Recording Accurate Process Documentation in the Presence of Failures

Zheng Chen, 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. The presence of failures may lead to inaccurate process documentation, which does not reflect reality and hence cannot be trustable and utilised. This paper outlines our solution, F_PReP, a protocol for recording accurate process documentation in the presence of failures.

1 Introduction

In scientific and business communities, a wide variety of applications have presented unprecedented requirements [11] 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 the proper decisions were made and the proper procedures were followed for a given patient, there is a need to trace back the origins of these decisions and procedures. In engineering manufacturing, keeping track of the history of generated data in simulations is important for users to analyze 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 users, reviewers and auditors to verify if these transactions are compliant with specific financial regulations.

To meet these requirements, Groth et al. [7] 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 [8], has been developed to provide interoperable means for recording process documentation in the context of service oriented architectures. In their work, process documentation is modelled as a set of assertions (termed *p-assertions*) made by *actors* (i.e., either

clients or services) involved in a process (i.e., the execution of a workflow). Each p-assertion documents some steps of the process, e.g., a client invoked a service or the amount of CPU an actor used in a computation. 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.

Process documentation serves as evidence for what actually happened in computer systems [9]. Users interpretate such a documentation as statements made by actors about what they have observed. Therefore, process documentation should in nature be accurate, i.e., it must document events that happened in a process and must not be based on inferences. Otherwise, users would not trust and utilize it when deriving the provenance of their data products.

Recording process documentation in the presence of failures is an issue that has been lacking attention. PReP assumes a system in which no failure occurs. However, large scale, open distributed systems are not failure-free [4]. For example, a service may not be available and network connection may be broken. Failures can lead to inaccurate process documentation: documentation may fail to describe events that occurred, it may describe events that did not happen, or the pointer chain may be broken. Inaccurate process documentation can have disastrous consequences. For example, in a provenance-based service billing system, if a user invoked a service, but documentation fails to describe this invocation, or if a user failed to invoke a service, but its recorded documentation reveals that the invocation occurred, then the user will be charged too little or too much, respectively, which is not acceptable. Also, process documentation distributed in multiple provenance stores may not be retrievable in its entirety because the pointer chain may be broken due to failures.

To address these problems, we have designed a protocol, F_PReP [3], to record accurate process documentation in the presence of failures. It consists of three phases: Exchanging, Recording and Updating. In Exchanging phase, two actors exchange an application message and produce documentation describing the exchange of that message. Asynchronously, in Recording phase, both actors submit their documentation to their respective provenance store. F_PReP provides guaranteed recording of documentation in the presence of failures through the use of redundant provenance stores. If the pointer chain is broken in the two phases, the Updating phase begins. A novel component, Recovery Coordinator, is introduced to fix any broken pointer chain. The protocol has been formalised as an abstract state machine and its correctness has been proved. In this paper, we outline F_PReP, and introduce its formalisation and proof.

2 F_PReP Overview

Terminology and Requirements Process documentation describes a process that led to a result. Such a process is modelled as a set of interactions between actors

involved in that process [7]. Each *interaction* is concerned with one application message exchanged between two actors, i.e., the sender and the receiver. Each actor documents the interaction using p-assertions and records them in a provenance store. Since the two *asserting actors* in an interaction may use different stores, they must also record a pointer, termed *viewlink*, indicating where the other actor records its p-assertions. After repeating these actions for all interactions of a process, the documentation of that process is obtained resulting in a bidirectional pointer chain, connecting all the stores hosting the documentation of that process. Therefore, to record *accurate* process documentation, we need to ensure that each *interaction record*, i.e., the documentation of an interaction, is accurate preserving the following properties: (1) Each actor's p-assertions describe what actually happened in that interaction from that actor's viewpoint (*Assertion Accuracy*); (2) Each actor's viewlink points to the provenance store where the other actor recorded p-assertions in that interaction (*ViewLink Accuracy*). In addition, the protocol needs to enforce that (3) all p-assertions produced by each actor in an interaction and the actor's viewlink must be recorded in a provenance store in the presence of failures (*Documentation Availability*).

