UNIVERSITY OF SOUTHAMPTON

Recording Process Documentation in the Presence of Failures in Service Oriented Architectures

by

Zheng Chen

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

September 2009

DECLARATION OF AUTHORSHIP

I, ZHENG CHEN declare that the thesis entitled

Recording Process Documentation in the Presence of Failures in Service Oriented Architectures

and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- none of this work has been published before submission, **or** [delete as appropriate] parts of this work have been published as: [please list references]

Signed: ……

Date:……

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Zheng Chen

Scientific and engineering communities (e.g., chemistry, bioinformatics and engineering manufacturing) have presented unprecedented requirements for knowing the *provenance* of their data products, i.e., where they originated from, how they were produced and what has happened to them since creation. Without such important knowledge, scientists and engineers cannot reproduce, analyse or validate experiments and processes.

Previous work has conceived a computer-based representation of a past process for determining provenance, termed *process documentation*. However, current provenance systems do not adequately address the problem of reliably recording process documentation in large scale environments like Service Oriented Architectures. For example, a service may not be available and network connection may be broken. In this context, reliably recording process documentation becomes challenging, given that the documentation produced in a process can be spread over multiple provenance repositories across the world.

The presence of failures (specifically, the crash of provenance repositories and communication failures) may prevent process documentation from being recorded, losing the evidence that a process occurred. This would have disastrous consequences and hence is not acceptable in the domains that rely on process documentation to determine the provenance of their data products.

In this thesis, we systematically analyse all situations that may occur during capturing process documentation in the event of assumed failures. We then present a novel coordinator-based protocol that is formally proved to record complete process documentation. In addition, we use graphs to intuitively represent the topology of process documentation recorded in multiple interlinked provenance repositories, which helps us to investigate the entire retrievability of distributed process documentation. Finally, we evaluate a system architecture that employs the protocol and supports practical issues such as communication, storage and performance. The results show that the system can record complete and retrievable process documentation while maintaining acceptable performance.

Contents

List of Figures

List of Tables

Acknowledgements

To my family in China, who have been a constant source of unwavering support throughout the duration of my PhD studies: this thesis is dedicated to them. My achievement here is as much theirs as it is mine.

A big note of gratitude is also due to my supervisor, Professor Luc Moreau, for his guidance and patience throughout the past four years. Without Luc this work would not have been possible.

I would like to thank my colleagues at IAM group, especially Paul Groth, Simon Miles, Victor Tan, Weijian Fang and David Baker. Thank you guys for answering questions, in-depth chats and constant supports.

The process of this Ph.D. was a long journey, in which I have not felt lonely because of my friends at Southampton. I would like to thank all my friends whom I have been privileged to know here. The memories with you all will continue to be treasured for a long time.

A note of thanks is due to all my friends in China who have, from time to time, maintained a lively correspondence through e-mail and msn. It is edifying to know that friendships can be maintained over distance and time.

At times our own light goes out and is rekindled by a spark from another person. Each of us has cause to think with deep gratitude of those who have lighted the flame within us - Albert Schweitzer

Chapter 1

Introduction

1.1 Provenance

The Oxford English Dictionary defines *provenance* as *(i) the fact of coming from some particular source or quarter; origin, derivation. (ii) the history or pedigree of a work of art, manuscript, rare book, etc.; concretely, a record of the ultimate derivation and passage of an item through its various owners.*

In the field of fine art, provenance has been well studied as the trusted and documented history of an art object [60]. For example, after a painting has been drawn, people who study it many years later would want to know who drew it, where and when it was produced, what materials and techniques were used to draw it, and who has owned it since it was created. These questions can be answered through a record that accurately documents all the information relevant to this painting.

In scientific and engineering communities, provenance explains the history of data products, e.g., where they originated from and what has happened to them since creation. With such information, people can interpret and judge the quality of data, and consequently derive trust in results produced by applications. In chemistry experiments, provenance can be used to detail the procedure by which a material is generated, allowing the material to be patented [132]. 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 [11]. In engineering manufacturing, engineers choose materials for the design of critical components, such as for an airplane, based on their statistical analysis and it is essential to establish the history of this data to prevent system failures and for audit [146]. In finance business, the provenance of some data item establishes the origin and authenticity of the data item (e.g., a trade execution) produced by financial transactions, enabling reviewers and auditors to verify if these transactions are compliant with specific financial regulations [91].

Traditionally, people typically use a log book to record provenance information of their work. For example, scientists manually record every step, operation and intermediate result of their experiment in a log book as the experiment is being executed. Therefore, they can determine the provenance of experimental results by going back through their log book. However, if the experiment is entirely *in silico*[1], scientists are not able to track their work since there are no physical records as there would be with a log book in those lab-based experiments.

To address this problem, much research has been seen to collect provenance information in computer systems (e.g., [63, 130, 131, 175, 54, 155, 10, 151]). The drawback of these provenance systems is that they are tightly coupled with specific application domains or technologies. This means that application developers have to re-implement components for recording or using provenance in different execution contexts. In addition, it becomes difficult to integrate provenance derived from different systems and represented using different models. One direction currently being investigated is the standardisation of representing provenance [41].

To achieve this goal, Groth et al. [77, 82, 80] designed and implemented a domain and technology independent infrastructure, PASOA, to provide interoperable means for recording and using provenance. It supports requirements from very different application areas [117], such as biology, chemistry, physics and computer science, and has been evaluated in a number of applications [44, 11, 157, 169, 158, 9, 96, 167, 140, 118].

PASOA defines several terms related to provenance. The *provenance* of a piece of data is redefined as the process that led to that piece of data. The term *process documentation* refers to the documentation of the process that led to the data item. PASOA distinguishes the two terms in that provenance is determined by performing a query over process documentation[2]. Process documentation is recorded in a dedicated repository, the *provenance store*. The role of a provenance store is to provide a long-term reliable and accessible storage of process documentation.

The lifecycle of process documentation consists of four phases: creating, recording, querying and managing, as shown in Figure 1.1. While executing and producing electronic data, provenance-aware applications create process documentation and store it in one or several provenance stores. After being recorded, process documentation is ready to be retrieved and reasoned over to derive the provenance of some data item. Also, process documentation stored in provenance stores can be managed, maintained or curated by administrators.

[1] An in silico experiment is a procedure that uses computer-based information repositories and computational analysis to test a hypothesis, derive a summary, search for patterns, or demonstrate a known fact [73] as in a wide range of areas, e.g., computational chemistry [66], bioinformatics [73], and drug discovery [93]

[2] To be compatible with current literature, the terms *provenance information* and *process documentation* are used interchangeably in this thesis.

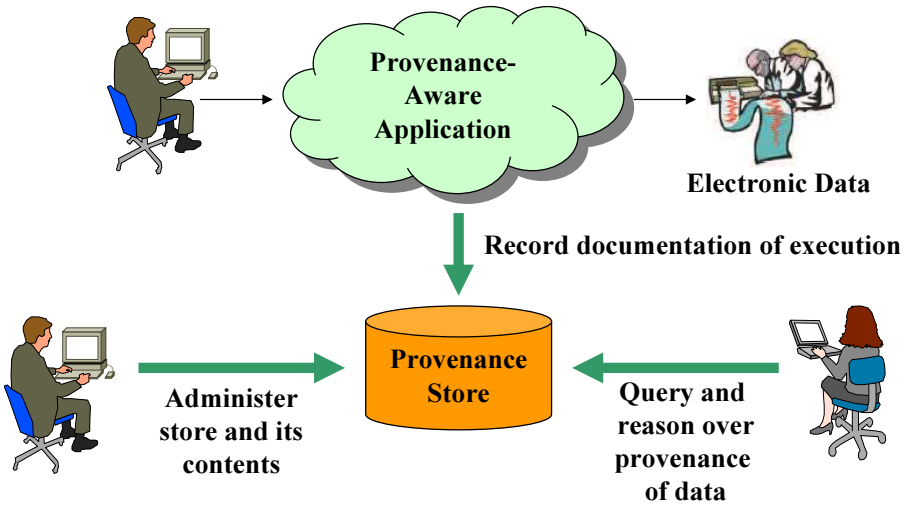


FIGURE 1.1: The lifecycle of process documentation [126]

1.2 Scalable Recording of Process Documentation

Large scale, open distributed systems are typically designed using a service-oriented approach [152], usually referred to as service-oriented architectural style [28]. As opposed to conventional distributed environments, one obvious and common challenge faced by SOAs is managing and sharing heterogeneous services across dynamic distributed organisations with different security policies. A representative example is the Grid, where computing and data resources are geographically dispersed in different administrative domains, and computing resources are highly heterogeneous, ranging from single PCs and workstations, clusters of workstations, to supercomputers [55].

The Grid has been used in a variety of domains including drug discovery [93], bioinformatics [73], earthquake engineering [136], weather forecasting [138], astronomy [154] and high energy physics [74]. Many of these experiments require large-scale resources with thousands of scientists, tens of thousands of computers, and trillions of petabytes of storage across continents in the world [3][4][5]. SF-Express is a distributed interactive simulation application that harnesses multiple supercomputers across nine organisations to meet the computational demands of large-scale network-based simulation environments [24]. In the drug discovery research, 1700 computers were simultaneously used in 15 countries around the world to tackle the scientific challenge. The EGEE project assembles over 250 sites around the world organised in 12 partner regions across 40 countries (Figure 1.2). Several communities in EGEE participated in a project Biomedical Grids to cope with the flood of bioinformatics and healthcare data [67].

When recording process documentation in such distributed and heterogeneous environments, it is impractical for a single provenance store to retain all provenance information

[3] EGEE, http://www.eu-egee.org/
[4] OSG, http://www.opensciencegrid.org/
[5] TeraGrid, http://www.teragrid.org/

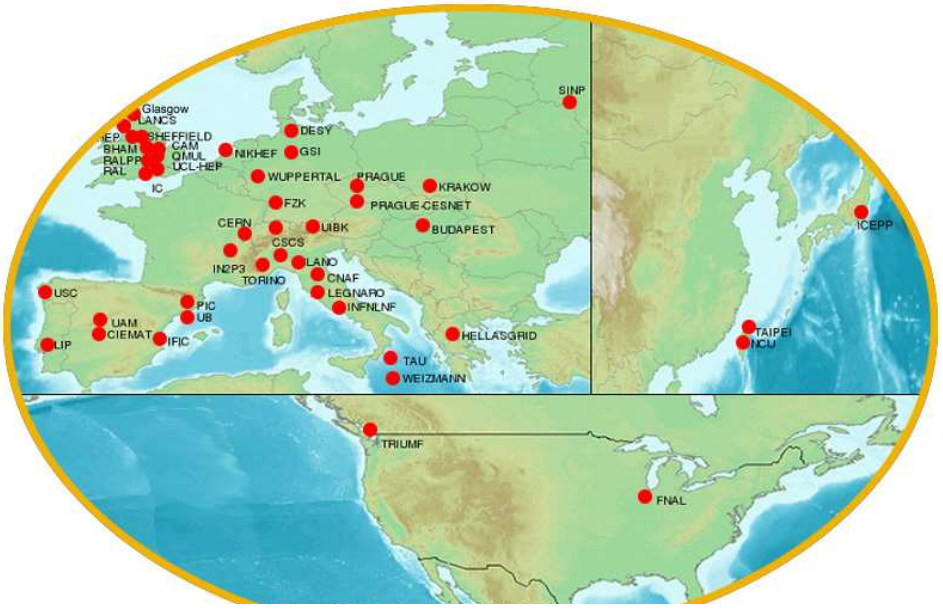


FIGURE 1.2: EGEE Project Participants [67]

of a distributed application. Thus, multiple provenance stores should be used to support scalable recording of process documentation for several advantages. One advantage is that the use of multiple repositories eliminates a central point of failure and performance bottlenecks. With multiple provenance stores, an organisation can record process documentation into a nearby store with short recording latencies.

Using multiple provenance repositories results in process documentation recorded in many locations. Hence there must be some mechanism to connect these stores in order to retrieve distributed process documentation. PASOA introduces a mechanism to interlink provenance stores. Links are recorded along with process documentation in stores, which form a pointer chain connecting all the provenance stores hosting the documentation of a process (Figure 1.3). Using the pointer chain, distributed documentation can be retrieved from one store to another. Section 2.3.4 will detail the linking mechanism.

1.3 Recording Process Documentation in the Presence of Failures

Distributed systems are susceptible to failures [37]. Failures, due to various reasons like server crash, hardware deficiencies, network partition, broken communication, power outage, and other sources of failures (e.g., machine rebooted by the owner, network congestion, excessive CPU load, etc) [90] are a significant cause for concern.

Since large-scale SOA-based applications usually involve heterogenous services provided and controlled by other organisations, another challenge of SOAs is the face of failures that would affect the reliability and availability of services.

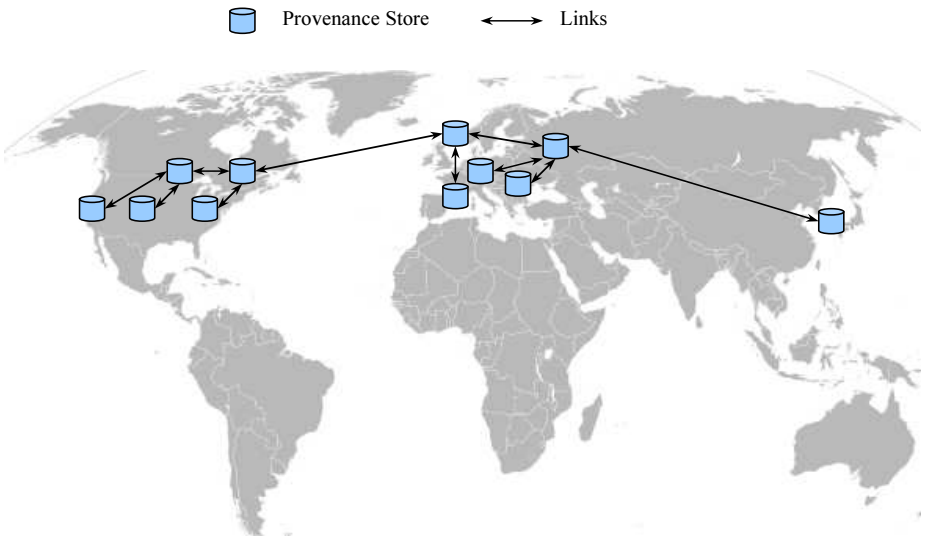


FIGURE 1.3: An Example of Interlinked Provenance Stores

The Grid[6] community has reported many results regarding failures. The weather forecast experiments on TeraGrid saw a significant portion of its workflows (60%) encountering software and/or hardware problems [23]. Grid2003 (now Open Science Grid) observed a 30% job submission failure rate with 90% of the failures caused by problems such as disk filling errors, site overloading and crashes, and network interruptions [57]. The FlexX and Autodock data challenges of the WISDOM project[7], conducted in 2005, showed that only 32% and 57% of the jobs completed successfully, respectively [173].

It is not surprising to find that failures are common in the Grid. This is because the Grid is geographically dispersed, involving a large number of components (e.g., instruments, display, computational and informational resource, and people) across multiple autonomous administrative domains.

When recording provenance in a distributed environment like the Grid, it is unavoidable that the communication with a provenance store (which can be deployed as a Grid service) or the provenance store itself can fail. Some of the current provenance systems, although able to reliably record process documentation [130, 47, 142, 62], do not support multiple provenance stores, whilst others tend to assume a failure-free execution environment or do not discuss this issue [175, 10, 150, 58, 35]. Such a limitation, assumption or omission will hinder the eventual utilisation of provenance. We now take the example of PASOA since it is a general provenance system and also supports interlinked multiple provenance stores.

A scientific application, to be described in Chapter 6, used PASOA to record process documentation in the presence of simulated failures. By analysing the contents of prove-

[6]Cloud Computing receives more and more attention nowadays. Although having its unique characteristics, it is evolved out of Grid Computing and relies on the Grid as its backbone and infrastructure support[59]. We will discuss Cloud Computing and its relationship with provenance in Chapter 7.

[7]WISDOM, http://wisdom.eu-egee.fr/

nance stores after the application completes, we find that the quality of the recorded documentation is poor in the presence of failures[8], as demonstrated in Figures 1.4 and 1.5. In Figure 1.4, as failure rate increases, a large proportion of process documentation fails to be recorded. Figure 1.5 reveals the increase in the number of dangling links, i.e., pointers to other provenance stores that were supposed to record part of process documentation but did not, and in the number of isolated documentation islands. In the example of Figure 1.6, distributed process documentation recorded in multiple stores are disconnected to several islands due to the breakage of the pointer chain.

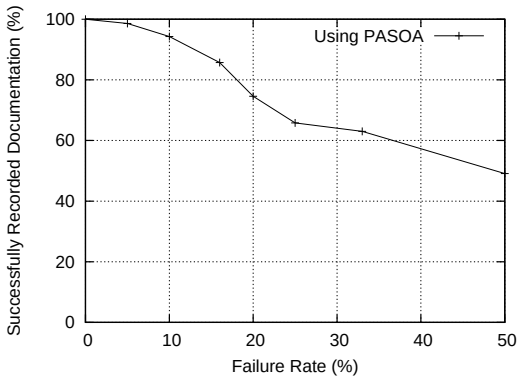


FIGURE 1.4: Loss of documentation records in provenance stores

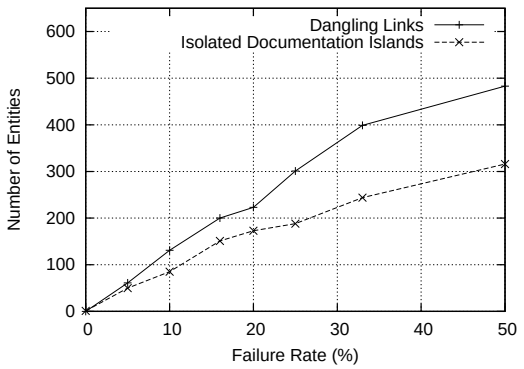


FIGURE 1.5: Dangling links and isolated islands in provenance stores

Process documentation of poor quality, i.e., incomplete and disconnected, is not acceptable in those domains that rely on process documentation to determine the provenance of their data products. For example, U.S. Food and Drug Administration requires drug companies to keep a complete record of drug manufacture and distribution as long as the drug is in use[9]. A number of SOA-based applications have used PASOA to derive provenance of their data products, including healthcare application [44], organ transplant management [11], aerospace engineering [96], RSS feeds [111], trust calculations [140], fault tolerance in the Grid and SOAs [157, 169, 158], biodiversity [167], and auditing of private data use [9]. Due to poor quality, entire process documentation cannot be obtained and hence those applications cannot verify the provenance of their data products.

In addition to recording poor quality process documentation, provenance systems without consideration of failures can also affect the execution of the corresponding provenance-aware application. For example, outstanding documentation which fails to be transported to a crashed provenance store may exhaust the application's memory if no flow control mechanism is implemented, leading to the crash of the application, or it may suspend the execution of the application if the application has flow control mechanisms. Current Grid applications often perform long running tasks that require several or more

[8] We considered the failures of provenance stores and communication links.

