, Heidelberg (2000)
18. Laibinis, L., Troubitsyna, E.: Refinement of fault tolerant control systems in B. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, pp. 254–268. Springer, Heidelberg (2004)
19. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Rigorous development of fault-tolerant agent systems. In: Butler, et al. (eds.) [6], pp. 241–260
20. Miles, S., Groth, P., Branco, M., Moreau, L.: The requirements of using provenance in e-science experiments. Journal of Grid Computing 5(1), 1–25 (2007)
21. Moreau, L.: A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers. In: The 17th ACM Symposium on Applied Computing (SAC 2002) — Track on Agents, Interactions, Mobility and Systems, Madrid, Spain, pp. 93–100 (March 2002)
22. Moreau, L., Dickman, P., Jones, R.: Birrell's Distributed Reference Listing Revisited. ACM Transactions on Programming Languages and Systems (TOPLAS) 27(4), 52 (2005)
23. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice Hall, Englewood Cliffs (1999)
24. Simmhan, Y.L., et al.: Performance evaluation of the karma provenance framework for scientific workflows. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145. Springer, Heidelberg (2006)
25. Yadav, D., Butler, M.: Rigorous design of fault-tolerant transactions for replicated database systems using event b. In: Butler, et al. (eds.) [6], pp. 343–363.
26. Zhao, J., Wroe, C., Goble, C.A., Stevens, R., Quan, D., Greenwood, R.M.: Using semantic web technologies for representing e-science provenance. In: International Semantic Web Conference, pp. 92–106 (2004)