Assumptions Our failure assumptions are the following: asserting actors, provenance stores may crash, i.e., they halt and stop any further execution, and never recover. However, we assume that there is no failure in a provenance store's persistent storage. We assume that each asserting actor keeps a list of provenance store addresses and at least one store is available. Communication channels can lose and reorder, but not duplicate messages. Process documentation may be inaccurate when an actor is maliciously recording incorrect information. However, we assume this case does not happen.

To ensure separation of concerns, each actor employs a Communication Agent (CA) to send/receive messages to/from other actors. We assume that the sender CA can use acknowledgement, timeout and retransmission to reliably deliver a message, and report the delivery outcome to the sending actor: *success* or *failure*. It reports *success* notification if the message is acknowledged, indicating that the receiver CA has received that message. It also reports *failure* notification if the message fails to be delivered or it is not acknowledged even after retry attempts. In this way, our protocol does not have to handle communication details but can focus on actions in response to the notifications (in sending actors) and messages (in receiving actors) provided by CA.

Exchanging Phase In the exchanging phase, two actors, the sender S and the receiver R, exchange an application message *app* and document the interaction, as demonstrated in Figure 1. To facilitate creating and recording interaction records, each actor employs a Recording Manager (RM). In order to form a pointer chain, the two actors also exchange a pointer to their respective provenance store. For this purpose, S embeds its pointer in *app*, while R informs S with its pointer in a separate message *linkr*. Meanwhile, each actor creates an interaction record which includes the p-assertions describing the interaction, and a viewlink, i.e., the other actor's pointer. The created interaction record

is accumulated in RM before being sent to a provenance store. This buffering of interaction records is designed to reduce the performance penalty upon the application by allowing the actor to send interaction records when convenient.

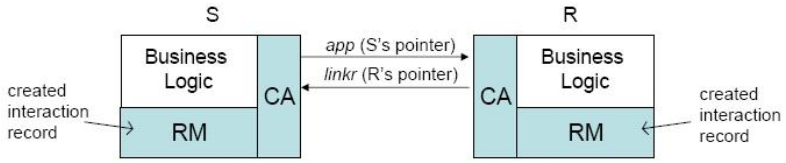

Fig. 1. Exchanging Phase

In order to create an *accurate* interaction record, an actor must only assert facts that it can observe. Hence, we specify some rules for asserting actors to follow. (1) S must assert that an interaction occurred if and only if it receives SUCCESS notification from its CA for delivering *app* message; (2) S must assert failure information when receiving FAILURE notification from its CA for delivering *app* message. One reason for this rule is that failure information provides evidence that an interaction was attempted even if that interaction failed. Without such information, there would be no record of the attempted interaction; (3) S must record R's pointer as its viewlink if it receives the pointer; (4) R must assert that an interaction occurred after it receives *app* message; (5) R must record S's pointer as its viewlink after it receives the pointer; (6) R must assert failure information when receiving a FAILURE notification from its CA for delivering *linkr*. This is because S may not receive the pointer, disconnecting the chain. In this case, S takes no action and the assertion made by R will be used to fix the broken chain in Updating phase; (7) S and R may generate application specific p-assertions.