[9] Section 600.12, Code of Federal Regulations, Food and Drug Administration. http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?fr=600.12

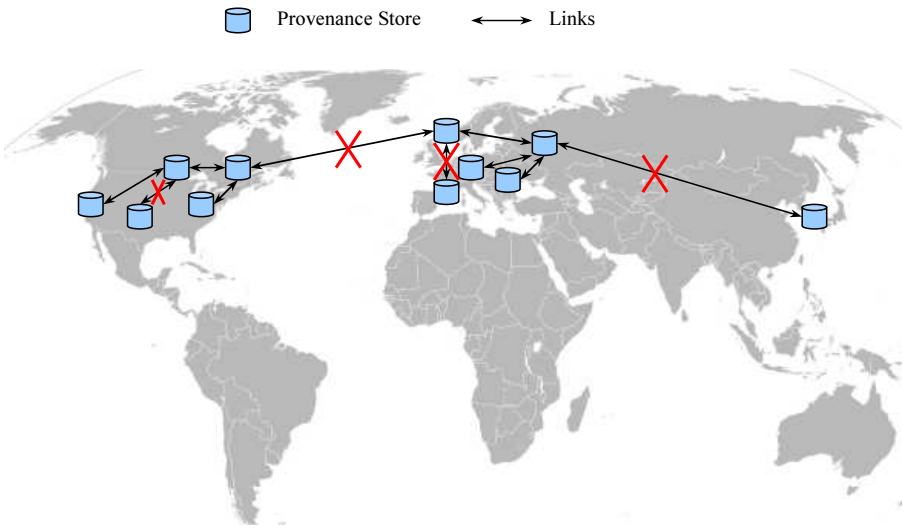


FIGURE 1.6: Broken Pointer Chain

days of computation [114, 55]. Although fault-tolerant mechanisms have been available for Grid applications to tolerate failures, the interruption due to recording process documentation would still cause great troubles to those applications.

1.4 Thesis Statement and Contributions

When recording provenance information in SOAs, the large-scale and heterogenous natures of SOAs require the need for multiple provenance stores and the consideration of failures. Our work aims to address these concerns by designing a generic recording protocol that deals with failures and still ensures the connectivity of distributed process documentation recorded in numerous linked stores.

We state the thesis of our research as follows:

In SOA-based applications, the problem of recording process documentation in the presence of failures (provenance store crashes and communication failures[10]) while still ensuring its entire retrievability is solved via a generic and efficient coordinator-based protocol to guarantee successful recording of complete documentation and to preserve accurate links that connect multiple provenance stores.

To establish this thesis, we firstly present F-PReP, a protocol to record documentation in the presence of failures. Then we formalise the protocol and prove that it guarantees the recording of complete documentation and maintains accurate links. In order to establish the entire retrievability, we analyse the topology of distributed process documentation after being recorded in provenance stores using F-PReP. Finally, we introduce the implementation of F-PReP and conduct evaluations to show that F-PReP is efficient and introduces acceptable recording overhead.

[10] Failure assumptions will be further justified in Chapter 3.

Given that PASOA is a domain and technology independent infrastructure and provides a mechanism to interlink provenance stores, our work extends PASOA with the following original contributions.

- We present and formalise a generic coordinator-based recording protocol F-PReP, which provides remedial actions to cope with failures. The coordinator plays a crucial role in maintaining accurate links to connect multiple stores. Four requirements are identified for the protocol to record complete and retrievable distributed process documentation in the presence of failures. The protocol's correctness is formally proved using mathematical induction.
- We graphically represent the topology of distributed process documentation recorded in interlinked provenance stores. We perform an exhaustive analysis on the forms of graphs, considering all possible topologies after documentation was recorded by using F-PReP in the presence of failures. We also identify a number of graph properties to help us demonstrate the entire retrievability of process documentation.
- A system architecture F-PReServ is described, which employs F-PReP and supports practical issues such as communication, storage and performance. Its features include a novel way of creating process documentation, a new retrieval function, and implementation strategies for achieving good recording performance in the presence of failures.
- An extensive evaluation of F-PReServ is performed, which reveals that it introduces acceptable recording overhead to a provenance-aware application's execution. The evaluation is conducted at several levels. First of all, we measure the throughput of the provenance store and coordinator. We demonstrate that a single coordinator does not result in performance bottleneck. Then, we benchmark the recording performance of F-PReServ and show that remedial actions introduce small overhead (below 10%). In addition, we investigate the performance impact on the execution time of a scientific application. Lessons are learned and recommendations are given on achieving good performance in the case of failures.

1.5 Dissertation Structure

This dissertation is organised as follows.

Chapter 2 provides background information on our work. It surveys the state of the art research for determining provenance in computational systems and reviews fault tolerance mechanisms for distributed systems. From this survey, we position our work and discuss various approaches to reliably recording process documentation.

Chapter 3 states failure assumptions and identifies requirements that F-PReP supports. Then it analyses the problems that may occur in the presence of failures and outlines the protocol's design philosophy. After that, we define protocol messages and detail F-PReP's behaviour. Finally, we formalise F-PReP using an Abstract State Machine approach.

Chapter 4 proves the protocol's correctness by following a systematic procedure based on mathematical induction. A number of properties are established showing that the protocol guarantees successful recording of complete documentation and maintains correct links when failures occur.

Chapter 5 graphically represents contents of interlinked provenance stores hosting distributed process documentation. We establish graph properties and present an exhaustive analysis on graph topologies, which facilitates us to demonstrate the entire retrievability of distributed documentation.

Chapter 6 firstly details the design and implementation of F-PReServ. Then it outlines our evaluation environment and methodology, followed by a series of performance experiments in controlled environments and in a scientific application. Based on lessons learned from the experimental results, several recommendations are given regarding further improvements of the system.

Chapter 7 discusses future work and concludes this dissertation.

1.6 Publications

Zheng Chen and Luc Moreau. Recording Accurate Process Documentation in the Presence of Failures. In *Proceedings of Workshop on Methods, Models and Tools for Fault Tolerance*, pages 128-138, 2007.

Zheng Chen and Luc Moreau. 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'08), volume 5272 of Lecture Notes in Computer Science, pages 92-105. Springer, 2008.

Zheng Chen and Luc Moreau. Recording Process Documentation in the Presence of Failures. *Methods, Models and Tools for Fault Tolerance*, volume 5454 of Lecture Notes in Computer Science, pages 196-219. Springer, 2009.

Chapter 2

Related Work

In this chapter, we review research on provenance and provide background information on failures and fault tolerance mechanisms. We also position our work and discuss various alternative approaches to addressing our problem.

In the first part of this chapter, we introduce Service-Oriented Architectures and workflows, and then review the state of the art research about provenance. The provenance community has been growing in recent years and many systems have been seen to support provenance in many research areas. Through a comprehensive comparison of major provenance-related systems, PASOA has several advantages over the other systems, which have been demonstrated in a wide range of applications.

Then we highlight key aspects of PASOA, such as the modelling and recording of process documentation, and the linking mechanism that is used to connect multiple provenance stores.

In the second part, we introduce failure models in distributed systems and survey major fault-tolerant techniques. We also highlight the importance of formal methods to the design of distributed protocol and review formal approaches to modelling fault-tolerant applications.

2.1 Background

2.1.1 Service-Oriented Architectures

Large scale, open distributed systems are typically designed using a service-oriented approach [152], usually referred to as service-oriented architectural style [28]. A Service-Oriented Architecture (SOA) consists of loosely coupled services communicating via a common transport. Typically, a service is only available through an interface represented

in some standard format. The main advantage of SOAs is that they hide implementation behind an interface allowing implementation details to change without affecting the user of the service.

One of the main ways to implement SOAs is to build cross-platform, interoperable applications out of Web Services [7, 134]. Using Web service technologies, a service's interface can be expressed in the Web Services Definition Language (WSDL) and it can communicate using the common transport protocol SOAP. Because of these properties, SOAs are particularly good for building large scale distributed systems, such as the Grid.

The Grid is a complex computing infrastructure designed to support the sharing of heterogeneous computational and data resource across dynamic and geographically distributed organisations [55]. In this context, a fundamental problem is how to provide an interoperable access model to all types of resources. This issue has been addressed by the Open Grid Services Architecture (OGSA) [55], where a SOA style of resource access is adopted. Due to the benefits of SOAs and Web Services, Grid middleware, e.g., Globus Toolkit 4 [56], gLite [67] and UNICORE [116], has moved towards these technologies. Hence, Grid resources have been modelled as Web Services to facilitate resource sharing in heterogeneous environments.

The same trend has been seen in scientific and engineering communities. When integrated with the Grid, domain-specific services have also been modelled as Web Services in a wide range of applications, such as scientific data simulation [87], astronomy [154], biology [95], and environmental science [50, 138].

In order to tie services together, one technique that is often used is workflow.

2.1.2 Workflows

A workflow specifies the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how information flows to support the tasks and how tasks are being tracked [52].

Workflows can be divided into two types, abstract and concrete. Abstract workflows are those in which the task dependencies are defined but are not bound directly to a particular service. In contrast, concrete workflows are those where the tasks are bound to services. Software such as Pegasus [43] takes abstract workflows and generates concrete workflows taking advantage of available resources. Users can also create concrete workflows directly, either by hand or by workflow editing software such as Taverna [89]. Workflows are typically executed using a workflow enactment engine like Taverna and Condor [65], which invokes various services on the user's local machine, or remotely such as the Grid, taking advantage of heterogeneous resources [53].

As workflows become more generic and reusable, they are beginning to be exposed as services themselves [172]. Essentially, in the context of SOAs, workflows that centrally coordinate services are moving towards multi-level nested workflows, which means services of a workflow may invoke other services [40].

Due to increasingly complex sets of computations and data analyses, workflows have emerged as a paradigm for representing and managing complex *in-silico* experiments [98]. In order to enable workflow re-execution and the reproducibility of results in these experiments, scientific workflow communities have identified that provenance information must be generated and captured during the course of workflow execution [72].

One of the challenges that workflow communities face is scalability [72]. Large scale workflows may involve thousands of steps and perhaps millions of tasks, where each step may integrate diverse models and data sources defined and developed by different organisations. The applications and data may be also distributed in the execution environment and many participants may define the workflow, managing its execution, and interpreting results. To record provenance information in large scale workflows, the need to connect multiple provenance repositories is obvious.

2.2 Provenance Research

In this section, we briefly overview provenance research in different domains. Through this review, we demonstrate that PASOA is a general approach to modelling and recording process documentation. We also notice that none of the current systems, which use multiple provenance repositories, has adequately addressed the failures that could occur during the recording of provenance information.

2.2.1 Overview

This section surveys literature on provenance related research. Prior research has used the term *lineage* [105] to refer to provenance. We use the two terms interchangeably in this section. The aim of this section is to show that provenance has attracted attention in many areas.

There are two important surveys regarding provenance research. One [19] is about lineage retrieval systems, workflow systems, and collaborative environments. Another one [149] surveys several systems based on taxonomy of provenance techniques.

Lanter conducted pioneering research in provenance in the early 1990's. He studied the lineage problem in geographic information systems (GIS) [19]. Provenance can help indicate the quality of derived map products in GIS applications. This is useful for GIS users to determine the fitness of the use of map in their application.

The database community focuses on data lineage problem since the late 1990's. This problem can be summarised as given a data product, determine the source data used to produce that item. A data product in a relational database can be a view, a table, a tuple, an attribute. It can also be a pointer to an external data resource such as a file.

There are three approaches to data lineage problems in database community. The first one uses annotations on attributes in databases [34]. Lineage annotations encode information about the data source and the query that created them. Databases can also use query inversion and function inversion techniques to trace the lineage from a data item back to its source [27, 166, 164]. These two approaches typically focus on situations in which all of the interactions with data take place in a single database, whilst the third approach proposed by P.Buneman recently is to track and manage the provenance of data that moves among multiple databases [26]. He proposes a copy-paste model describing user actions in assimilating external data sources into curated database records.

Low-level provenance recording has been studied at the levels of program execution and operating systems. Provenance-Aware Storage System (PASS) is proposed to automatically collect provenance at the operating system level [129, 153]. It observes all processes that run on a PASS-enabled operating system, and generates provenance data about low-level details like the loaded kernel modules, installed libraries and process environment. Provenance information is then maintained in a file store for later query by users. The Earth System Science Server (ES3) project is developing a local infrastructure for managing Earth science data products derived from satellite remote sensing [64]. Similarly to PASS, ES3 extracts provenance information automatically from arbitrary applications by monitoring their interactions (arguments, file I/O, system calls, etc) with their execution environment. Provenance information is then logged to the ES3 database.

Provenance is a relatively new research area in SOA-based applications, which are usually represented in the form of workflows. The provenance of a data result of an experiment is determined by provenance information (e.g., input and output data to each service) recorded during the execution of the workflow. Many provenance systems have been developed, such as myGrid, CMCS, PAC, gLite, Kepler, VisTrail, Karma, VDS and PASOA. We will detail and compare these systems in Section 2.2.2.

The provenance community has been growing in recent years. Five international workshops related to provenance were held: two workshops on Data Provenance and Annotation (DPAW'02, DPAW'03), two International Provenance and Annotation Workshops (IPAW'06, IPAW'08) [124, 20, 61], and a workshop on the Theory and Practice of Provenance (TaPP'09) [33]. Three provenance challenges[1], a community effort to understand and compare systems addressing provenance, have been organised in 2006, 2007 and 2009, attracting more than 20 institutions to participate. The first challenge estab-

[1] http://twiki.pasoa.ecs.soton.ac.uk/bin/view/Challenge/WebHome

lishes an understanding of the similarities, differences and common issues among available provenance-related systems with the focus on documenting processes and answering provenance-related queries [127]. It also identifies that interoperability between different systems as a key issue, which is the focus of the second challenge. The second challenge deals with integrating provenance information derived from different provenance systems and represented using different models, which all contribute to the provenance of a data product. It concludes that a common data model is required to achieve interoperability among the existing provenance systems. This has led to a proposed specification of a provenance data model, the Open Provenance Model (OPM) [125]. After a review period, a third Provenance Challenge was organised to evaluate this model in June 2009.

2.2.2 Major Provenance Systems

In this section, we review and compare several systems supporting provenance collection for SOA applications. Nine representative systems are chosen based on five provenance related international workshops (DPAW'02, DPAW'03, IPAW'06, IPAW'08 and TaPP'09). Such a survey serves two aims: first, the reason why our work is built on PASOA; second, how these systems deal with failures when recording provenance information.

Table 2.1 compares the nine systems against several criteria, which are essential requirements that a provenance system should support in a wide range of application domains [117, 76]. We first introduce each system in Section 2.2.2.1 and then discuss this table in Section 2.2.2.2.

	myGrid	CMCS	PAC	gLite	Kepler	VisTrail	Karma	VDS	PASOA
Domain Independent?	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Technology Independent?	No	No	No	No	No	No	No	No	Yes
Type of Provenance	Data	Data	Job	Job	Data	Data, Workflow	Data	Data	Open
Multi-Site Recording?	No	No	No	No	No	No	Yes	Yes	Yes
Multiple P-Stores?	No	No	No	No	No	No	No	Yes	Yes
Reliable Recording?	No	Yes	Yes	Yes	No	No	Yes	No	No

TABLE 2.1: Comparison of Provenance-related Systems

2.2.2.1 Overview of Candidate Systems

The myGrid project provides web service-based middleware in support of *in-silico* experiments in bioinformatics [175]. The Taverna workflow enactment engine has been modified to generate provenance information based on semantic web technologies, such as Resource Description Framework (RDF) [115], and Life Science Identifiers (LSIDs) [174]. Provenance information documents the invoked services, their inputs/outputs, performed functions and any other metadata that might be of interest to scientists. Provenance information is then gathered by Taverna during workflow execution and stored in a centralised MySQL database.

The CMCS project provides informatics tools to support multi-scale data management and provenance tracking across web service-based chemical experiments [130, 131]. CMCS uses a centralised Scientific Annotation Middleware (SAM) [21] repository to store provenance information, which is URL referenceable files representing all resources such as data objects, processes and invoked web services of a workflow. Provenance information is populated by applications in workflows or manually entered by scientists via a portal interface. Open Source Java Messaging Service, OpenJMS, is used to reliably deliver messages, including provenance information.

Provenance-Aware Condor (PAC) transparently gathers provenance information while a job runs on Condor [141, 142]. The provenance information, modelled by PAC schema includes a job's execution environment and all files related to the job's execution. Provenance information can help identify if a job was run on machines with a faulty processor and to determine if a job is affected by a hardware problem. The components in PAC are Condor, Quill and FileTrace. Quill gathers job execution runtime information from Condor daemons whilst FileTrace collects information about files used by Condor jobs. Provenance information is recorded into a central PostgreSQL database, which is evaluated to have little performance impact on Condor. Quill provides fault tolerance for reliably recording provenance information.

Similarly to PAC, gLite, the Grid middleware of EGEE project, automatically collects and keeps tracking the provenance of a job [47, 97, 61]. Provenance information includes two aspects: job input (e.g., job description, a job's input files and parameters) to enable job re-running and the job's runtime environment (e.g., versions of used software and environment settings). A Logging and Bookkeeping service securely and reliably collects job provenance as the job is running on gLite. It submits the provenance information into a Job Provenance (JP) service, which stores provenance data in a backend database. Since only a few JP services are installed in the whole EGEE gLite middleware, each JP service is required to be scalable enough in order to deal with provenance data generated from millions of jobs. Although gLite uses several JP repositories, the provenance of one job is only maintained in one repository.

Kepler is a scientific workflow system which supports diverse types of workflows from those designed for job control and data movement in Grids to those for high-level conceptual scientific experiments [10, 22, 110]. To achieve this, Kepler provides a configurable provenance collection framework to track all aspects of provenance in scientific workflows, including runtime context, input/output data, intermediate data products, workflow definition, and information about the workflow's evolution. Provenance data is collected in XML format by the workflow enactment engine and archived in a provenance repository. Kepler also provides a unique "smart"rerun functionality that extracts provenance information to rerun partial workflow in the case of a workflow parameter change [10].

VisTrails is a visualisation and scientific workflow system [62]. VisTrail not only collects the provenance for data products, but also becomes the first system to capture the provenance of a workflow, i.e., the evolution of a workflow, by recording all versions of a workflow and all its instances since the workflow was created by a user. With VisTrail, users can return to previous versions of workflows and workflow runs to compare their results. The provenance information is captured by workflow editor or the enactment engine and recorded in a centralised repository.

Provenance information in the above work is mainly collected by their respective workflow enactment engine. This approach however cannot capture sufficient provenance information in distributed workflow systems where the invoked services use external services or call other workflows. Another shortcoming is the difficulty of verification. If only the enactment engine records documentation about a process, it is impossible for third-parties to verify if the process took place as documented, because they have no other record to compare with. Therefore, by allowing multi-site recording, i.e., both services and enactment engine to record documentation independently, provenance information can also be used for accountability or verification purposes.