Recording Phase F_PReP provides guaranteed recording of interaction record in the presence of failures through the use of redundant provenance stores. Figure 2 illustrates this phase. In Step 1, an actor's RM submits an interaction record to a provenance store PS. In Step 2, PS stores the interaction record that it receives in its persistent storage. After successfully recording the interaction record, it replies the submitting actor with an acknowledgement (Step 3). We have assumed that there is no failure in persistent storage; hence any interaction records stored in a provenance store's persistent storage remain available forever. If the actor's RM receives FAILURE notification from CA for delivering the interaction record or it does not receive the acknowledgement before a timeout, then it can imply that failures may have occurred, e.g., PS has crashed. In the two cases, the RM may resend the interaction record to PS. Since a crashing provenance store can no longer be used for further recording, the RM needs to use an alternative store after retry attempts also fail. We have assumed that each asserting actor keeps a list of provenance store addresses and at least one

store is available, therefore, the use of redundant stores ensures that an actor's interaction record is eventually recorded. Only after the acknowledgement is received, can the RM eliminate the accumulated interaction record. The use of an alternative store would result in a broken pointer chain if an actor's original pointer has been sent to the other actor, which now does not point to a correct location. Hence, the RM needs to add an assertion documenting the use of an alternative store in its interaction record so that actions can be taken to fix any broken chain in the next phase.

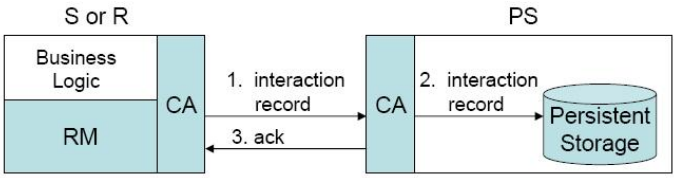

Fig. 2. Recording Phase

Updating Phase In this phase, the protocol updates an actor's viewlink in order to fix a potentially broken pointer chain. A pointer chain may be broken in two situations, as demonstrated in Figure 3. (1) R gets a FAILURE notification when sending *linkr* to S in Step 2, hence S may not know R's pointer; (2) If an actor, say S, does not successfully record its interaction record in Step 3 and selects an alternative store, say $PS1'$, to submit the record, then S's pointer sent to R in Step 1 does not point to the correct location, $PS1'$. In either case, an actor has made an assertion documenting failure information when delivering *linkr* or the use of an alternative store, as described in the previous two phases. We use BROKEN to denote any of the two assertions in Step 4, since either assertion documents a fact that may cause a broken pointer chain. Upon receipt of a BROKEN, a provenance store requests a novel component, Recovery Coordinator, to facilitate repairing the broken chain (Step 5). By taking remedial actions, the Recovery Coordinator updates the viewlink in a destination store (Step 6) with any broken pointer chain fixed. In the example of Figure 3, when the protocol terminates, $PS1'$ has a viewlink to $PS2$ and vice versa.

We assume that *the Recovery Coordinator does not fail.* There is only one Recovery Coordinator, so we can use traditional fault-tolerant mechanisms such as replication to ensure its reliability and availability. Recovery Coordinator is necessary to fix a broken pointer chain, since both actors in an interaction may each report a BROKEN, as shown in Figure 3. In this case, direct communication between two actors' provenance stores does not help, because at that moment, one does not know which store the other actor is using. For example, in the figure, R does not know that S is using $PS1'$ and S does not know where R's p-assertions are stored. Assuming that the pointer chain is not broken frequently,

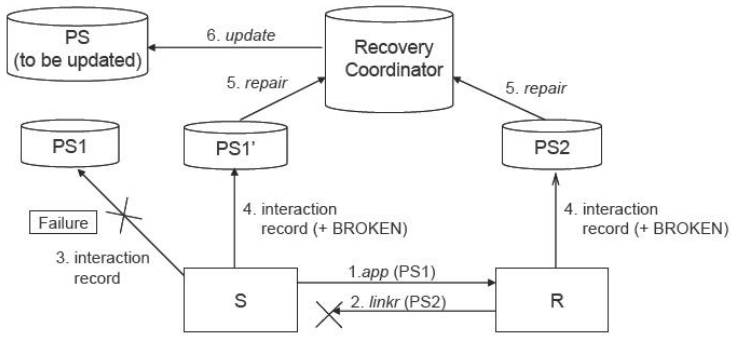

Fig. 3. Updating Phase

the Recovery Coordinator is not involved in each interaction and hence does not affect the system's scalability, despite being centralised.

3 Protocol Analysis and Formalisation

Protocol Analysis The agreement on interaction occurrence may not be reached by the sender and receiver of an application message. If the sender gets a *failure* notification, it is impossible for it to decide whether the receiver has received that message or not [10]. If the receiver happens to receive the application message, an inconsistency occurs, i.e., the sender asserts failure information while the receiver documents that the interaction has occurred. Such an inconsistency reflects the difference between the sender and receiver's knowledge of an interaction, which does not contradict the *Assertion Accuracy* requirement. Therefore, our protocol does not need to remove such an inconsistency.

Concurrency is a major concern to the correctness of the protocol. The protocol specifies actions for asserting actors, provenance stores, and Recovery Coordinator, which may co-operate with one another. On receiving messages from different components, the receiver has to respond properly against all the possible message arrival orders. For example, a provenance store may concurrently receive an interaction record from an asserting actor and an *update* message from Recovery Coordinator; Recovery Coordinator may receive two *repair* messages from two provenance stores in any order. The concurrency issue hence requires us to rigorously design the protocol.

Formalisation F_PReP has been formalised through the use of an abstract state machine (ASM). The machine's behaviour is described by states and transitions between those states. Such a formalisation provides a precise, implementation-independent means of describing the system.

Figure 4 shows the system state space. We identify specific subsets of actors in the system, namely, senders, receivers, provenance stores, and Recovery

Coordinator. Protocol messages are sent over communication channels, denoted by $\mathcal{K}$. Since no assumption is made about message order in channels, channels are represented as bags of messages between pairs of actors. 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, ID and OL[1]. The set of all messages $\mathcal{M}$ is defined as the union of these message sets. CA's notifications are modelled as two messages: failure(m) and success(m). The power set notation (P) denotes that there can be more than one of a given element. We specify several notations representing p-assertions. The notation Occurrence stands for an assertion which documents the occurrence of an interaction. The notation FailureInfo(pa) denotes any assertions describing failure information during the delivery of an application message, while Broken(pa) represents any assertions documenting a fact that may cause a broken pointer chain.

The internal functionality of each kind of processes is modelled as follows. (1) The set ACTOR models all the asserting actors, each identified by an actor identity. Informally, each asserting actor contains a table ($actor\_T$) that maps an interaction identity (id) and the actor's view kind (v), i.e., Sender or Receiver, to a tuple: the state of its ownlink (stl), the state of an interaction record (st), the actor's ownlink (ol) and its viewlink (vl). An asserting actor's p-assertions are accumulated in a message queue before being sent to a provenance store. The queue is modelled by a table ($queue\_T$). The table $timer\_T$ maintains timing information such as timer status, current time, timing interval and timeout, which is used by an asserting actor's RM in Recording phase. The notation LC defines a function that maps a sender identifier to a natural number, used to distinguish interactions between the sender and receiver. The sender needs to ensure that the natural number is locally unique in each interaction. (2) The set PS models provenance stores, each identified by an actor identity. Each provenance store contains a table ($store\_T$) that maps an interaction identity (id) and a view kind (v) to a tuple: recorded p-assertions (pas) and a viewlink (vl). (3) The set C models coordinators. There is only one coordinator, identified by a_c, and it also keeps a table ($coord\_T$). For each interaction (id) and for each view kind (v) in the interaction, the table stores a tuple: the destination provenance store (a_{dps}) to be updated and the ownlink (ol) of the asserting actor with this view kind.

Given the state space, the ASM is described by an initial state and a set of transitions. Figure 4 contains the initial state, which can be summarised as empty channels and empty tables in all processes. The ASM proceeds from this state through its execution by going through transitions that lead to new states.