To address this problem, Karma supports provenance collection from both the workflow enactment engine and services [150]. Karma is a general provenance framework for scientific workflows. It uses asynchronous Publish/Notification model to record provenance-related workflow activities such as service invocations, input/output data for each invocation. Workflow enactment engine and services publish these activities as notifications to WS-Messenger [88], a Web service-based messaging middleware specifically for Grid applications. A provenance service then listens for those notifications and stores them in a relational database. Reliably delivery of provenance information is ensured by WS-Messenger.

The Virtual Data System (VDS) targets at large scale grid applications (e.g., high-energy physics and astronomy) with the provision of data virtualisation independent of data's location, representation and physical materialisation [58, 176, 35]. VDS uses Virtual Data Language (VDL) to describe the computational procedures used to derive

data, the invocations of those procedures as well as the datasets produced by those invocations. Then VDL statements are compiled to create abstract workflows, which are further materialised to executable workflows that can be run on the Grid. During workflow execution, provenance information about both application runtime behaviour and the runtime environment is collected and recorded in a repository called virtual data catalog (VDC). Similarly to Karma, VDS also supports provenance collection from multi-sites. In addition, VDS supports multiple VDCs, connected by virtual data hyperlinks, to support large-scale grid applications [58]. One limitation of VDS is that it does not store explicit relationships between input and output data of services within a workflow, so determining provenance of data relies on the presence of the same workflow definition as was executed at the time.

The Provenance Aware Service Oriented Architecture (PASOA) project has built a generic, domain-independent and technology-independent infrastructure to provide interoperable means for recording and using provenance [77, 126]. PASOA uses an SOA-based approach, which models a process as a set of causally related interactions between services via message passing [80]. An open provenance recording protocol PReP [82] is developed, which, to our knowledge, is the first formalised recording protocol to specify the behaviour of recording actors and provenance stores. Due to the open nature, users can capture data provenance at any granularity level. All services involved in a process for generating a result contribute to the process documentation for determining the provenance of that result. A linking mechanism is developed to connect multiple provenance stores to facilitate provenance collection in large scale environments [76]. A concrete implementation of PReP is provided, named Process documentation Recording for Services (PReServ) [79]. PReServ contains a Provenance Store Web Service, a set of interfaces for recording and querying provenane store, a set of Java client libraries for easily accessing those interfaces, and an Axis handler for automatically recording process documentation for Axis based web services. A Python version of the client side library is also available [18]. A wide range of applications have used PASOA to record their process documentation, which we will detail in Section 2.2.3.

2.2.2.2 Discussion

We now discuss these systems according to Table 2.1.

Domain Independent Early provenance-related research was related to specific domains, such as Biology (myGrid) and Chemistry (CMCS). The shortcoming of being domain dependant is that it hinders the reusability of provenance components. Application developers have to re-implement components for recording or using provenance in different execution contexts. Recent works have focused on the development of general systems that can collect provenance information from a wide range of applications.

Technology Independent All the above works except PASOA are technology dependent since they are tightly coupled to the workflow technologies. myGrid, CMCS, Kepler and VisTrail all have their own workflow systems with built-in functionalities or third party software to support provenance collection. PAC and gLite are strictly restricted to their respective execution engines (Condor and gLite, respectively). Although Karma is designed to be a general provenance framework independent of any specific workflow systems, it still relies on the notion of a workflow, because it is designed to capture workflow activities. VDS is not confined to any particular workflow system either. But it does not store explicit relationships between input and output data of services within a workflow, so determining provenance of data relies on the presence of the same workflow definition as was executed at the time.

There are two drawbacks of being technology dependent. Firstly, distributed workflow execution involves execution under different workflow engines and heterogeneous provenance collection mechanisms. Therefore, it becomes difficult to integrate provenance derived from those different workflow systems and represented using different models. In addition, many applications do not need or wish to commit to a workflow environment, which we will illustrate in Section 2.2.3. Provenance systems replying on workflow technologies, however, cannot cope with these applications.

PASOA's domain and technology independent nature distinguishes it from all the other provenance-related systems. It is a generic and standalone provenance architecture independent of the notion of workflow.

Type of Provenance The surveyed systems all focus on different types of provenance. Data provenance gives the history of deriving a data product; job provenance is concerned with workflow runtime information that enables job rerun; workflow provenance reveals the evolution of a workflow. PASOA allows users to capture data provenance and a component's state information [167], which can be used for job provenance. Although PASOA has not demonstrated that it can record workflow evolution, it has successfully tracks the provenance about workflow refinement [118], which is similar to workflow evolution as both reflect how a workflow is changed before execution.

Multi-Site Recording Provenance information in most reviewed systems is only provided by a workflow enactment engine. As mentioned before, this approach has disadvantages such as the incapability of giving sufficient information to determine the provenance of a piece of object when a workflow is highly decentralised. Karma, VDS and PASOA support gathering provenance from both workflow enactment engine and services.

Multiple Provenance Stores Given that large-scale SOA applications may involve services owned by many institutions, it is impractical to expect a single provenance store to be used to retain all of the process documentation due to issues such as single point of failures, scalability, security and access control. PASOA and VDS provide similar Web-like linking mechanisms to connect multiple provenance repositories whilst other systems

assume a centralised data store to maintain provenance information.

PASOA records links along with process documentation in provenance stores. A link is a pointer to a remote provenance store. The set of links forms a pointer chain 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. VDS [58] connects multiple provenance repositories using hyperlinks, which form a provenance chain spanning across a range of servers to enable scalable recording. However, VDS is technology-dependent and does not have a well-defined API for querying its provenance repositories.

Reliable Recording PAC and gLite provide fault-tolerant support for provenance collection mainly through persisting messages in local site and retransmitting them in the event of failures. In PAC, the Quill service writes provenance information to log files and then periodically reads the logs and inserts the data into a relational database shared by all machines in a Condor pool. In gLite, the Logging and Bookkeeping service (LB) provides a notification-based messaging infrastructure to securely and reliably collect job information (in the form of events) from individual Grid components. In order to provide reliability of transporting event messages, LB also logs events on a component's local disk before delivering them to a LB server.

Both CMCS and Karma employ third party message queue middleware (OpenMQ and WS-Messenger, respectively) to reliably record their provenance data. Messages are placed onto a queue and stored in the queue until the recipient retrieves them. Fault-tolerant functionalities are provided by the message queueing systems to ensure that messages do not get lost and can be successfully delivered in the event of a system failure. We will detail message queue systems in Section 2.6.3.

The other provenance systems do not consider failures; hence they cannot guarantee the successful recording of provenance information when failures occur.

Summary From the above discussion, we can see that PASOA has a generic nature of recording provenance information in large scale heterogeneous environments. Its domain and technology independent model allows users to capture any type of provenance at any granularity for any application no matter if it is workflow centric. In addition, it supports provenance collection from multi-sites and provides a mechanism to connect multiple repositories to enable scalable recording. The main deficiency of PASOA is that it cannot guarantee the reliable recording of provenance data in the presence of failures which are common in large scale heterogeneous environments like the Grid. In the next section, we will describe a wide range of applications that have used PASOA to record process documentation to further demonstrate the advantages of PASOA as a generic provenance architecture.

2.2.3 Applications that Used PASOA

A number of applications have used PASOA to derive provenance of their data products, including healthcare application [44, 11], fault tolerance in the Grid and SOAs [157, 169, 158], auditing of private data use [9], aerospace engineering [96], biodiversity [167], trust calculations [140] and workflow refinement [118]. In addition to the Java version, the client side library developed in PASOA has also been implemented in Python, enabling a wide range of Python applications, applications with Python interface and Python-glued workflows to be provenance-aware [18].

We now briefly introduce these applications. The purposes of introducing these applications are twofold: firstly, they provide a range of evidences demonstrating the advantages of PASOA, such as openness, technology independence, and support of multi-site recording and multiple provenance repositories; secondly, the lack of reliable recording of provenance information has profound impact on a wide variety of applications that have used PASOA and hence deserves further investigation.

Healthcare application An SOA-based healthcare application used PASOA to record history information of a patient's treatments in order to facilitate auditors to verify if a particular process was executed as expected, and to facilitate doctors to adopt proper subsequent treatments of the patient [44, 11]. In a typical healthcare scenario, healthcare data is often distributed among several heterogeneous and autonomous information systems (actors) under different healthcare authorities, e.g., general practitioners, hospital departments. This means each actor operates independently and defines its processes and data representation without the assumption on a pre-described workflow. The open, technology-independent nature of PASOA enables provenance information to be gathered and recorded in such a scenario. Multiple provenance repositories are also suitable to be used by those heterogeneous and autonomous actors in this application.

Fault Tolerance in the Grid and SOAs A provenance-aware fault-tolerance framework, FT-Grid, is developed to tolerate software faults that occur in service-oriented applications through multi-version design (MVD) [157, 169, 158]. In order to provide users with a correct result, FT-Grid invokes multiple functionally equivalent services and performs voting on their results in order to mask software faults (i.e., incorrect results) from services. However, MVD may still give incorrect results due to the possible presence of common-mode failures, which means multi-version services may share common faulty services, thus leading to similar errors between versions of an MVD system. In order to detect common mode failures, FT-Grid employs PASOA to capture provenance information, which reflects the causal relationship between service interactions and hence provides topological awareness of service dependency. Therefore, with recorded provenance information, FT-Grid can improve the voting algorithm and return correct results to users without being affected by common mode failures. Although it is unclear if multiple provenance stores are used in FT-Grid, we can foresee that it is necessary to employ

multiple stores if FT-Grid is used by large-scale distributed applications.

Auditing of Private Data Use A provenance-based auditing architecture based on PASOA has been developed for auditing the processing of private data in organisations' IT systems as required by many regulatory frameworks such as UK Data Protection Act (DPA) [9]. DPA provides protection for an individual's private information placing restrictions on how organisations can acquire, store, share or dispose personal information that they hold. Provenance information tells the regulators the history of data use, which can be verified if the use of private data within organisations is compliant with specific regulations. In order to provide complete and sufficient provenance information, multiple parties are required to record such information into several provenance repositories due to privacy concerns. Provenance information captured using this architecture has been evaluated to be able to answer a range of queries from DPA regulators.

Aerospace Engineering An aerospace engineering application uses PASOA to record documentation of complex simulation processes during the manufacturing of flights. Process documentation is used for compliance and liability reasons and to facilitate process analysis [96]. A workflow regarding the engineering process is specified consisting of several computational components in the simulations. Due to the distributed simulation environments of the application, several provenance stores are required to record documentation produced at different simulation sites.

BioDiversity A biodiversity experiment is conducted to demonstrate how provenance information can help identify execution bottleneck, result accuracy and service throughput in the process of making predictions of the anticipated effects of climate change upon biodiversity [167]. An experimental resource is wrapped as a Web service tailored to that particular resource's inputs and exposed by a standard set of methods. These web services are then invoked by a workflow enactment engine. A monitoring daemon is used to monitor the execution of the workflow, running at the site of enactment engine and each service. Information about service workload, CPU/disk/memory usage, service response latency and throughput is also documented as provenance information using PASOA and recorded in multiple provenance stores. Provenance information is then queried by scientists in order to evaluate experimental results.

Trust Calculations Provenance information has been used for evaluating trust in the outcome of workflow execution [140]. A rule-based analysis tool is introduced to perform subsequent analysis on the process documentation aiming to automatically calculate trust measures for a workflow's result. Using this approach, a workflow enactment engine is able to automatically choose the most trustworthy service.

Workflow Refinement Workflow compiler Pegasus has been integrated with PASOA to capture provenance information regarding workflow refinements (i.e., transformations) so that a user can understand how an abstract workflow description defined by the user is transformed into an executable workflow by the workflow compiler [118]. PASOA

models the refinement process as interactions between several services, i.e., Pegasus and five refiners. All these services record documentation about their interactions as well as the relationships between these interactions (i.e., the relationship between an output and an input of a service). Hence, a causality graph can be produced reflecting the whole refinement process leading to an executable workflow to be executed on the Grid. This example also demonstrates the open and technology independent nature of PASOA since the refinement process is not workflow itself but local method calls between Pegasus and several refining functions.

Python Applications In addition to the Java version, the client side library developed by PASOA has also been implemented using Python, enabling a wide range of Python applications, applications with Python interface and Python-glued workflows to be provenance-aware [18]. Python is a general-purpose high-level programming language, whose greatest strength is a large standard library providing a wide variety of tools to interact with programs written in lower-level languages such as C and C++. Therefore, Python is a powerful glue language between different languages and tools [46]. For this reason, more and more scientific applications in the fields of mathematics, physics, or engineering are developed in Python and those written in C, C++ or Fortran usually provide Python interfaces for convenient integration in working environments. In addition, many computationally intensive parts of application codes are still written in C and C++ whereas Python is used as a means to configure these codes, to setup the overall computing workflow, or to manage the involved data [92]. With the Python version of client side library, all the above applications can benefit from the recording of process documentation to determine the provenance of their data products.

2.2.4 Summary

This section reviews the state of the art research about provenance. The provenance community has been growing in recent years and many efforts have been seen to support provenance in a wide range of applications. Through a comprehensive comparison of major provenance-related systems, PASOA has several advantages over the other systems, which have been demonstrated in a wide variety of applications.

The advantages of PASOA are summerised as follows. Firstly, it is domain and technology independent. It uses an SOA approach to model provenance information, which can support different domain applications. It also specifies a generic recording protocol, which can be implemented in different languages such as Java and Python. Secondly, it is an open architecture. It allows application developers to customise the granularity level provenance information is collected at. Thirdly, it supports multi-site recording, i.e., all participating parties in a process, such as workflow enactment engine and services, contribute to the provenance information of the data product of that process. This is crucial for obtaining sufficient provenance data in highly decentralised applications.

Fourthly, it introduces a linking mechanism to connect multiple provenance stores, which have been demonstrated in many applications to be necessary.

The major disadvantage of PASOA is the lack of mechanism for reliably recording provenance information in the presence of failures. Given that PASOA has been widely adopted in various applications, such a disadvantage would have profound negative impact on those applications and hinder the use of PASOA in future applications.

Based on the analysis in this section, we have decided to extend PASOA with functionalities supporting reliable recording of process documentation. In this way, we can remove the disadvantage whilst preserving the advantages of PASOA. We now detail PASOA's key aspects.

2.3 PASOA

In this section, we introduce key aspects of PASOA: the SOA-based approach to representing provenance [76, 77, 80], the generic recording protocol PReP [82] and the linking mechanism that connects multiple provenance stores [76].

2.3.1 Process Documentation

In service-oriented architectures, clients typically invoke services, which may themselves act as clients for other services. The term *actor* is used to denote either a client or a service. In SOAs, messages are the only mechanism used to transfer information between actors. We note that the SOA approach adapted by PASOA is not limited to Web services. In order to be generic, the following are all considered as "services" because they all exchange messages (i.e., inputs and outputs) in one way or another: local functions/methods, Web services, Corba or RMI objects, and command line programs.

An actor that sends an application message is referred to as a *sender*, whereas an actor that receives an application message is known as a *receiver*. One message exchanged between a sender and a receiver is an *interaction*. PASOA defines a *process* as a causally connected set of interactions between actors involved in that process. By documenting all the interactions that have taken place between actors involved in the computation of some data, one can replay an execution, analyse it, verify its validity or compare it with another execution. Describing such interactions is thus core to producing process documentation.

An actor documents an interaction by making *p-assertions* to provide a sender or receiver's view of the interaction and how those interactions are related. Process documentation therefore consists of a set of p-assertions. There are three types of p-assertions.

- An *interaction p-assertion* documents the content of a message and is created by the actor that has sent or received that message. Both the sender and receiver of an interaction make an interaction p-assertion in order to support multi-site recording.

- A *relationship p-assertion* is made by an actor to describe how the actor produced output data (whether the returned result or invocation message to other actors) from input data that it received by applying some function on the input data.

  The output data is the *effect* and the input data is the *cause*. An effect can have multiple causes. We use terms *effect interaction* and *cause interaction* to denote the interactions where the effect and cause are transferred, respectively. Therefore, a relationship p-assertion captures causal connections between an effect interaction and cause interaction(s).

- An *actor state p-assertion* is made by an actor about its internal state in the context of a specific interaction, which may include the function the actor performs, the workflow that is being executed, the amount of disk and CPU used in a computation, and application-specific state descriptions, etc.

An interaction key is generated by the sender of an interaction for uniquely identifying the interaction from all other interactions. The receiver then uses the interaction key to generate and record p-assertions about the same interaction.

We now illustrate these p-assertions using Figure 2.1, which shows a simple process consisting of two interactions, represented by interaction keys I1 and I2. Actor A1 sends to actor A2 a message M1 containing data d1. After receiving M1, A2 performs a function f on d1 and produces a result d2. A2 then returns the result in message M2 to A1. We assume A2 needs to record the version number of function f.

In this figure, d2 and d1 are the effect and the cause, respectively. Correspondingly, interactions I2 and I1, where d2 and d1 are exchanged to/from other assertors, are effect interaction and cause interaction, respectively. A relationship p-assertion can be created to capture the causal connection (f) between I2 and I1.

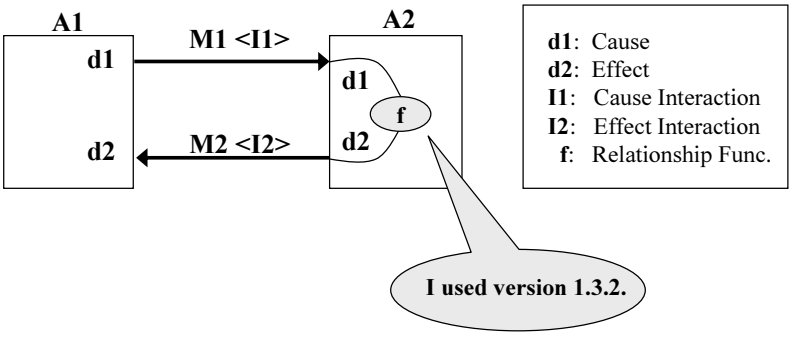


FIGURE 2.1: A simple process

Table 2.2 summarises several p-assertions that are made in this process. In the first interaction where A1 sends M1 to A2, A1 creates an interaction key I1 and then makes an interaction p-assertion documenting M1. We use S to denote A1's view kind in the interaction, i.e., the sender. Interaction key I1 is exchanged to A2 in M1, so A2 can document the same interaction by making an interaction p-assertion about the receipt of M1, where its view kind is the receiver (denoted by R). After performing function f(d1) and obtaining result d2, A2 creates interaction key I2 and embeds d2 and I2 in a response message M2. In the second interaction, A2 makes an interaction p-assertion about the returning of M2 and a relationship p-assertion describing the causal relationship between d2 and d1: d2 is produced from d1 using f. In addition, A2 makes an actor state p-assertion documenting the version number of function f. On the other side, when receiving M2, A1 documents M2 in an interaction p-assertion using the same interaction key I2. The six p-assertions contribute to the process documentation that can be used to determine the provenance of d2.