The permissible transitions in the ASM are described through rules. Rules are identified by their name and a number of parameters that the rule operates over. 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

[1] An asserting actor's *ownlink*, OL, refers to the provenance store where the asserting actor records its *own* p-assertions.

$$
\begin{array}{rcll}
A & = & \{a_1, a_2, \ldots, a_n\} & \text{(Set of Actor Identities)} \\
\text{SID} & \subset & A & \text{(Sender Identities)} \\
\text{RID} & \subset & A & \text{(Receiver Identities)} \\
\text{PID} & \subset & A & \text{(Provenance Store Identities)} \\
a_c & \subseteq & A & \text{(Coordinator Identity)} \\
 & & & \\
\mathcal{M} & = & \mathsf{app} : \textsc{Data} \times \text{ID} \times \text{OL} \to \mathcal{M} & \text{(Application Messages)} \\
 & & \mid \mathsf{linkr} : \text{ID} \times \text{VK} \times \text{OL} \to \mathcal{M} & (R\text{'s Ownlink Messages}) \\
 & & \mid \mathsf{record} : \text{ID} \times \text{VK} \times \text{VL} \times P(\textsc{PAssertion}) \to \mathcal{M} & \text{(Interaction Record Messages)} \\
 & & \mid \mathsf{ack} : \text{ID} \times \text{VK} \to \mathcal{M} & \text{(Record Ack Messages)} \\
 & & \mid \mathsf{repair} : \text{ID} \times \text{VK} \times \textsc{DestPS} \times \text{OL} \to \mathcal{M} & \text{(Repair Messages)} \\
 & & \mid \mathsf{update} : \text{ID} \times \text{VK} \times \text{OL} \to \mathcal{M} & \text{(Update Messages)} \\
 & & \mid \mathsf{failure} : \mathcal{M} \to \mathcal{M} & \text{(Failure Notifications)} \\
 & & \mid \mathsf{success} : \mathcal{M} \to \mathcal{M} & \text{(Success Notifications)} \\
 & & & \\
\text{ID} & = & \{id_1, id_2, \ldots, id_n\} & \text{(Set of Interaction Identifiers)} \\
\text{VK} & = & \{\text{S, R}\} & \text{(Set of ViewKinds)} \\
\text{OL} & = & \text{PID} & \text{(Set of an Actor's Ownlinks)} \\
\text{VL} & = & \text{PID} & \text{(Set of an Actor's Viewlinks)} \\
\textsc{DestPS} & = & \text{PID} & \text{(Set of Destination Stores)} \\
\textsc{PAssertion} & = & \{\mathsf{Occurrence}, \mathsf{FailureInfo}(pa), \mathsf{Broken}(pa), pa_1, \ldots, pa_n\} & \text{(Set of P-Assertions)} \\
 & & & \\
\text{ACTOR} & = & A \to \text{ID} \times \text{VK} \to \text{STL} \times \text{STR} \times \text{OL} \times \text{VL} & \text{(Set of Asserting Actors)} \\
\text{STL} & = & \{\bot, \mathsf{SENT}, \mathsf{F}, \mathsf{OK}\} & \text{(States of an OwnLink)} \\
\text{STR} & = & \{\bot, \mathsf{SENT}, \mathsf{OK}\} & \text{(States of Interaction Record)} \\
 & & & \\
\text{QUEUE} & = & A \times \text{ID} \to Bag(\textsc{PAssertion}) & \text{(Set of Quened P-Assertions)} \\
 & & & \\
\text{TIMER} & = & A \times \text{ID} \to \textsc{Status} \times \textsc{CurrentTime} \times & \\
 & & \textsc{Interval} \times \textsc{Timeout} & \text{(Set of Timers)} \\
\textsc{Status} & = & \{\bot, \mathsf{Enabled}, \mathsf{Disabled}\} & \text{(Set of Timer Statuses)} \\
\textsc{CurrentTime} & = & \{t_1, t_2, \ldots, t_n\} & \text{(Set of Current Times)} \\
\textsc{Interval} & = & \{\Delta_1, \Delta_2, \ldots, \Delta_n\} & \text{(Set of Timing Intervals)} \\
\textsc{Timeout} & = & \{to_1, to_2, \ldots, to_n\} & \text{(Set of Timeouts)} \\
 & & & \\
\text{LC} & = & \text{SID} \to N & \text{(Sender's Local Counts)} \\
 & & & \\
\text{PS} & = & A \to \text{ID} \times \text{VK} \to \text{VL} \times P(\textsc{PAssertion}) & \text{(Set of Provenance Stores)} \\
 & & & \\
\text{C} & = & A \to \text{ID} \times \text{VK} \to \textsc{DestPS} \times \text{OL} & \text{(Set of Coordinators)} \\
 & & & \\
\mathcal{K} & = & A \times A \to Bag(\mathcal{M}) & \text{(Set of Channels)}
\end{array}
$$