Actor	Interaction Key	View Kind	P-assertion Type	P-assertion Content
A1	I1	S	interaction	M1
A2	I1	R	interaction	M1
A2	I2	S	interaction	M2
A2	I2	S	relationship	d2=f(d1)
A2	I2	S	actor state	version 1.3.2
A1	I2	R	interaction	M2

TABLE 2.2: P-assertions generated in the example process of Figure 2.1

2.3.2 Representing Provenance

The provenance of a particular data item can be represented as an annotated causality graph, which consists of several elements (Figure 2.2). The edges of the graph represent causal relationships between data items. These relationships denote functions or operations applied to data. The nodes of the graph are data items, which are the effects or causes indicated by a causal relationship. Data items are also annotated by an interaction key and actor states. The interaction key indicates the interaction where the corresponding data is exchanged between actors whilst the actor states describe an actor's knowledge about the receipt and sending of the data. The nodes and edges are extracted from relationship p-assertions and the annotation information is obtained from the interaction p-assertion and actor state p-assertions.

A causality graph is a Directed Acyclic Graph (DAG) that indicates where and how the data was used and derived [76]. Such a DAG starts with the data item followed by the relationships in scope that represent the process that lead to such data item. Therefore, in order to answer provenance questions, users can traverse the causality graph to identify how a data item was produced. As the example of Figure 2.1, the provenance of d2 is represented as a graph shown in Figure 2.3. The graph can be used

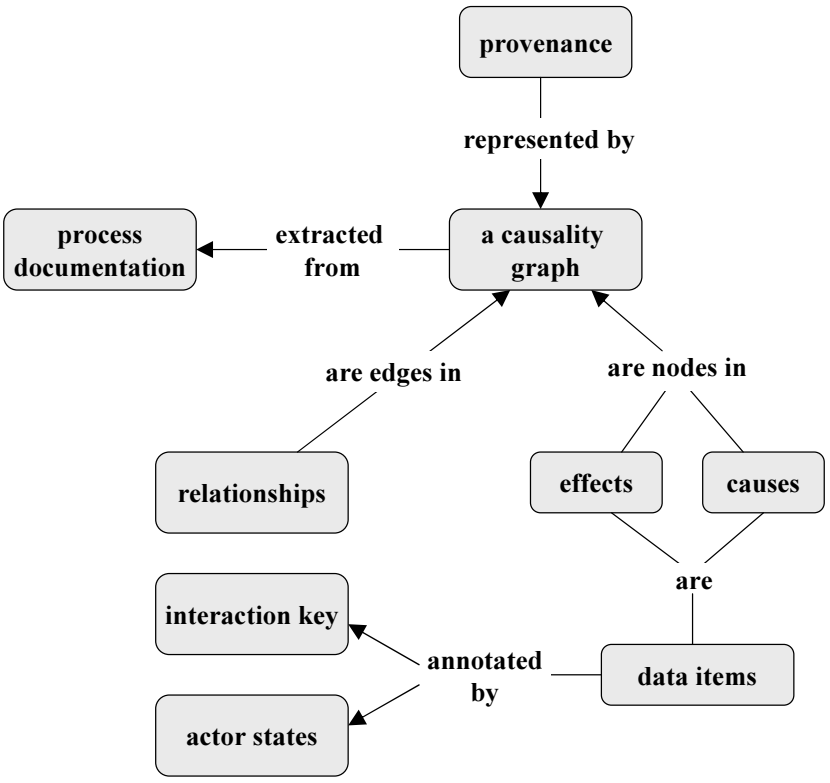


FIGURE 2.2: Concept map representing provenance (revised from [76])

to answer questions such as which algorithm was used to generate d2, what the version number of the algorithm is, and which input data was used to derive d2.

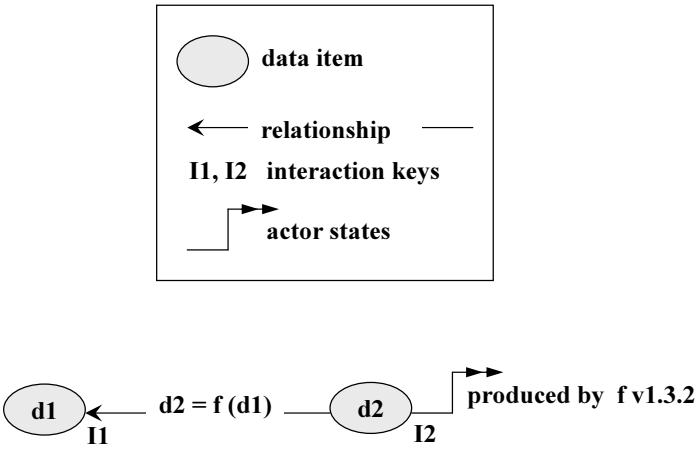


FIGURE 2.3: An example causality graph

Figure 2.4 gives another example process, where five institutions are modelled as actors, participating in four interactions. For simplification, we do not show the messages exchanged between actors. Data d3 is the output of function f1, which takes two inputs d1 and d2. Therefore, d3 is the effect of two causes d1 and d2. Another function f2 is used by A4 to generate d4 from d3. Figure 2.5 shows this process's causality graph, where each institution's name is recorded as actor state information. This graph reflects how d4 is derived from d1 and d2.

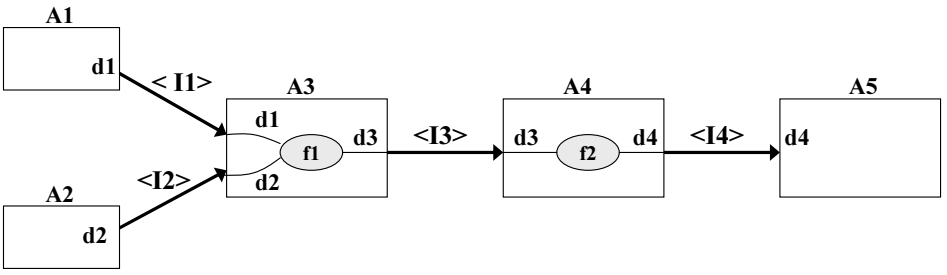
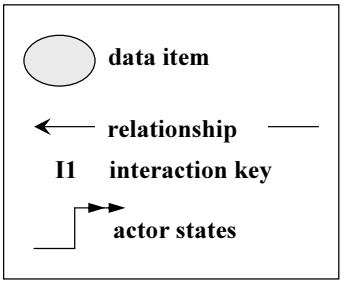


FIGURE 2.4: Another example process



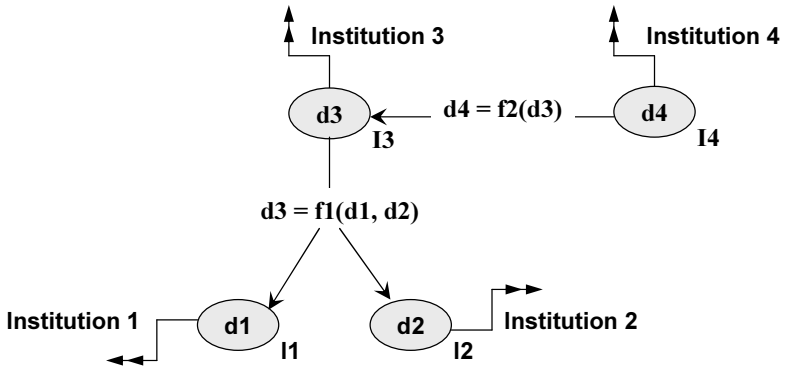


FIGURE 2.5: Generated causality graph

2.3.3 Recording Process Documentation

Actors record process documentation to a dedicated repository, the provenance store, which is built to persistently store large amounts of process documentation and to deal appropriately with problems such as security and access control.

PASOA provides an implementation independent P-assertion Recording Protocol, PReP, to specify the communication between actors and the expected behaviour of those actors when recording p-assertions to a provenance store. PReP enforces that p-assertions are created and organised as appropriate process documentation and maintained in provenance stores ready to be retrieved.

In order to efficiently locate and extract p-assertions from process documentation, PReP specifies that all p-assertions created by an actor about the same interaction are submitted to a provenance store in a single message, which is termed an *Interaction Record* (IR). Table 2.3 gives the four interaction records documenting the example process in

Figure 2.1. For simplification, we use ipa, rpa and apa to denote interaction p-assertion, relationship p-assertion and actor state p-assertion, respectively. In this table, A1 makes two interaction records IR1 and IR4, whilst A2 creates IR2 and IR3. We note that IR2 and IR3 document the cause interaction and effect interaction of the relationship p-assertion, respectively. The relationship p-assertion has been specified by PReP to be included in the interaction record about the effect interaction since it describes the causal reason for the occurrence of the effect interaction.

IR	Actor	Interaction Key	View Kind	IR Elements	Element Content
IR1	A1	I1	S	ipa	M1
IR2	A2	I1	R	ipa	M1
IR3	A2	I2	S	ipa rpa apa	M2 d2=f(d1) version 1.3.2
IR4	A1	I2	R	ipa	M2

TABLE 2.3: Four interaction records documenting example process in Figure 2.1

In order to derive a causality graph to determine provenance, both actors in an interaction must document their view of the interaction by making an interaction record, which includes a *compulsory* interaction p-assertion reflecting the exchange of a data item, any necessary relationship p-assertions indicating the causal connections between data items, and optional actor state p-assertions.

A concrete implementation of PReP is provided by PReServ [79], which contains a Provenance Store Web Service and a client side library for recording p-assertions and querying a provenance store. By using PReServ, applications can easily record process documentation to a provenance store.

2.3.4 Linking Multiple Provenance Stores

With the linking mechanism, an actor can record an interaction record into any provenance store. This means that the two actors in an interaction can employ two different stores to record their respective interaction record. In addition, an actor can record the interaction records about the effect and cause interactions captured by a relationship p-assertion into different stores.

There are two types of links, *viewlink* and *causelink*. Each actor records a *viewlink* in its interaction record, pointing to the provenance store where the opposite actor records its interaction record about that interaction. Therefore, both views of an interaction can be retrieved by navigating from one provenance store to the other. A *causelink* is embedded in the relationship p-assertion when the actor makes the p-assertion, indicating the provenance store where the interaction record about the corresponding cause interaction is stored in.

We now detail the linking mechanism in Figure 2.6, using the same example as in Figure 2.1. We assume A1 and A2 use provenance stores PS1 and PS2, respectively. An actor can obtain its viewlink via built-in knowledge or from request/response messages. In this figure, the address of PS1 or PS2 is exchanged to the other actor in messages M1 or M2. An actor then extracts store address and records it as its viewlink, i.e., VL1, VL2, VL3 and VL4. Table 2.4 summarises the content of IR1, IR2, IR3 and IR4 with their respective provenance store to be recorded into. Since A2 uses the same store to record IR2 and IR3, causelink CL in IR3 refers to PS2. After knowing the viewlink, an actor can record its interaction records to its respective store. Figure 2.6 shows PS1 and PS2 are interlinked via arc VL1, VL2, VL3, and VL4.

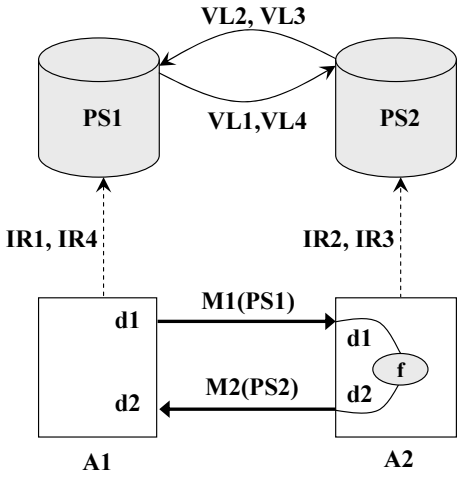


FIGURE 2.6: Using two provenance stores

PS	IR	Actor	Interaction Key	View Kind	IR Elements	Element Content
PS1	IR1	A1	I1	S	ipa VL1	M1 PS2
PS2	IR2	A2	I1	R	ipa VL2	M1 PS1
PS2	IR3	A2	I2	S	ipa rpa apa VL3	MS2 d2=f(d1), CL = PS2 version 1.3.2 PS1
PS1	IR4	A1	I2	R	ipa VL4	M2 PS2

TABLE 2.4: Interaction records with links appended (using two stores)

In order to retrieve the documentation of the process that led to data d2, a querying actor starts from PS1 which stores IR4 that describe the receipt of d2. Then by following the viewlink and causelink embedded in each interaction record, the other distributed records IR1, IR2, IR3 can be retrieved from PS1 and PS2. Figure 2.7 shows the retrieving path, where an interaction record is indexed by the tuple of an interaction key and a viewkind to help locate the interaction record in a store.

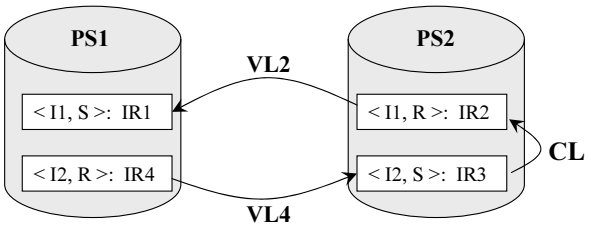


FIGURE 2.7: Retrieving documentation recorded in Figure 2.6

In the next example (Figure 2.8) where four stores are in use, we assume A2 sends message M2 to a third actor A3 , and A2 records IR2 and IR3 into two different stores PS2 and PS3, respectively. Each actor can obtain its viewlink from built-in knowledge or request/response message. In order to simplify the figure, we only show the exchange of PS1 and PS3 in messages M1 and M2. Table 2.5 give the contents of interaction records with links. Since A2 uses two stores to record IR2 and IR3, the causelink in IR3 now points to PS2. By following links, the documentation of the process that led to data d2 can be retrieved across the four provenance stores from PS4, as shown in Figure 2.9.

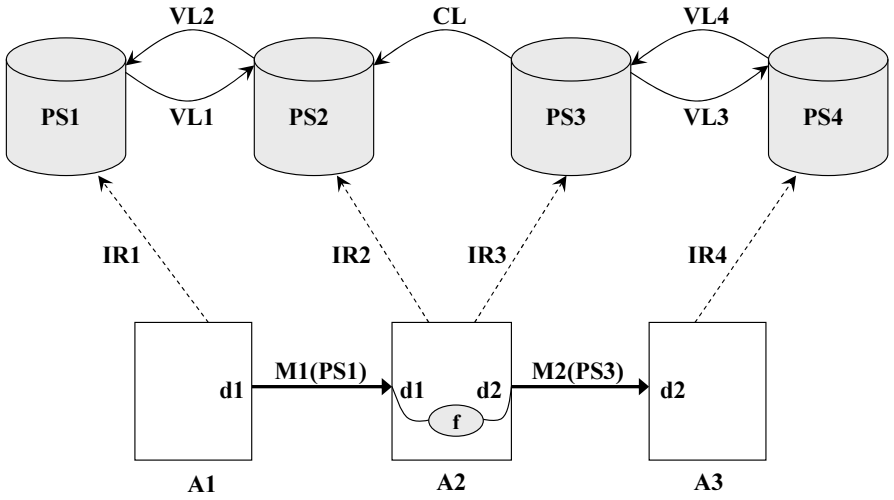


FIGURE 2.8: Using four provenance stores

PS	IR	Actor	Interaction Key	View Kind	IR Elements	Element Content
PS1	IR1	A1	I1	S	ipa VL1	M1 PS2
PS2	IR2	A2	I1	R	ipa VL2	M1 PS1
PS3	IR3	A2	I2	S	ipa rpa apa VL3	M2 d2=f(d1), CL = PS2 version 1.3.2 PS4
PS4	IR4	A3	I2	R	ipa VL4	M2 PS3

TABLE 2.5: Interaction records with links appended (using four stores)

In summary, a viewlink points to a store that contains the other actor's interaction record (which provides a different view on a same interaction). A causelink points to a store containing the interaction record asserted by the same actor (which is making assertions about a cause interaction). Two actors in a same interaction each record a viewlink pointing to the other actor's provenance store. Although only one viewlink is used when retrieving documentation, the two-way approach is useful in some cases where both views of an interaction need to be verified and compared.

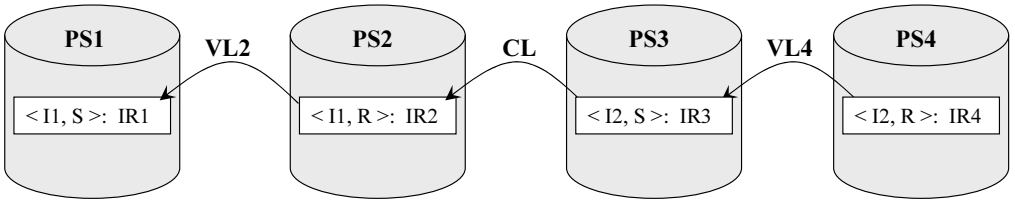


FIGURE 2.9: Retrieving documentation recorded in Figure 2.8

2.3.5 Summary

In this section, we highlighted key aspects of PASOA. PASOA uses an SOA-based approach to modelling a process as a set of causally related interactions between services through message passing. It defines several types of assertions to describe an interaction and introduces a protocol PReP to record process documentation. This approach is generic, open and independent of domains and technologies.

We also introduced an annotated causality graph, which is extracted from process documentation to represent the provenance of a data object. The linking mechanism was then detailed, which can flexibly connect a large number of provenance stores in order to retrieve distributed process documentation.

PASOA, however, does not consider failures that may happen when an actor is recording its interaction records into a provenance store. The consequence is the incapability to retrieve complete process documentation from multiple interlinked provenance stores either due to the loss of documentation or a broken pointer chain. The rest of this chapter will review research related to failures, fault-tolerant mechanisms and formal specifications for fault-tolerant protocols, which help us to develop our solution.

2.4 Failures

Distributed systems are generally modelled as asynchronous and synchronous systems [113][2]. An asynchronous system does not make assumptions about process execution

[2]The term of *partially synchronous* is also introduced to refer to various systems between completely asynchronous and completely synchronous [48].

speeds, message delivery delays and/or clock drift. On the contrary, a synchronous system makes timing assumptions, where the relative speeds of processes, any delays associated with communication channels and clock drift are bounded. Since all systems can be modelled as asynchronous systems, a protocol designed for use in an asynchronous system can be used in any distributed system [113]. Therefore, we are more interested in the failures of asynchronous systems.

Failures can be categorised by abstract models that describe how a system will behave in the presence of failures. A variety of failure models are commonly found in the literature of asynchronous distributed systems. All are based on assigning responsibility for faulty behaviour to the system's components: processes and communication channels. We summarise these models as follows.

- Crash. A process fails by halting. Once it halts, the process will not execute any further steps of its program ever [31].
- Crash-Recovery. A process fails by halting but later recovers [31, 8].
- Send-Omission. A process fails by halting, or intermittently omits to send messages it was supposed to send, or both [137].
- Receive-Omission. A process fails by halting, or intermittently omits to receive messages sent to it, or both [137].
- Channel-Omission. A channel fails by losing some messages, but does not create, duplicate or corrupt messages [137].
- Arbitrary (Byzantine) Failures. A process or a channel fails by exhibiting arbitrary behaviour [104].

Crash failures in asynchronous systems have been studied extensively. When a process crashes, it loses the content of its volatile memory. The crash-no recovery model has been considered unrealistic to a major part of applications [31, 8]. To model real distributed systems that support user applications, the crash-recovery model is proposed. In crash-recovery model, processes are provided with stable storage to log critical data in order to make them able to recover from crash failures. A process may keep on crashing and recovering indefinitely.

The other three models, send-omission, receive-omission and channel-omission, are all concerned with message loss. Each models a different cause for the loss and attributes the loss to a different component. Send-omission and receive-omission model overflows of local message buffers of a process, or the behaviour of a malicious adversary with control over the message flow of certain processes, or message loss due to a crashed

process. Essentially, for every send or receive omission failure, there is a process responsible for it; channel-omission failure puts the blame of a message loss to the unreliable communication channel.

Finally, arbitrary failures are the most disruptive. In addition to encompassing the above failures, it also represents random software or hardware faults as well as malicious attacks by a hacker. For example, a process or channel may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step and produce an incorrect result while continuing to interact with the rest of the system. A system that can tolerate arbitrary failures can tolerate any failures.

2.5 Fault Tolerance

Fault tolerance allows a distributed system to survive a variety of failures. As the size of a distributed system increases, the number of its components increases and so does the probability that some of its components will fail. Thus, fault tolerance must be considered when designing distributed applications.

Fault tolerance is carried out via detection and system recovery [15]. Failure detection is the first building block in the design, analysis and implementation of a lot of fault-tolerant distributed applications. A failure detection service detects that a failure has occurred so that a recovery procedure can be activated in order to bring the system back from a failure state to a normal state. Figure 2.10 outlines common techniques involved in fault tolerance. We firstly brief common techniques regarding detection (Section 2.5.1) and recovery (Section 2.5.2). Then we introduce concrete fault-tolerant approaches (Section 2.6). In Section 2.7, we discuss fault-tolerant solutions used in grid applications.

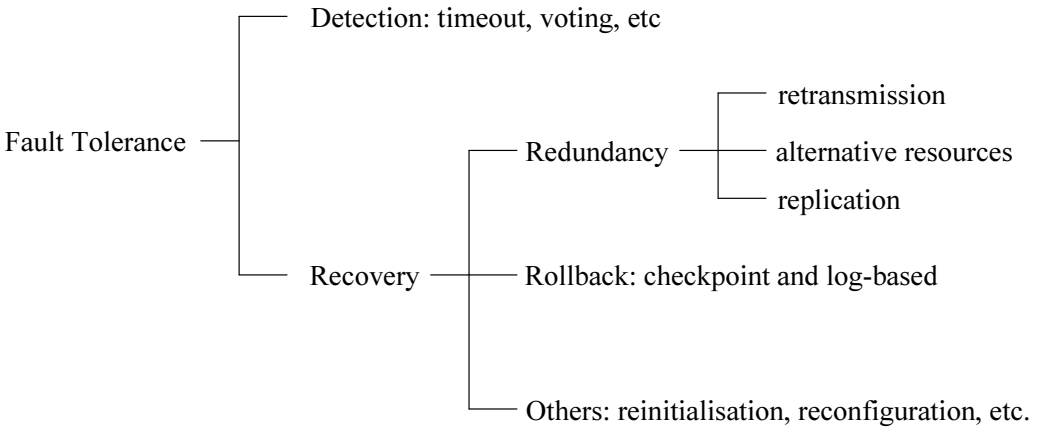


FIGURE 2.10: Fault Tolerance Techniques (revised from [15])

2.5.1 Detection

The timeout-based approach is commonly used to implement failure detection components such as failure detectors [31] and grid monitoring services [159, 85] to detect crash failures. Two types of keep-alive messages are usually used: heartbeat and ping [51, 85]. Heartbeat is a message periodically sent from a monitored process to the monitoring component (e.g., failure detector) to inform that it is still alive. If the heartbeat does not arrive before a timeout expires, the monitoring component suspects the process is down. On the other hand, ping is a message continuously sent from a monitoring component to a monitored process. The monitoring component expects to receive an acknowledgment as response before a timeout. However, it is difficult or even impossible to determine whether a process has actually crashed in asynchronous environments, where a process may be slow and keep-alive messages may be delayed or even omitted. Therefore, processes which do not contact a monitoring component or respond to its question in time are normally considered as suspect.

Failure detector deserves further introduction as it has been widely studied in distributed system literatures [143, 135]. It helps address problems that are impossible to solve in asynchronous systems prone to failures. One of the most famous problems is the consensus problem: the impossibility of deterministically reaching agreement among remote processes subject to crash failures in completely asynchronous systems. By augmenting the asynchronous system with unreliable failure detector, which may wrongly suspect that a correct process has crashed, the consensus problem can be solved under certain conditions with various failure models, such as crash model with reliable communication channels (i.e., they do not lose messages) [31] and crash-recovery model with channels that can omit messages [8].

Timeout is also used to facilitate reliable message delivery, as has been seen in many specifications such as TCP and Java Message Service (JMS). A timer is set by a sender after sending a message to the receiver, which is expected to provide an acknowledgement. If the acknowledgement is received by the sender before the timeout, then the sender can confirm that the message has been delivered to the receiver; otherwise, the sender cannot determine if the receiver has received or processed that message since either the original message or the acknowledgement can be lost, or the receiver can crash before providing the acknowledgement, all leading to a timeout on the sender's side. In the presence of a timeout, the sender can take remedial actions such as resending the original message.

Byzantine failures cannot be detected by seeing whether a process or an application responds to requests, because it might arbitrarily omit to reply or produce arbitrary results. A solution to detecting Byzantine failures is via a voting procedure [106], which invokes multiple functionally equivalent versions of a component and votes on their outputs or actions. In the event of a disagreement, those versions that do not meet

certain criteria are considered to be faulty.

2.5.2 Recovery

Redundancy (time redundancy or space redundancy) is one of the efficient ways to provide fault-tolerance for distributed systems [128]. Time redundancy is achieved by repeatedly performing key actions, e.g., message retransmission upon a timeout to mask transient omission failures, whilst space redundancy is achieved by using alternative components, or replicating software/hardware modules to provide backup capacity in order to tolerate failures.

Replication can take one of two forms: active replication and passive replication [16]. In active replication, a second machine receives a copy of all inputs to the primary one and independently generates an identical system state by running its own copy of all necessary software. In passive replication, a *cold spare* machine is maintained as a backup system to the primary. In both cases, should the primary machine crash does the second one take over control of the primary's responsibilities. The difference is that the changeover in the active mode requires a negligible amount of time at the cost of doubling the number of computing resources whilst that in the passive mode may incur some interruption of service. We will provide more details regarding replication in Section 2.6.2.

It should be noted that some redundancy-based techniques, such as duplicated copies of same software in different machines, assume independent system failures. They cannot tolerate correlated failures like a software bug in all copies of the software. In this case, Multi-Version Design (MVD) is favoured, where two or more systems aim at delivering the same functionalities through separate designs and implementations [107]. MVD is also used to tolerate Byzantine failures [106]. As introduced in Section 2.5.1, the voting procedure weights results from multiple functionally equivalent versions of a component, compares their outputs with the consensus output, and forwards the consensus output as the final system result. Thus Byzantine faults can be tolerated by discarding those outputs that do not meet specified criteria.

Another common approach to handling failures is to use checkpointing techniques to save system state, and rollback techniques to revert back to a state that was saved before a failure occurred. The challenge is to orchestrate checkpoints and rollbacks, while maintaining high-performance and scalability. This approach has been studied in multiple contexts, including Messaging Passing Systems [49], Web Services [45] and Transactions [75].

In the case of Messaging Passing Systems, rollback recovery techniques model a message-passing application as a fixed number of processes in a distributed system that communicate over a network by sending and receiving messages. A process is assumed to have

access to some kind of stable storage that will survive even in the event that the process crashes. During the execution of an application, the system periodically records a snapshot of processes composing the application to the stable storage. In the event that a process fails, the application's computational state can be restored to some failure-free state by rolling back all processes to the most recent checkpointed state.

In addition to checkpoint-based rollback, there is a log-based rollback technique, which supplements normal checkpointing with a record of messages sent and received by each process. If a process fails, the log can be used to replay the progress of the process after the most recent checkpoint in order to reconstruct its previous state. This has the advantage that process recovery results in a more recent snapshot of the process state than checkpointing-based technique can provide. The range of applications that benefit from these rollback techniques tend to be long-running, scientific applications [49].

Reinitialising or restarting a system is a good way of dealing with some kinds of faults [159]. However, this may result in significant system down time and may restart correctly functioning components. To minimise these side effects, system should be divided into small and independent pieces in a well organised approach. If a fault occurs in one of these components, those components that are directly affected can be restarted while the rest of the system remains untouched and continues working. For example, a web server could be dependent on a database. If the web server experiences a crash, it should be restarted individually without interfering with the database.

2.6 Major Fault Tolerance Solutions

In this section, we introduce several fault-tolerant approaches: transactions, cluster-based architectures and messaging middleware. These approaches have been widely used to build reliable distributed applications and databases, thus being possible candidates to solve the problem of reliably recording process documentation.

2.6.1 Transactions

Transactions have been used to provide reliable information processing in many application domains from classical *debit-credit* style centralised and distributed database operations, to more recent workflow management and Web Services. The amount of published research work on transactions is huge and a number of survey papers and books have been published [162, 163].

In the database world, a transaction consists of a group of operations executed to perform some specific functions by accessing and/or updating a database. From a broader range of application areas such as workflow management, Web services and Grid computing,

a transaction refers to a reliable and coherent process unit interacting with one or more systems, independently of other transactions, that provides a certain service or function for a running application [162].

The checkpoint and log based mechanism is used to support failure recovery in database transactions. When creating a database checkpoint, the entire state of a database is made persistent, e.g., all operations stored in volatile memory are written to stable storage. When a transaction fails, the database can rollback to the state created at the checkpoint, and/or redo any operations performed thereafter using the information stored in persistent transaction logs. Such a mechanism is widely used in current database systems, such as Oracle Database, IBM DB2, Berkeley DB.

A distributed transaction consists of sub-transactions that may access multiple local database systems. These transactions typically require *all-or-nothing* atomicity to maintain system consistency [163] via several approaches, e.g., Two-Phase Commit (2PC) protocol. 2PC is a simple and elegant distributed algorithm that results in either all parties committing the transaction or aborting, even in the case of network failures or crash failures. A coordinator manages all participants, coordinates their decisions to start, commit or roll back, and ensures atomicity at a global level. For example, a distributed database transaction may contain a sequence of database operations, spawning several sites to read or update data objects. A global commit decision of updating a data object is based on the agreement of all the participating sites. Should any of these sites abort the updating operation, a global decision of abort is made by the coordinating site. Consequently, all sites abort updating the data object, which ensures the consistency of distributed databases.

However, in some long-running business transactions the *all-or-nothing* atomicity is not always possible to be held because parts of a transaction may have been committed or because parts of a transaction (e.g., communications with external agents) are impossible to undo. In such cases, *compensations*, actions taken to recover partial executions of transactional processes [29], can be used as a way of dealing with faults [25].

2.6.2 Cluster-based Architectures

Many systems are based on clusters where a number of computing resources simultaneously share the load and act as a backup to each other. Cluster-based architectures take advantage of resource redundancy to meet both high availability and scalability requirements. To have redundant resources, data needs to be replicated across all servers or databases (for availability) and the load is partitioned to use all available resources (for scalability).

In a typical cluster, as exemplified in Figure 2.11, the network traffic is first offered to the cluster head(s), where a load balancer is instantiated to route incoming requests to an

appropriate processing server. Each component of the cluster has redundant components in order to eliminate single point of failure at the entry or inside the cluster. Stateful components such as databases are replicated on a number of replicas during failure-free periods to maintain consistent state. Should the primary component fail, a backup component takes over control of a failed one's responsibilities.

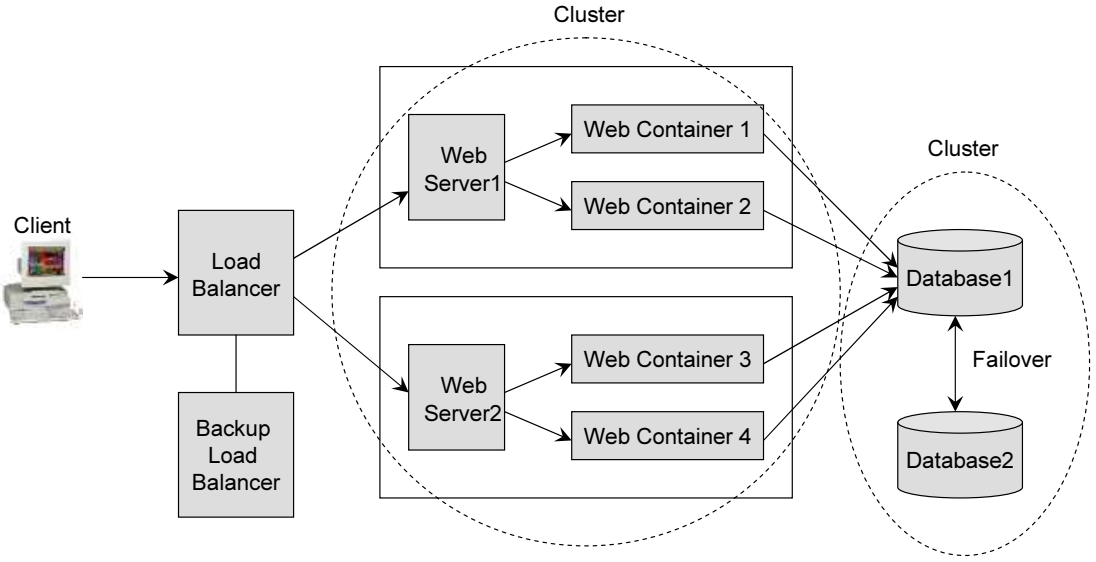


FIGURE 2.11: An example of cluster-based architecture [145]

As introduced in Section 2.5.2, there are two types of replication mechanisms: active and passive replication. The passive replication also has two main forms: checkpointing approach and message logging approach [16]. Active replication has short recovery time at the cost of performing redundant computation in all replicas whilst passive approaches may have long recovery time and introduce some interruption of service. All replication mechanisms tend to be sophisticated in order to balance the tradeoff between preserving consistent state in replicas and reducing replication overhead. The following summary is based on [16].

The active replication ensures that all redundant machines receive a copy of all inputs to the primary one and maintain the same consistent system state by running its own copy of all necessary software. In addition, this approach guarantees that only one machine is replying to a client at a given time by consolidating outputs from all replicas. In order to ensure that a group of replicas reliably receive the offered requests sent to the primary one, different techniques can be used. The first technique is based on protocols originally considered for setting up group membership, e.g., the atomic broadcast/multicast protocols. A second solution consists of delivering the traffic exchanged between a client and the replicas to an intermediate gateway or proxy that would reliably perform one-to-many message delivery to the replicas on one hand and many-to-one message delivery to the client on the other hand.

In the checkpointing approach, the primary machine's state is periodically copied to

standby machine(s). If the primary node fails, the most recent checkpoint is recovered, so that the processing can resume using the restored state. Different checkpointing approaches exist, differing in terms of their frequency and completion time. The most aggressive checkpointing approach is incremental checkpointing, which aims to maximising the consistency of the replicated states by performing checkpoints each time a state change occurs at the primary node. The major drawback of this approach is its cost in CPU consumption at the primary node and the added latency to end-to-end communication.

Message logging approach is to redundantly store or log on a replica all the messages delivered to the primary server. For reliability purposes, a message would not be processed until an acknowledgment is received from the replica confirming that the message has been successfully stored. During failsafe periods, replicas are idle. Once the primary server fails, the logged messages are replayed and reprocessed on the elected replica. Different approaches exist for message logging. Pessimistic message logging logs a message into a stable storage as soon as it is received. Dependency-based message logging proposes to copy each received message into a volatile log space, which will be flushed into stable storage once it becomes full. Optimistic message logging also copies the incoming message into a volatile log space, but proposes to flush it on stable storage periodically or when the number of logged messages reaches a given threshold.

2.6.3 Message Oriented Middleware

The Java Message Service (JMS) API [4] is a Java Message Oriented Middleware (MOM) API for reliably sending messages between two or more clients. Message senders do not need to have precise knowledge of message receivers, since communication is performed via an intermediary component, a messaging queue. Messages are placed onto the queue and stored there until the recipient retrieves them. Message queueing systems typically provide resilience functionality to ensure that messages do not get lost in the event of a system failure. There are many free, open source and proprietary message queue products, such as OpenMQ, IBM WebSphere MQ and Tibco Enterprise JMS.

Figure 2.12 illustrates a typical architecture of a message queueing system. At the heart of the system is a broker [4, 42]. The broker reliably delivers messages, and provides administrative tools to manage, monitor and tune the messaging system. In order to send or receive messages, a JMS client (a message producer and/or a message consumer) must first connect to the broker before producing or consuming messages. Message transmission between producer and consumer can be based on point-to-point or publish/subscribe pattern. Using point-to-point pattern, a client sends a message to a queue destination from which only one receiver may get it. With the publish/subscribe pattern, a client sends a message to a topic destination from which any number of consuming subscribers can retrieve it.

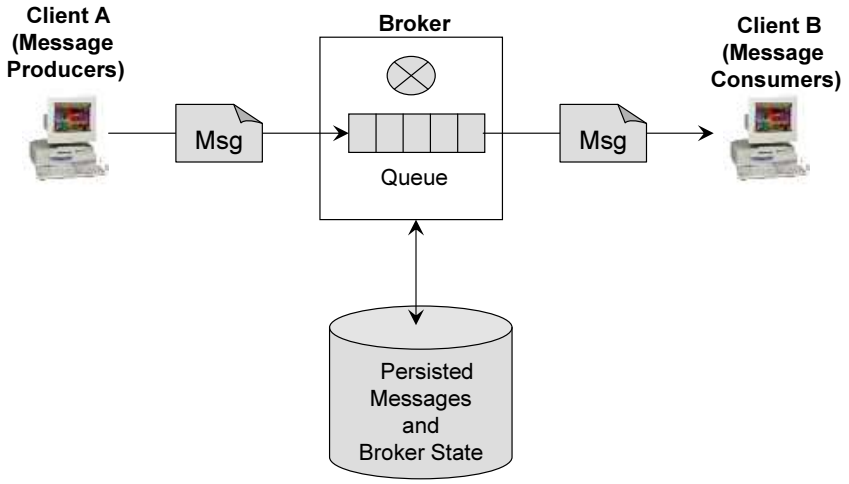


FIGURE 2.12: Overview of a Message Queueing System

There are several approaches to reliable message delivery. The broker may persist messages and its states in a stable storage so that if the broker fails before the message is consumed, the broker can retrieve the stored copy of the message and state information to retry an operation upon recovery. Acknowledgements are also used between a message producer/consumer and the broker to ensure reliable delivery of messages. In the case of message production, the broker replies the producer with an acknowledgement confirming that it has received the message and has stored it persistently. The producer blocks until it receives this broker's acknowledgement. In the case of message consumption, the consumer acknowledges that it has received a message and consumed it. Upon receiving the consumer's acknowledgement can the broker delete the message from its persistent storage. JMS specifies different acknowledgement modes that balance the tradeoff between reliability and performance in different degrees. In addition, brokers can be interconnected into a cluster: a set of brokers that work collectively to perform message delivery between message producers and consumers. Clustered brokers further enhance scalability and availability to a Message Queue service.