Characteristic Variables:
$a \in A$, $a_s \in \text{SID}$, $a_r \in \text{RID}$, $a_{ps} \in \text{PID}$, $m \in \mathcal{M}$, $d \in \textsc{Data}$, $pa \in \textsc{PAssertion}$, $pas \in P(\textsc{PAssertion})$, $id \in \text{ID}$, $v \in \text{VK}$, $a_{dps} \in \textsc{DestPS}$, $ol \in \text{OL}$, $vl \in \text{VL}$, $stl \in \text{STL}$, $str \in \text{STR}$, $status \in \textsc{Status}$, $t \in \textsc{CurrentTime}$, $\Delta \in \textsc{Interval}$, $to \in \textsc{Timeout}$, $actor\_T \in \text{ACTOR}$, $queue\_T \in \text{QUEUE}$, $timer\_T \in \text{TIMER}$, $lc \in \text{LC}$, $store\_T \in \text{PS}$, $coord\_T \in \text{C}$, $k \in \mathcal{K}$

Configuration: $c = \langle actor\_T, queue\_T, timer\_T, store\_T, coord\_T, k \rangle$

Initial State: $c_i = \langle actor\_T_i, queue\_T_i, timer\_T, store\_T_i, coord\_T_i, k_i \rangle$
where:
$actor\_T_i = \lambda a \lambda id v \cdot \langle \bot, \bot, \bot, \bot \rangle$, $queue\_T_i = \lambda a id \cdot \emptyset$,
$timer\_T_i = \lambda a \lambda id v \cdot \langle \bot, \bot, \bot, \bot \rangle$, $store\_T_i = \lambda a \lambda id v \cdot \langle \bot, \emptyset \rangle$,
$coord\_T_i = \lambda a \lambda id v \cdot \langle \bot, \bot \rangle$, $k_i = \lambda a_i a_j \cdot \emptyset$

Fig. 4. System State Space

```
send_app(a_s, a_r, id, ls, d) :
    id = newIdentifier(a_s, a_r)
→ {
      send(app(d, id, ls), a_s, a_r);
      actor_T(a_s)(id, S) := ⟨SENT, ⊥, ls, ⊥⟩ ;
    }

failure_app(a_s, a_r, id, ls, d) :
    failure(app(d, id, ls)) ∈ k(a_s, a_r)∧
    {FailureInfo(pa), pa_2, ..., pa_n} = createPA()
→ {
      receive(failure(app(d, id, ls)), a_s, a_r);
      queue_T(a_s, id) := queue_T(a_s, id)⊕
      {FailureInfo(pa), pa_2, ..., pa_n} ;
      actor_T(a_s)(id, S).stl := F;
    }

success_app(a_s, a_r, id, ls, d) :
    success(app(d, id, ls)) ∈ k(a_s, a_r)∧
    {Occurrence, pa_2, ..., pa_n} = createPA()
→ {
      receive(success(app(d, id, ls)), a_s, a_r);
      queue_T(a_s, id) := queue_T(a_s, id)⊕
      {Occurrence, pa_2, ..., pa_n} ;
      actor_T(a_s)(id, S).stl := OK;
    }

receive_linkr(a_s, a_r, id, lr) :
    linkr(id, R, lr) ∈ k(a_s, a_r)
→ {
      receive(linkr(id, R, lr), a_r, a_s);
      actor_T(a_s)(id, S).vl := lr;
    }
```