The JMS API also supports distributed transactions, which means the production and consumption of messages can be part of a larger, distributed transaction that includes operations involving other resources, such as database systems. The JMS message service tracks the various send and receive operations within the distributed transaction, persists transactional states, and completes the messaging operations using a two-phase commit protocol. We note a JMS messaging service supports distributed transactions only when it is used in a Java Enterprise Edition (Java EE) application server [4], which provides a Java Transaction Service (JTS) as a distributed transaction manager.

2.6.4 Discussion

This section discusses how transactions, cluster-based architecture and messaging middleware can help us to reliably record process documentation.

Provenance store has been designed to maintain process documentation in a database, therefore local database transactions are useful when dealing with database operations. We now discuss distributed transactions, which involve a transaction manager, an actor and a provenance store during the recording of the actor's interaction record. A distributed transaction ensures that either the interaction record is successfully recorded in the store and removed from the actor's memory, or it remains in the actor's memory without being recorded in a store. When the distributed transaction is committed, the actor knows that its interaction record has been successfully recorded in the store and been safely removed from its memory. If the provenance store fails, the actor is notified that the transaction is aborted. Then the actor can resend the interaction record to the same store or another store until the transaction is committed.

This approach is feasible but comes with performance overhead for reasons including rollback segments maintenance, forced logging, connections with transaction manager, and the cost introduced by commit protocols. Given that a process may involve thousands or tens of thousands of interactions and each interaction will have two distributed transactions (one in the sender side and one in the receiver side), the performance penalty can be significant.

Instead of using distributed transactions, we adopt a looser consistency model. An actor can set a timeout after sending its interaction record to a provenance store, which provides an acknowledgement after successfully recording the message. Only after the actor receives an acknowledgement from the store can it remove the message from its local memory. If it does not receive the acknowledgement before the timeout, it can take remedial actions such as resending the interaction record to the same or an alternative store until the message is acknowledged before a timeout. The use of timeout, however, may result in redundant information recorded in a provenance store in the case where the store actually received the interaction record but the actor saw a timeout and used an alternative store. To reduce the probability of having potential redundant information, an actor can be configured to resend an interaction record to a same store for certain times before using an alternative store.

Provenance stores can be organised as a cluster sharing a same store address. If the primary store fails, another one takes over the work of the failed one so as to provide services to actors.

Messaging middleware can also be used to reliably transport process documentation. As discussed in Section 2.2.2.2, third party messaging services (OpenMQ and WS-Messenger) have been used in CMCS and Karma. A messaging service typically requires

the configuration of a broker, residing between a client application and a provenance store. In the event of a provenance store crash, process documentation is persistently maintained in the broker until the store comes back online and consumes the documentation.

Both approaches can support reliable recording of provenance information but with limitations. Firstly, in open distributed environments like SOAs, where there could be a large number of provenance stores managed by different institutions across different regions, we should not assume the use of clustered provenance stores or messaging middleware in all situations. In addition, these approaches do not deal with disconnected process documentation that could appear in the presence of failures. Therefore, we aim to design a generic approach to reliably recording process documentation while ensuring its retrievability. These approaches, however, can be complementary to ours when necessary.

2.7 Fault Tolerance for Grid Applications

Section 1.3 states that the Grid community has reported many results regarding failures. Therefore, this section overviews fault detection and handling/recovery mechanisms for Grid applications such as GWE, Triana, Unicore, Pegasus, Condor, DAGMan, etc. Most of this section is based on survey [139].

2.7.1 Fault Detection

In Grid applications, failures can take place at Hardware, Operating Systems, Middleware, Workflow Task, Workflow or User level. At the lowest level, Hardware level, machine crashes and network disconnection can happen. At the level of Operating Systems, tasks may run out of memory or disk space, or exceed CPU time limits or disk quota. At the Middleware level, non-responding services can be found, probably caused by too many concurrent requests. Authentication, file staging or job submission failures can also happen, and submitted jobs could hang in local queues, or even be lost before being executed. At the Task level, job-related faults can happen, like deadlock, memory leak, uncaught exceptions, missing shared libraries or job crashes, even incorrect output results could be produced. At Workflow level, failures can occur in data movement or infinite loops in dynamic workflows. Incorrect or not available input data could also produce faults. Finally, at the highest level, the User level, user-definable exceptions and assertions can lead to failures.

Many Grid monitoring services have been developed to monitor Grid applications and detect various failures. The pinging-and-timeout mechanism is used to detect task failures. A task is considered to have failed if the monitoring service does not receive a

response from the task within a certain amount of time. Hwang et al. [90] presented a failure detection service (FDS) on the Grid, aiming to detect both task crashes and user-defined exceptions. A notification mechanism is developed based on the interpretation of notification messages being delivered from different entities (i.e., the task itself, the Grid server and the heartbeat monitor) residing on each Grid node.

2.7.2 Fault Recovery and Fault Handling

Fault recovery and handling techniques for Grid application mainly fall into two levels: Task level and Workflow level.

Task-level techniques refer to recovery techniques that are to be applied in the task level to mask the effect of task crash failures. These techniques include retrying, checkpointing, etc. Upon detecting a failure, a failed task is rescheduled to either the same or an alternative resource to reattempt. Resubmission can cause significant overheads if the following tasks have to wait for the completion of the failed task and if a failed task has to be restarted over from the beginning. A good solution to this is to save checkpoints and resume task execution from the last checkpoint later.

Workflow-level techniques refer to recovery techniques that enable the specification of failure recovery procedures as part of application structure. These techniques include using alternative tasks, data and workflow redundancy, checkpointing and transactions. Redundancy means one task is executed concurrently on several resources, assuming that one of the tasks will survive any independent failure. It can cause overhead by occupying more resources than necessary, but guarantees failure-free execution as long as at least one task does not fail. Checkpointing technique can also be used to save an intermediate state of a whole workflow for a restart later. Distributed transactions can also be used for handling failures in workflow level, which has been used in Condor.

There are several other approaches to supporting reliable Grid services. A web service-based messaging middleware, WS-Messenger, is developed to deliver messages for SOA-based Grid applications [88]. By wrapping up the underlying message queuing systems, WS-Messenger creates interoperable Web services-based publish/subscribe systems to decouple event producers and consumers, and achieve scalable, reliable and efficient message delivery. In addition, as has been introduced in Section 2.2.3, a fault tolerance framework, FT-Grid [169], is developed to tolerate software faults that occur in SOA-based Grid applications through multi-version design (MVD) and the use of provenance information.

2.8 Formal Methods for Fault Tolerance

Fault-tolerant protocols are designed to be resistant to failures. Proving the resistance of protocols to failures is very challenging, 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. Formal methods can help us to rigorously develop and reason about fault-tolerant protocols.

Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems [165]. The application of formal methods makes it possible to achieve provable correctness and reliability in the various steps of system design and implementation. A formal specification precisely and unambiguously describes what a system should do. The use of formal specification reduces the complexity of a system by hiding irrelevant details so that users can understand the system without understanding all the details of its construction. Once a formal specification has been produced, the specification can be used as a guide to develop the concrete system. For example, the observed behaviour of the concrete system can be compared with the behaviour of the specification. Given a formal specification, it is possible to use formal verification techniques to demonstrate that a candidate system design is correct with respect to the specification. This has the advantage that incorrect candidate system designs can be revised before a major investment has been made in actually implementing the design.

2.8.1 Fault Tolerance Specification

Based on papers by Gartner [70, 69], this section generally introduces formal specification for fault tolerance.

When specifying interactive systems it is necessary to distinguish between the system and its environment. A system is usually defined as a "thing" that interacts with its environment in a discrete fashion across a well-defined boundary (called an interface). The environment consists of all "things" that have access to the interface of the system.

A system may be constructed using many subsystems, each of which having its own interface and its own specification. For example, in a distributed system consisting of n processes $p_1, \ldots, p_n$ that communicate via a communication subsystem, each process as well as the communication channel form a larger system (Figure 2.13). Each subsystem is part of the environment of the other. The specification for a larger distributed system defines what tasks should be solved by coordinated actions of its subcomponents.

A specification of a system S asserts that S will guarantee a property M under the assumption that the environment guarantees some property E (formally, $E \Rightarrow M$). In the example of a specification for process p_i, E will define what p_i can expect from the

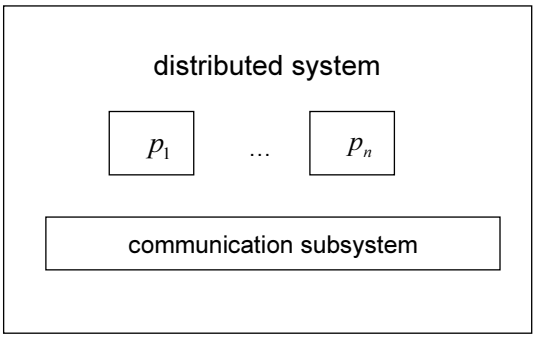


FIGURE 2.13: System and Environment [70]

communication subsystem (i.e., the type and order of messages received) and M will describe how p_i reacts (i.e., what messages it will send).

We now use a state-based approach to illustrate a property. A *state* is some assignment of values to the "variables" of an interface. The set of all possible combinations of value assignments to variables is called the *state space*. An *execution* of a system is a sequence of states and a *property* is a set of executions. A system by itself defines a property, which is the set of all executions starting from an initial state. A property P *holds* for a system S if the set of executions defined by S is contained in P. A state change is called an *action*. State changes are either initiated by the system or by the environment.

The correctness of a distributed protocol typically has two kinds of requirements: safety and liveness [113, 101]. The safety requirement informally states that some bad thing will never happen. It imposes that a certain property must hold for each execution of the system in *every* configuration reached in that configuration. On the other hand, the liveness requirement informally states that some good thing will eventually happen. It imposes that a certain property must hold for each execution of the system in *some* configuration reached in that execution. Termination is an example of liveness property.

Given $E \Rightarrow M$, a fault tolerance specification is usually obtained by weakening either E or M (or both). Weakening E resembles the anticipation of abnormal behaviour of the environment, whereas weakening M indicates that the system itself will sometimes deviate from its original failure-free specification.

As introduced in previous sections, a failure model describes the manner in which system or subcomponents of a system may fail. Volzer [160] observed that failure models formally consist of two distinct parts: (1) the impact model, a specification of the additional faulty behaviour of the system, and (2) the rely specification, a specification of assumptions that restrict the set of possible system executions. The impact model is responsible for weakening the ideal (failure-free) specification of some system (i.e., M): more executions become possible through the added behaviour. On the other hand, the rely specification is usually a global assumption (i.e., E) on a composed system, bounding the additional behaviour from becoming too bad.

For example, a system crash can be modeled by adding a boolean variable *up* to the state of the system and by inhibiting all affected transitions if $\neg up$ holds. The state transition from *up* to $\neg up$ can be viewed as a crash failure. In addition, a global failure assumption is usually made to be "at most $t < n$ processes may crash".

The idea to represent failures as additional program actions goes back to a paper by Cristian [39]. It was further developed in a series of papers by Arora et al. [12, 14, 13], who stress that every form of failure can be modeled by this method. Gartner [69] gives formal specifications of several failure models such as Crash, Send Omission, Receive-Omission, General-Omission and Byzantine.

2.8.2 A Brief Survey

Gartner [71] surveys methods to specify and verify fault-tolerant systems based on a notion of transformation. A transformation is a general concept and almost anything where there is a notion of "change" can be formulated as a transformation. So it has been interesting to formulate fault tolerance methodologies which do not directly refer to the notion of a transformation, e.g., multitolerance [13] and the state machine approach [148].

Transformations offer the potential of being automated and thus can help aid the mechanical verification of fault-tolerant systems. For example, a failure model is a transformation, i.e., a function that maps a program A to a program A'. Program A is the original program, which by itself runs in an ideal fault-free environment; A' is program A that may be subject to failures. We note that A' will never be explicitly implemented; the transformation process is just a means to be able to reason about transformed programs to evaluate fault tolerance properties.

The state machine approach was first described in Lamport [102] for environments in which failures could not occur. It was generalised to handle fail-stop failures in Schneider [147], a class of failures between fail-stop and Byzantine failures in Lamport [102], and full Byzantine failures in Lamport [103].

The input/output (I/O) automaton model [112], developed by Lynch and Tuttle, is a labelled transition system model for components in asynchronous distributed systems. The I/O automaton is a simple type of state machine in which transitions are associated with named actions, classified as either input, output or internal. The inputs and outputs are used for communication with the automaton's environment, while the internal actions are visible only to the automaton itself. This model has been used to describe faulty communication channels and process crash failures [113]. The proof method supported in the automaton model for reasoning about the system involves invariant assertions. An invariant assertion is defined as a property of the state of a system that is true in all executions. A series of invariants relating state variables and reachable states are

proved using the method of induction. The work done using I/O Automaton has been carried out by hand [68, 113].

B [5] is a tool-supported formal method based on Abstract Machine Notation. The B Method adopts the top-down approach to system development, which starts from creating a formal system specification and continues with refinement of the specification. In B, a specification is represented by a collection of modules, called Abstract Machines. An abstract machine encapsulates a local state (local variables) of the machine and provides operations on the state.

Recently, another formal method called Event-B [5] has been developed, which is considered an evolution of B. Event-B is particularly suitable for developing distributed, parallel and reactive systems. It is a simpler notation and comes with tool support in the form of the Rodin Platform [6]. The operations in Event-B are called *events*, which are atomic meaning that, once an event is chosen, its execution will run until completion without interference. The guard of an event represents the necessary conditions on the state of the system for the event to be triggered. When the guard is true, event actions are executed, possibly changing the state and allowing another event to be triggered.

Applying formal methods to the design and reasoning of fault-tolerant protocols has been practiced in many applications, e.g., mobile agent systems [100, 121], control systems [99, 168] and replicated distributed database systems [170].

Moreau [121] presented a fault-tolerant directory service for mobile agents, which can be used to route messages reliably to them, even in the presence of crash failures of intermediary nodes between senders and receivers. This algorithm relies on redundancy information stored in different locations, hence able to tolerate a maximum of $N - 1$ failures of intermediary nodes. The distributed directory service is formalised as an Abstract State Machine (ASM). The formalisation adopts the impact model and the rely specification, introduced in Section 2.8.1, to model crash failure and communication omission failures by adding a boolean variable and several transitions regarding failures to the system. Liveness and safety properties are stated and proved by hand based on mathematical induction. A fully mechanical proof of the algorithm's correctness is also derived using the proof assistant Coq [36].

Xu et al. [168] used coordinated atomic (CA) actions to design and validate a sophisticated and embedded control system that has high reliability and safety requirements. Their work was based on an extended production cell model, which represents a manufacturing process involving redundant mechanical devices to maintain specified safety and liveness properties even in the presence of device and sensor failures. They formalised CA action-based designs as a state transition system, which is characterised by its (global) state space, a set of initial states, and a next-state relation.

Yadav and Butler [171] used Event-B to design fault-tolerant transactions for replicated

distributed database systems. They analysed the problem of formation of deadlocks among conflicting transactions and outlined an approach to preventing deadlocks and transaction failures. They also demonstrated how to formally verify by refinement that the design of a replicated database confirms to the one copy database abstraction.

Laibinis et al. [100] presented a formal approach for the development of fault-tolerant mobile agent systems based on Event-B framework. They started from an abstract system specification modelling agents together with their communication environment and gradually introduced implementation details in a number of correctness-preserving transformations. In the refinement steps, loss of connections and agent failures are considered and modelled by additional events in the system. To tolerate loss of connections, a timeout mechanism is adopted, modelled by a variable $timers$ in the formalisation. In addition, several variants, such as $disconn\_limit$, $recover\_limit$ are introduced to avoid deadlock by limiting the number of successive disconnections and the amount of error recovery attempts, respectively. They are initialised with an intial value and their values decrease by 1 whenever a related event occurs. As soon as for some agent $recovery\_limit$ becomes zero, the agent's error recovery terminates and the error is treated as irrecoverable.

2.9 Summary

This chapter provided relevant background information on provenance, failures, fault-tolerant mechanisms and formal method for fault tolerance. We also positioned our work and discussed various solutions to addressing our problem.

We firstly reviewed provenance research. Provenance has attracted attention in many fields, especially in fast growing workflow communities, which support SOA-based *in silico* scientific applications. We then compared several major provenance systems for SOA applications. Through this comparison, we demonstrated that PASOA is a generic, domain and technology independent approach to modelling and recording process documentation. This conclusion was further supported by a wide range of applications that used PASOA to record their provenance information.

Then we highlighted key aspects of PASOA. PASOA models a process as a set of causally related interactions between services through message passing. It defines several types of assertions to describe interactions and introduces a protocol PReP to record process documentation. Based on this part of the survey, we have decided to extend PASOA in our work in order to inherit the following advantages.

PASOA has a number of advantages over other provenance systems. Firstly, it is domain and technology independent. It uses an SOA approach to modelling provenance information, supporting different domain applications. It also specifies a generic record-

ing protocol PReP, which can be implemented in different languages, such as Java and Python. Secondly, it is an open architecture. It allows application developers to customise the granularity level provenance information is collected at. Thirdly, it supports multi-site recording, i.e., all participating parties in a process, such as workflow enactment engine and services, contribute to the provenance information of the data product of that process. This is crucial for obtaining sufficient provenance data in highly decentralised applications. Fourthly, it introduces a linking mechanism to connect multiple provenance repositories, which have been demonstrated in many applications to be necessary.

PASOA, however, does not consider failures that may happen when an actor is recording its interaction records into a provenance store. The consequence is the incapability to retrieve complete process documentation from multiple interlinked provenance stores either due to the loss of documentation or a broken pointer chain. Therefore, we introduced several failure models and surveyed major fault-tolerant techniques. We aim to design a *generic* approach to reliably recording process documentation for SOA-based applications without the assumption on specific implementation strategies.

Formal methods can assist us in rigorously designing a fault-tolerant distributed protocol and verifying its correctness. Therefore, we will use them to formalise our solution and establish properties to prove its correctness.

With these conclusions, we now present our approach in the next chapter.

Chapter 3

Protocol

At the beginning of this dissertation, we outlined the need for recording process documentation in the presence of failures while ensuring its entire retrievability for SOA-based applications. Furthermore, we surveyed a number of provenance systems and concluded that we adopt PASOA's approach to modelling process documentation and extend PReP to inherit a number of advantages.

This chapter now presents our solution F-PReP, a coordinator-based protocol to record process documentation in large, open distributed environments where numerous provenance stores could be present and failures may occur.

The main idea of our protocol is to analyse all the possible behaviours that a system can exhibit in the presence of failures and to provide remedial actions when failures occur. Basic fault-tolerant mechanisms such as timeouts, retransmission of messages and alternative provenance stores are used to guarantee the recording of complete process documentation. To preserve the retrievability of distributed documentation, an Update Coordinator is introduced to update incorrect links in provenance stores so that multiple provenance stores are still properly connected.

The contributions of this chapter are twofold:

Firstly, we describe a generic protocol, F-PReP, for recording process documentation in the presence of failures (specifically, provenance store crash and communication failures). It is a distributed protocol specifying the behaviour of three components: recording actor, provenance store and the update coordinator. We define protocol messages and their exchanges between components.

Secondly, we formalise F-PReP using an Abstract State Machine (ASM) approach. We begin with a formalisation without consideration of failures. Then we model failures by extending system state space and adding extra transitions. Our ASM-based formalism provides a precise and implementation independent means of specifying the protocol.

It sketches the essence of the protocol and accurately defines protocol's behaviour. In addition, it promotes a rigorous design of the protocol and helps us better understand the complex system behaviour in the presence of failures.

This chapter is organised as follows. Section 3.1 recalls several terms defined by PASOA. Sections 3.2 and 3.3 state assumptions on the kinds of failures that we consider and present the requirements that F-PReP supports. In Section 3.4, we outline F-PReP's design philosophy based on the analysis of PReP's problems in the presence of failures. After that, we define protocol messages and detail F-PReP in Section 3.5. Section 3.6 formalises the protocol and specifies the internal behaviour of recording actors, provenance stores and the coordinator. Finally, we discuss the protocol and conclude this chapter.

3.1 Terminology

We expect readers to be familiar with Section 2.3, which provides an introduction to PASOA. This section recalls several terms that are important to the understanding of our work.

In service-oriented architectures, clients typically invoke services, which may themselves act as clients for other services. The term *actor* is used to denote either a client or a service. An actor that sends an application message is referred to as a sender, whereas an actor that receives an application message is known as a receiver. One application message exchanged between a sender and a receiver is an *interaction*. A *process* is modelled as a causally connected set of interactions between actors involved in that process.

We use the term *assertor* to refer to an actor that creates and records p-assertions. Using *p-assertions*, an assertor documents an interaction to provide a sender or receiver's view of the interaction and how those interactions are related. In order to efficiently locate and extract p-assertions from process documentation, PReP specifies that all p-assertions created by an assertor about the same interaction are submitted to a provenance store in a single message, which is termed an *interaction record*. Process documentation therefore consists of a set of interaction records. PReP specifies that *both* assertors in an interaction must make their interaction record documenting the interaction for accountability or verification purposes.

One kind of p-assertion is *interaction p-assertion*, which documents the content of an application message and is created by the assertor that has sent or received that message.

Another kind of p-assertion is *relationship p-assertion*. It is made by an assertor to describe how the assertor produced output data from input data that it received by applying some function on the input data. The output data is the *effect* and the input

data is the *cause*. An effect can have multiple causes. We use terms *effect interaction* and *cause interaction* to denote the interactions where the effect and cause are transferred, respectively. Hence, a relationship p-assertion captures causal connections between an effect interaction and cause interaction(s). A relationship p-assertion has been specified by PReP to be included in the interaction record about the effect interaction since it describes the causal reason for the occurrence of the effect interaction.

In the example of Figure 3.1, we assume that output data $d2$ is produced by assertor a, which applies a function f on the input data $d1$. Hence, $d2$ and $d1$ are the effect and the cause, respectively. Correspondingly, interactions $I2$ and $I1$, where $d2$ and $d1$ are exchanged to/from other assertors, are effect interaction and cause interaction, respectively. A relationship p-assertion can be created to capture the causal connection (f) between $I2$ and $I1$, and it is sent to a store as part of $IR2$.

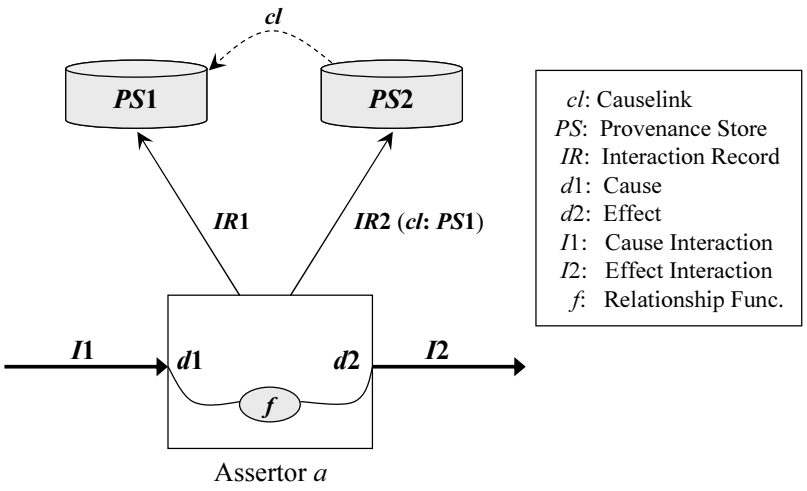


FIGURE 3.1: An example of a simple process and causelink

For scalability, an assertor can use various stores to record interaction records about different interactions. A notion of link, i.e., a pointer to a provenance store, has been introduced to connect multiple provenance stores. PASOA defines two types of links, *causelink* and *viewlink*. By following links, distributed process documentation can be retrieved from across multiple stores.

A *causelink* is embedded in the relationship p-assertion, indicating the provenance store that records the interaction record about the corresponding cause interaction. The assertor has the knowledge about causelinks at deployment time. For example, a causelink can be determined from a configuration file or be hard coded into an application's program. In Figure 3.1, the assertor creates two interaction records $IR1$ and $IR2$ documenting the cause interaction $I1$ and effect interaction $I2$, respectively. It also makes a relationship p-assertion, capturing the causal relationship between $I1$ and $I2$. The assertor uses provenance stores $PS1$ and $PS2$ to record $IR1$ and $IR2$, respectively. Therefore, it embeds a causelink to $PS1$ in the relationship p-assertion, which is included in $IR2$ and recorded to $PS2$. Then there is a causelink in $PS2$, pointing to $PS1$ where the

interaction record about the cause interaction is recorded.

Another type of link is *viewlink*. Each assertor includes a *viewlink* in its interaction record, pointing to the provenance store where the opposite assertor records its interaction record about the same interaction. Therefore, both views of an interaction can be retrieved by navigating from one provenance store to the other. *Each assertor must have a viewlink before it submits its interaction record to a provenance store.* The viewlink can be built into an assertor at deployment time or obtained from application messages from another assertor. In Figure 3.2, the sender sends an application data to the receiver in an interaction I. Both sides document their view of the interaction by creating interaction records $IR1$ and $IR2$. After recording its interaction record with a viewlink into a provenance store, we know that $IR2$ is recorded in $PS2$ by checking the viewlink in $IR1$ from store $PS1$, and vice versa.

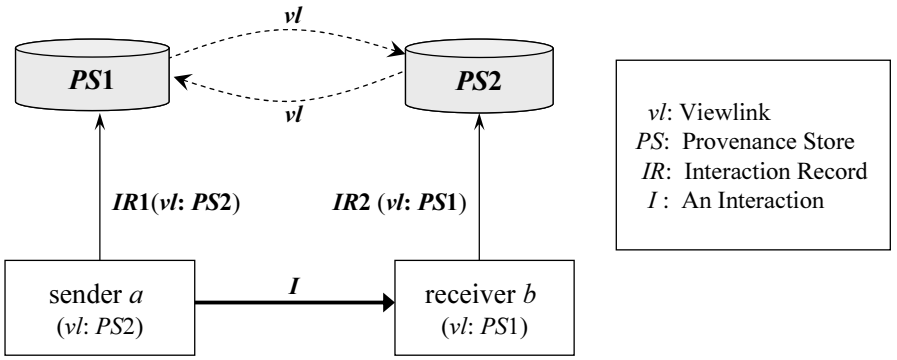


FIGURE 3.2: An example of viewlink

3.2 Assumptions

Failures are non-deterministic in nature and typically very hard to predict. Restricting our scope to particular failures is hence necessary. We state the following assumptions.

Assumption 1. *We assume a crash-recovery model on a provenance store, i.e., a store fails by halting and later restarts from its latest consistent state.*

A provenance store has been implemented as a stateless web service with a database storage system. We denote as a *crash* all kinds of failures that bring down the provenance store server and cause all data in volatile memory to be lost, but leave all data on stable secondary storage intact.

Assumption 1 states that a provenance store has the ability to restart from its most recent and consistent state, which refers to the initial state of the web service as well as the latest consistent state of the database system. Such a consistent state can be preserved via a recovery procedure following a crash, e.g., restarting the web service

and/or performing fault-tolerant operations (e.g., checkpoint/rollback/rollforward) on the database.

The crash that we assume in Assumption 1 is sometimes referred to as *soft crash* case, since it leaves all data on secondary storage intact, unlike a *hard crash* that corrupts secondary storage media. A hard crash model is concerned with *media failures* [133], such as disk head crash and magnetic decay. The consequence of media failures is the loss of partial or complete process documentation, which cannot be retrieved even after being successfully recorded in provenance stores. Since disk storage technology is already highly reliable and recovery from media failures is not likely to happen more often than once or twice a year [84], it is reasonable to ignore the hard crash case and state the following assumption.

Assumption 2. *Process documentation is persistent once being successfully recorded in provenance stores.*

In real world, applications with different requirements on process documentation can decide if media failures should be considered. For those with less requirement on the durability of process documentation, this type of failures can be ignored given the low probability of occurrence. For industrial-strength applications, where process documentation is critical, recovery from media failures is mainly through two approaches. The traditional one is the combination of periodically creating database backups during normal operations and maintaining archive logs. The logs can be applied to the backup data to redo committed transactions in order to restore the data up to the point of the media failures. Another approach is mirroring disk storage based on techniques such as RAID ("Redundant Array of Independent Disks"). Mission-critical applications usually combine both approaches: backups with archive logging as well as RAID storage technology.

Assumption 3. *Messages to/from provenance stores can be omitted, reordered but not duplicated in communication channels.*

Assumptions 1 and 3 give the failure model with regard to recording process documentation whilst Assumption 2 lays the foundation for our analysis on the retrieval of process documentation in later chapters as it assumes no documentation is lost once being recorded to provenance stores.

Assumption 4. *We do not consider application failures.*

Applications[1] should provide fault tolerance mechanisms to ensure assertors' availability and the completion of an application. Section 2.7 has reviewed some fault-tolerant approaches for grid applications.

[1] In the rest of the dissertation, we use the term *application* to denote a provenance-aware application that can create and record process documentation.

Assumption 5. *An assertor has a choice of several provenance stores to use.*

In SOAs, it is convenient to deploy a number of provenance store services for an assertor to use.

Assumption 6. *The update coordinator does not fail.*

As will be detailed later, a coordinator is used to facilitate viewlink update in order to enable documentation retrievability. We can use a coordinator cluster to ensure its availability for two reasons. First, a coordinator only maintains a minimum amount of information, therefore the cost for preserving consistent states in all replicas can be small. Second, a coordinator is not involved in every interaction of a process[2], which means a single coordinator can support all applications in the system.

However, we do not assume the use of a provenance store cluster. First, it does not deal with message loss and hence cannot prevent disconnected process documentation. Second, replication is sophisticated and comes with a significant cost due to maintain consistent states among replicas. Given that the documentation produced in a process can be in large quantity, e.g., on the order of terabytes, replication may become very expensive in terms of computing resource utilisation and performance impact. Therefore, a provenance store cluster may not be appropriate in all situations. Third, we are dealing with SOA-like systems where there may be any number of provenance stores managed by different institutions across different regions, so it is unrealistic to assume each is facilitated with replicated backups.

3.3 Requirements

F-PReP is defined based on an *interaction*. It specifies the behaviour of the sender and receiver in terms of creating and recording their respective interaction record about an interaction. It also defines the actions of a provenance store and the coordinator in terms of dealing with messages (e.g., an assertor's interaction record) with regard to the interaction.

In order to establish our thesis statement, we identify several requirements for F-PReP to support. Given that a process consists of a set of interactions, if the protocol can ensure that the following requirements are supported on interaction level, then the documentation of the whole process is shown to meet these requirements. Chapter 4 will formalise these requirements as properties and prove that F-PReP preserves these properties. Chapter 5 will use these properties as building blocks to establish the properties of whole process documentation.

[2] A coordinator is only used when an assertor wants to update a link in a provenance store after failures occurred.

The most important requirement is to ensure the successful recording of an interaction record in the presence of failures.

Requirement 1 (Guaranteed Recording)**.** *An assertor's interaction record must be eventually recorded in a provenance store.*

Multiple provenance stores are connected by a chain of pointers (i.e., links) to enable the retrieval of distributed documentation. Thus, we have the following two requirements.

Requirement 2 (Causelink Accuracy)**.** *Accurate causelink(s) regarding an effect interaction must eventually exist in a provenance store. Each must point to the store where an interaction record about the corresponding cause interaction was successfully recorded.*

Requirement 3 (Viewlink Accuracy)**.** *Accurate viewlinks regarding an interaction must eventually exist in a provenance store. Each must point to the store where the other assertor in the interaction successfully recorded its interaction record documenting that interaction.*

Creating and recording interaction records have already introduced overhead into the application [78]. Remedial actions coping with failures may however take up computing resources and interfere with the application. Therefore, we identify a nonfunctional requirement:

Requirement 4 (Efficient Recording)**.** *The recording of interaction records should be efficient and introduce acceptable recording overhead on the application's performance.*

3.4 Design Philosophy

In this section, we analyse several problems that may occur when recording interaction records to a provenance store in the event of failures and outline how F-PReP addresses these problems to satisfies the four requirements.

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. Secondly, the protocol may involve the cooperation of several parties such as provenance-aware applications, provenance stores and the coordinator. 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.

PReP does not specify well-defined behaviour when recording documentation in the presence of failures. Firstly, PReP allows a provenance store to return an acknowledgement before persisting an interaction record. This has the risk that the acknowledged

interaction record can be lost in the event of a store crash whilst the assertor is unaware of it. In addition, PReP assumes an assertor always obtains an acknowledgement from a provenance store and hence does not consider the situation where the acknowledgement is lost or the provenance store crashes.

To address the first problem, F-PReP enforces that a provenance store returns an acknowledgement to an assertor only *after* successfully recording the interaction record that is being acknowledged.

Regarding the second problem, we systematically analyse several failure scenarios where an assertor sends an interaction record (IR) to a provenance store (PS) and PS replies the assertor with an acknowledgement (Ack) *after* recording IR. Since messages can get lost (Assumption 3) and a store can crash (Assumption 1), we present the following cases based on the failure assumptions. We discuss the loss of IR and Ack and the situations where a PS can fail.

1. The message IR is lost;
2. PS crashes before completely receiving IR;
3. PS crashes after completely receiving IR, and before successfully recording IR and replying Ack;
4. PS crashes after successfully recording IR and before sending out Ack;
5. PS crashes after sending out Ack;
6. The message Ack is lost.

If a provenance store crashes after providing Ack (Scenario 5) this means it has successfully recorded IR. From an assertor's perspective, all the other scenarios result in failure to receive an acknowledgement from a provenance store.

Without the acknowledgement, an assertor does not know if its IR has been successfully recorded in a provenance store. To avoid waiting for the acknowledgement indefinitely, F-PReP explicitly sets a timeout for an assertor after it sends IR to a store. If the assertor does not receive a response before the timeout, it knows failures *may* have occurred but cannot ascertain if the provenance store has recorded its IR due to the incapability of distinguishing the loss of IR or Ack from store crash. In addition, a low speed network or a provenance store experiencing slowdown can also cause a timeout.

In order to guarantee the successful recording of documentation, an assertor interprets a timeout as "failure has occurred" and takes remedial actions. Several fault-tolerant mechanisms are adopted by F-PReP in the presence of a "failure" (i.e., timeout): retransmitting messages and using alternative provenance stores. An assertor can resend

an IR to a same provenance store or an alternative one until the IR is acknowledged before the timeout. A provenance store has been designed to handle duplicate retransmitted IRs and always return the same *Ack* for a specific IR. The use of alternative provenance stores is a general approach to tolerating provenance store crashes without assuming any provenance store cluster, although the latter can be complementary. Guaranteed Recording is met when IR is acknowledged before a timeout.

It may be the case where there is an infinite series of crashes and/or channel omissions, resulting in the impossibility of receiving any acknowledgement. However, we do not consider this case since such a case indicates a fundamental problem with the entire system rather with the recording of process documentation. Since a provenance store is able to recover from failures (Assumption 1), we make another assumption stating that an acknowledgement is "eventually" received by an assertor given the presence of an appropriate timeout value as well as the use of message retransmissions.

Assumption 7. *Within the set of provenance stores that an assertor decides to use, at least one acknowledgement is received by the assertor before a timeout.*

Since provenance stores hosting distributed process documentation have been interlinked using a pointer chain, one challenge F-PReP has to face is the potential breakage of the pointer chain due to the use of alternative provenance stores. This means a link to the original store becomes invalid if a specific interaction record is not recorded in the original store. We now discuss causelinks and viewlinks using the examples in Section 3.1.

In Figure 3.3, if the assertor fails to record $IR1$ to $PS1$, it uses an alternative store $PS1'$, which successfully records $IR1$. Since the recording of interaction records is asynchronous to the execution of the application, the use of alternative stores is not known by the assertor when the causelink to $PS1$ is embedded in the relationship p-assertion. Therefore, the causelink becomes inaccurate as it points to a location where $IR1$ cannot be found.

In the case of viewlinks, since one or two assertors may use alternative stores, we discuss two situations. In Figure 3.4, the sender uses another store $PS1$ to record $IR1$, hence making the receiver's viewlink in $PS2$ inaccurate, which still points to the sender's original store $PS1$. Figure 3.5 shows the situation where both assertors use alternative stores to record their interaction record.

The presence of inaccurate links results in distributed process documentation unable to be retrieved from multiple provenance stores. Therefore, F-PReP must provide actions to fix inaccurate causelinks and viewlinks, making them point to the correct location.