Fig. 5. The Sender's rules in Exchanging phase

```
receive_app(a_s, a_r, id, d, ls, lr) :
    app(d, id, ls) ∈ k(a_s, a_r)∧
    {Occurrence, pa_2, ..., pa_n} = createPA()
→ {
      receive(app(d, id, ls), a_s, a_r);
      queue_T(a_r, id) := queue_T(a_r, id)⊕
      {Occurrence, pa_2, ..., pa_n} ;
      send(linkr(id, R, lr), a_r, a_s);
      actor_T(a_r)(id, R) := ⟨SENT, ⊥, lr, ls⟩ ;
      // business logic
    }

failure_linkr(a_s, a_r, id, lr) :
    failure(linkr(id, R, lr)) ∈ k(a_s, a_r)∧
    {Broken(pa)} = createPA()
→ {
      receive(failure(linkr(id, R, lr)), a_r, a_s);
      queue_T(a_r, id) := queue_T(a_r, id)⊕
      {Broken(pa)} ;
      actor_T(a_r)(id, R).stl := F;
    }

success_linkr(a_s, a_r, id, lr) :
    success(linkr(id, R, lr)) ∈ k(a_s, a_r)
→ {
      receive(success(linkr(id, R, lr)), a_r, a_s);
      actor_T(a_r)(id, R).stl := OK;
    }
```

Fig. 6. The Receiver's rules in Exchanging phase

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. A rule's pseudo-statements consist of a set of *send*, *receive* and table update operations. Informally, $send(m, a_1, a_2)$ inserts a message m into the channel from actor a_1 to actor a_2, and $receive(m, a_1, a_2)$ removes m from the channel between a_1 and a_2. A table update operation places a message into a table or changes the state of a table field.

Due to space restriction, we only give the Sender and Receiver's rules in Exchanging Phase in Figure 5 and 6. These rules precisely specify an asserting actor's behaviour described in Section 2. The whole set of rules can be found at [3]. The function $newIdentifier(a_s, a_r)$ creates a globally unique interaction identifier, which can be a tuple consisting of the sender and receiver's identity plus a locally unique number managed by the sender, expressed as $\langle a_s, a_r, lc(a_s)\rangle$. The function $createPA()$ generates a set of p-assertions and the operator $\oplus$ denotes union on bags.

The properties *Documentation Availability*, *Assertion Accuracy* and *ViewLink Accuracy* have been formalised as the following invariants. The notations v and $\overline{v}$ stand for the two views in an interaction and $actor\_T(a_s, id, S).stl = \mathsf{OK}$ marks

the occurrence of an interaction.

Documentation Availability:
(P1) *If* $actor\_\mathcal{T}(a_s)(id,S).stl = \mathsf{OK}$, *then*
$\forall v \in \mathrm{VK}$, $\mathsf{Occurrence} \in store\_\mathcal{T}(a_{ps})(id,v).pas \wedge store\_\mathcal{T}(a_{ps})(id,v).vl \neq \perp$,
such that $a_{ps} = actor\_\mathcal{T}(a_v)(id,v).ol$.
(P2) *If* $actor\_\mathcal{T}(a_s)(id,S).stl = \mathsf{F}$, *then*
$\mathsf{FailureInfo}(pa) \in store\_\mathcal{T}(a_{ps})(id,S).pas$, *such that* $a_{ps} = actor\_\mathcal{T}(a_s)(id,S).ol$.
Assertion Accuracy:
(P3) *If* $\forall v \in \mathrm{VK}$, $\mathsf{Occurrence} \in store\_\mathcal{T}(a_{ps})(id,v).pas$, *then*
$actor\_\mathcal{T}(a_s)(id,S).stl = \mathsf{OK}$, *such that* $a_{ps} = actor\_\mathcal{T}(a_v)(id,v).ol$.
(P4) *If* $\mathsf{FailureInfo}(pa) \in store\_\mathcal{T}(a_{ps})(id,S).pas$, *then*
$actor\_\mathcal{T}(a_s)(id,S).stl = \mathsf{F}$, *such that* $a_{ps} = actor\_\mathcal{T}(a_s)(id,S).ol$.
Viewlink Accuracy:
(P5) *If* $actor\_\mathcal{T}(a_s)(id,S).stl = \mathsf{OK}$, *then*
$\forall v \in \mathrm{VK}$, $store\_\mathcal{T}(a_{ps})(id,v).vl = actor\_\mathcal{T}(a_{\overline{v}})(id,\overline{v}).ol$,
such that $a_{ps} = actor\_\mathcal{T}(a_v)(id,v).ol$.

We have proved that these invariants hold when the protocol terminates. 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 can be easily encoded in a mechanical theorem prover.

4 Related Work and Conclusion

Much research has been seen to support recording process documentation, e.g., Chimera [5], myGrid [12], PReP [8, 7] and Kepler [1]. From an analysis of these works, only PReP provides an application-independent solution to recording process documentation. However, all the surveyed systems either assume a failure-free execution environment or do not discuss this issue.

Redundancy has long been used as a means to provide fault tolerance in distributed systems [2]. Key components may be replicated (replication in space) or re-executed (replication in time) to protect against hardware malfunctions or transient system faults. Our work adopts this mechanism through the use of redundant provenance stores and retransmission of messages.

Distributed transactions typically requires *all-or-nothing* atomicity to maintain system consistency [6]. The *all-or-nothing* property is not applicable to our work. Assume that the asserting actors and provenance stores are the participants in a transaction. If any participant fails after the interaction took place, then the recording action is aborted and hence the documentation about that interaction cannot be recorded. This is not desired since process documentation must reflect reality and document events that happened in a process. As

long as the interaction has occurred, its documentation must be recorded in a provenance store.

In conclusion, we have presented a protocol F_PReP for recording accurate process documentation in the presence of failures. Compared with PReP, F_PReP not only keeps the application-independent nature, but also guarantees that process documentation is accurate and available in a provenance store in the presence of failures. Also, it enables distributed process documentation to be still retrievable in large scale distributed environments where failures may occur. The protocol is being implemented and its performance will be evaluated in future work.

References

1. Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 118–132, 2006.
2. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
3. Zheng Chen. Accurately recording process documentation in the presence of failures. Mini thesis, School of Electronics and Computer Science, University of Southampton, UK, http://www.ecs.soton.ac.uk/~zc05r/protocol, 2007.
4. Ewa Deelman and et. al. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *Proceedings of the e-Science 2006 Conference in Amsterdam, the Netherlands*, December 2006.
5. Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. The virtual data grid: A new model and architecture for data-intensive collaboration. In *CIDR*, 2003.
6. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
7. Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An architecture for provenance systems. Technical Report D3.1.1, University of Southampton, February 2006.
8. Paul Groth, Michael Luck, and Luc Moreau. A protocol for recording provenance in service-oriented grids. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS'04)*, France, 2004.
9. Paul Groth, Simon Miles, and Steve Munroe. Principles of high quality documentation for provenance: A philosophical discussion. In *Proceedings of Third International Provenance and Annotation Workshop, Chicago*, 2006.
10. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.
11. Simon Miles, Paul Groth, Miguel Branco, and Luc Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 2006.
12. Jun Zhao, Chris Wroe, Carole A. Goble, Robert Stevens, Dennis Quan, and R. Mark Greenwood:. Using semantic web technologies for representing e-science provenance. *International Semantic Web Conference*, pages 92–106, 2004.