To satisfy Causelink Accuracy, F-PReP specifies the following actions. Firstly, all IRs created by the same assertor are placed into a local queue before being recorded to

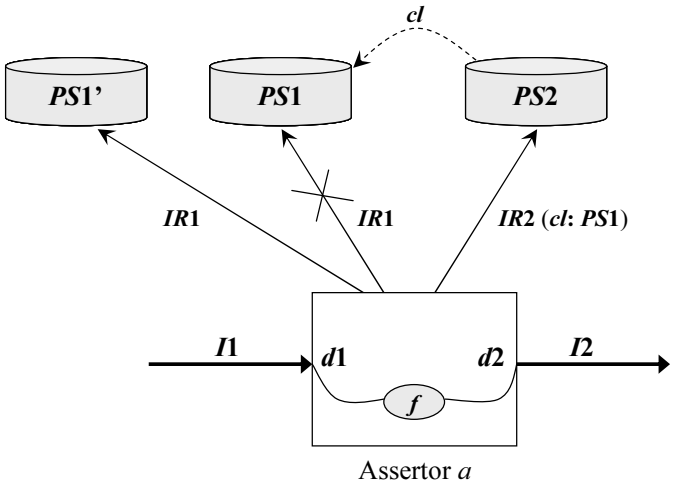


FIGURE 3.3: An example of inaccurate causelink

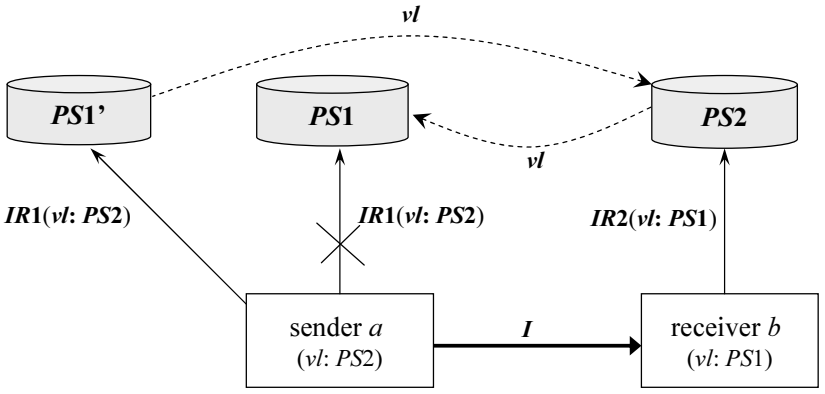


FIGURE 3.4: An example of inaccurate viewlink

a provenance store. Only after the IR at the head of the queue has been successfully recorded can it be removed from the queue. Secondly, given a relationship p-assertion, the IR about any cause interaction is always enqueued before the IR about the effect interaction. Thirdly, an assertor maintains history information in a log table about the use of alternative stores when recording an IR.

The use of a queue has two purposes. Firstly, it allows the recording of IRs to be asynchronous to application execution, which improves recording performance. Secondly, it serialises the recording of IRs about the effect interaction and cause interaction(s). Therefore, when the assertor is ready to submit the IR about the effect interaction, which is the head of the queue, it knows if an alternative store was used when recording the IR about the cause interaction by checking the log table. Then the assertor can update any incorrect causelink according to the history information in the log table.

In the example of Figure 3.6, an alternative store $PS1'$ was used to record $IR1$, which is about the cause interaction $I1$. The log table now reflects that in interaction $I1$ where the assertor was a receiver (R), an alternative store $PS1'$ was used to record $IR1$. When $IR2$ is to be submitted, the assertor checks all relationship p-assertions in $IR2$ according

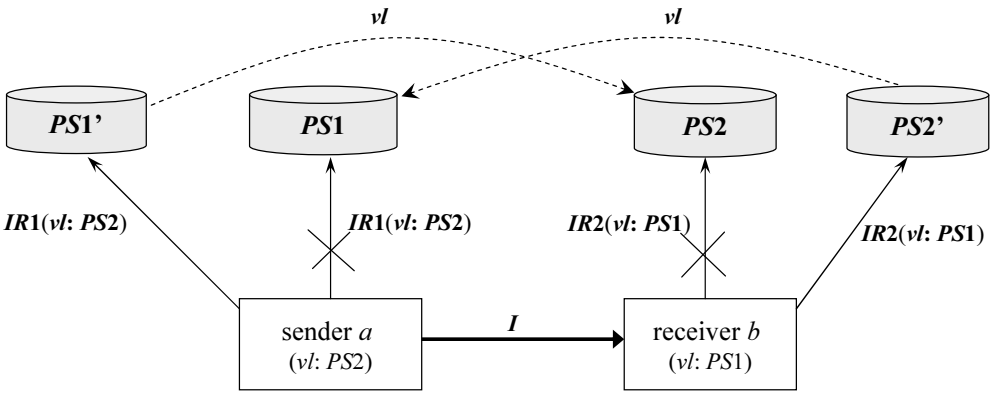


FIGURE 3.5: An another example of inaccurate viewlink

to the log table. If there is an entry regarding a cause interaction in the table, then the assertor uses the alternative store's identifier to update the corresponding causelink. In the figure, the causelink is redirected to $PS1'$, which is the store that successfully recorded $IR1$.

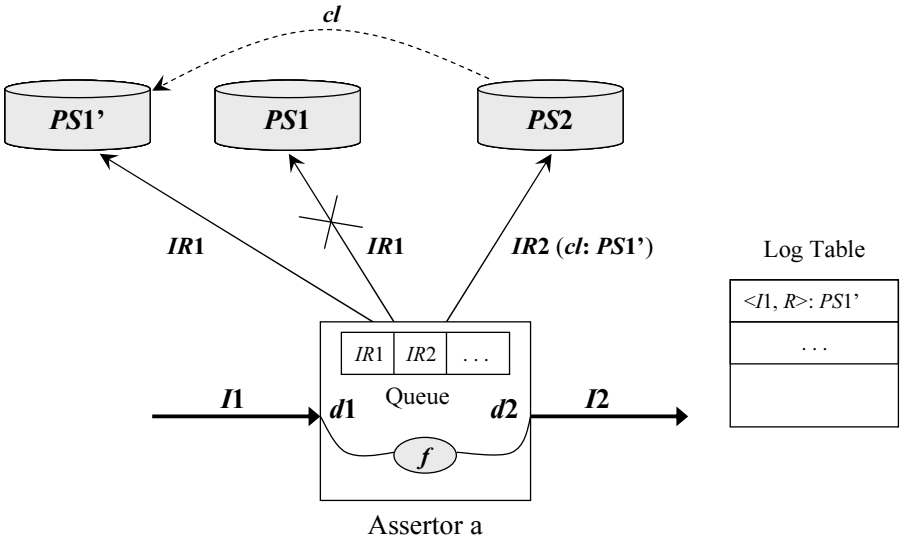


FIGURE 3.6: Causelink updated

To achieve VIEWLINK ACCURACY, an update coordinator is employed to facilitate viewlink update. F-PReP specifies that any assertor that successfully recorded its IR to an alternative store takes the initiative to update the other's viewlink by sending a request to the coordinator.

In the case of Figure 3.4, after recording $IR1$ in $PS1'$, the sender knows that the receiver's viewlink must have been out-dated. Hence it requests the coordinator to update $PS2$, which the sender assumes the receiver is still using. Then the coordinator sends an update request to $PS2$, making the receiver's viewlink point to $PS1'$. Therefore, the viewlink in $PS2$ becomes accurate.

If the receiver also used an alternative store, as illustrated in Figure 3.5, the coordinator's

update is not successful as $PS2$ did not record $IR2$[3]. However, since the receiver also sends a request to the coordinator for updating the sender's viewlink, the coordinator now possesses global knowledge from both sides. Then it makes another decision on how to update two assertors' viewlink: it updates the viewlink in $PS1'$ by redirecting it to $PS2'$ and updates the viewlink in $PS2'$ by redirecting it to $PS1'$. Therefore, both viewlinks become accurate (Figure 3.7). We will detail the coordinator's behaviour in Section 3.5.2.

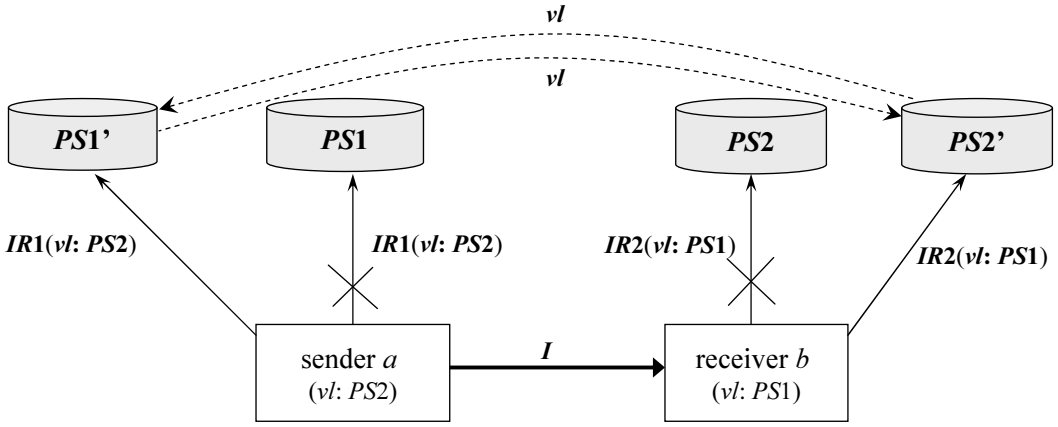


FIGURE 3.7: Viewlink updated

To meet EFFICIENT RECORDING, F-PReP is designed with a number of considerations. Firstly, it is an asynchronous protocol, allowing assertors to send IRs at any time without delaying their execution. In addition, remedial actions, e.g., selecting alternative stores, are taken by the protocol irrespective of the application. We will introduce more implementation considerations to achieve good recording performance in Chapter 6.

After a brief description of F-PReP, next section will detail the protocol and provide more justifications about design decisions.

3.5 Protocol Description

3.5.1 Definitions

To facilitate the description of our protocol, we define three new terms:

Definition 8 (OWNLINK). *It refers to the provenance store that an assertor is currently using to record an interaction record.*

Definition 9 (DEFAULT LINK). *It refers to the provenance store that an assertor initially used when recording an interaction record but may not have been successful in doing so.*

[3]It may be the case that $PS2$ happens to record $IR2$, and the receiver experiences a timeout and uses an alternative store. In this case, $PS2$ records duplicate information, which we will discuss in Section 3.7.

Definition 10 (Default Store)**.** *If a provenance store is referred by an assertor's default link, then it is the assertor's default store.*

We note that an actor is free to use a different provenance store when recording another interaction record. So the default link may not be the same in each interaction.

In the example of Figure 3.8, we assume assertor a initially used $PS1$ to record its documentation IR but failed. Then it used another store $PS2$ to record IR. In this case, a's default link refers to $PS1$, which is its default store. Its own link is changed from $PS1$ to $PS2$ since $PS2$ is currently being used.

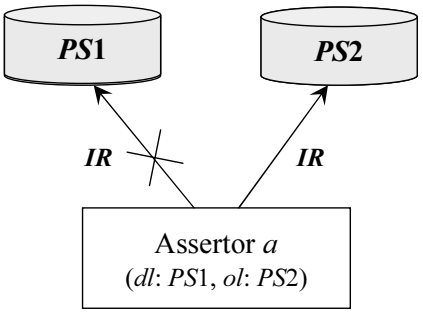


Figure 3.8: An Example of ownlink and default link

3.5.2 Messages

F-PReP specifies the behaviour and communications of assertors, provenance stores and the update coordinator. 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 3.9, which provides an example of message exchanges.

3.5.2.1 Application Message

The application message app is exchanged by all assertors in the application. 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 interaction key to generate and record p-assertions about the same interaction.

In Figure 3.9, we assume that the key for the interaction in which 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 sends an app to b

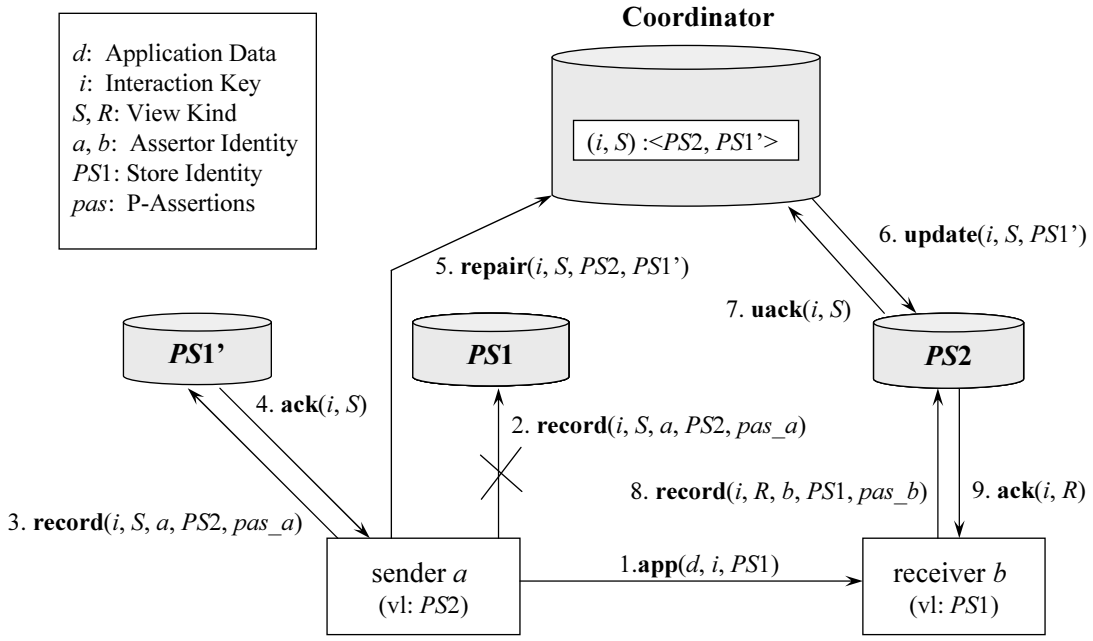


FIGURE 3.9: Protocol message exchanges

containing application data d, interaction key i and a's ownlink to $PS1$. 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, transferred to a in a response message, or in an extra message from b.

3.5.2.2 Interaction Record Message

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

In Figure 3.9, a and b create a set of p-assertions, pas_a and pas_b, respectively, about the interaction i. They send pas_a and pas_b in record messages with their respective viewlink to PS2 and PS1 (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 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 assertor accumulates record messages in a local queue and submits them to their respective provenance store at its most convenient time. When a record message becomes the head of the queue, the assertor checks all the relationship p-assertions in the message and updates inaccurate causelinks before submitting it to its provenance store. These actions are detailed in Section 3.6.3.

3.5.2.3 Record Ack Message

A provenance store acknowledges a 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.

An assertor sets a timeout when waiting for an ack immediately after it sends a record to a provenance store. This helps the assertor take remedial actions without waiting too long. If an ack is not received before the timeout, then the assertor resends the same record to the assertor's default store or an alternative store. Only after receiving an ack acknowledging a record can the assertor 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.

An assertor keeps history information of using alternative stores in a log table in order to facilitate causelink update. It always places the record about a cause interaction into the queue before placing the record about the effect interaction. The FIFO nature of the queue ensures that when the record about the effect interaction is ready to be submitted from the queue, the assertor knows if any alternative store was used for submitting the record about the cause interaction. Therefore, by checking the log table, the assertor can update any incorrect causelink in the record about the effect interaction.

In Figure 3.9, we assume 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). In Section 3.5.3, we will discuss the case where both assertors use an alternative store.

3.5.2.4 Repair Message

After an assertor records a record to an alternative store, the other assertor's viewlink points to an incorrect store. F-PReP specifies that any assertor that successfully recorded its IR to an alternative store takes the initiative to update the other's viewlink by sending a repair request to the coordinator.

In Figure 3.9, 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 the application, 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 viewlink recorded in $PS2$, which the sender thinks the receiver is still using.

A repair message consists of four elements: (1) an interaction key, indicating in which interaction the opposite assertor's viewlink is to be updated; (2) the requesting assertor's view kind in the interaction; (3) a pointer ($DestPS$) to the provenance store where the requesting assertor thinks the opposite assertor's viewlink is recorded; (4) the requesting assertor's ownlink, which points to the provenance store from which the requesting assertor received an ack for that interaction, indicating the store has recorded the assertor's interaction record.

A coordinator is necessary since both sender and receiver may issue a repair request in an interaction (Figure 3.5). This cannot be achieved by a direct update of the other assertor's provenance store, because at that moment, an assertor does not know which store the opposite assertor is actually using. In Figure 3.5, the receiver uses an alternative store to record its record; hence the sender's viewlink to $PS2$ becomes incorrect as well. In that case, the receiver needs to issue another repair request to the coordinator. We will detail this case in Section 3.5.3.

In order to deal with the case where both assertors in an interaction each issue a repair request, which could be in any order, the coordinator maintains request information: the identity of the destination store, specified by the $DestPS$ field in the repair message, and the requesting assertor's ownlink. This request information is indexed by the pair of interaction key and view kind. In Figure 3.9, after receiving a repair request from the sender, the coordinator records a tuple $\langle PS2, PS1' \rangle$ indexed by the pair (i, S), indicating the sender in interaction i sent a repair message.

Since an update coordinator is not involved in every interaction, we recommend that all assertors participating in a process employ one coordinator. If using more than one, then any two assertors exchanging an application message must share the same one in order to ensure Viewlink Accuracy. The identifier of a coordinator can be built in assertors or exchanged to other assertors in the application message app. Figure 3.9 employs the former approach.

By Assumption 3, we do not consider the loss of repair messages in the channel between an assertor and the coordinator[4]. Hence, Assumption 6 implies that a repair request is always processed by the coordinator.

[4]This can be relaxed by using an extra acknowledgement and message retransmissions.

3.5.2.5 Update Message

After receiving a repair message, the coordinator sends a 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, an assertor's viewlink in the store needs to be updated; (2) the view kind of the requesting assertor that issued a repair request for that interaction; (3) the ownlink of the requesting assertor. The $DestPS$ field in the repair message tells the coordinator where to send the update message.

In Figure 3.9, the coordinator sends to $PS2$ an update message containing the sender's ownlink to $PS1'$ (Step 6). Therefore, the receiver's viewlink in $PS2$ is replaced with $PS1'$ and hence becomes accurate.

If the coordinator receives two repair messages each from one assertor in an interaction, then it sends out two update messages after performing operations using the maintained request information to ensure that both update messages are delivered to correct destination stores. We will detail this case in Section 3.5.3.

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 requirement *Viewlink Accuracy*.

3.5.2.6 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 coordinator acknowledging the respective update message. Since messages update or uack may be lost in channel and a destination store may crash, the coordinator sets a timeout when waiting for a uack. A timeout event leads the coordinator to resending the update message to the same provenance store. Given that a store eventually recovers (Assumptions 1), retransmission of update eventually results in the destination store being updated.

3.5.3 Dealing with Two Repair Messages

The coordinator needs to deal with the case where two repair messages are received regarding the same interaction. We now use Figure 3.10 to illustrate this. To simplify our presentation, we only show message parameters of repair and update as the two messages are relevant to the coordinator. We also omit messages uack in the figure.

In Figure 3.10, both assertors used alternative stores $PS1'$ and $PS2'$ to record their respective interaction record (Steps 3 and 8). Hence, each sent one repair message to the coordinator (Steps 5 and 10). When receiving one repair, the coordinator firstly checks

if it has a record from the other assertor of the given interaction. If not, it keeps a record of the destination store and an assertor's ownlink, and sends an update (Step 6) to the destination store provided the destination is available. However, in this figure, the coordinator's update is not successful as $PS2$ did not record the receiver's viewlink due to failure.

Later, when the coordinator receives another repair from the other assertor (Step 10), it now possesses global knowledge from both sides. It replaces one view's destination store with another view's ownlink, making each view's destination store correct (Step 11). After the replacement, the destination store of each tuple becomes $PS2'$ or $PS1'$, which is the ownlink in the other tuple (Figure 3.10). Then the coordinator dispatches two update messages to their new destination (Steps 12 and 13): it updates the viewlink in $PS1'$ by redirecting it to $PS2'$ and updates the viewlink in $PS2'$ by redirecting it to $PS1'$. Therefore, both viewlinks in $PS1'$ and $PS2'$ become accurate (as has been shown in Figure 3.7).

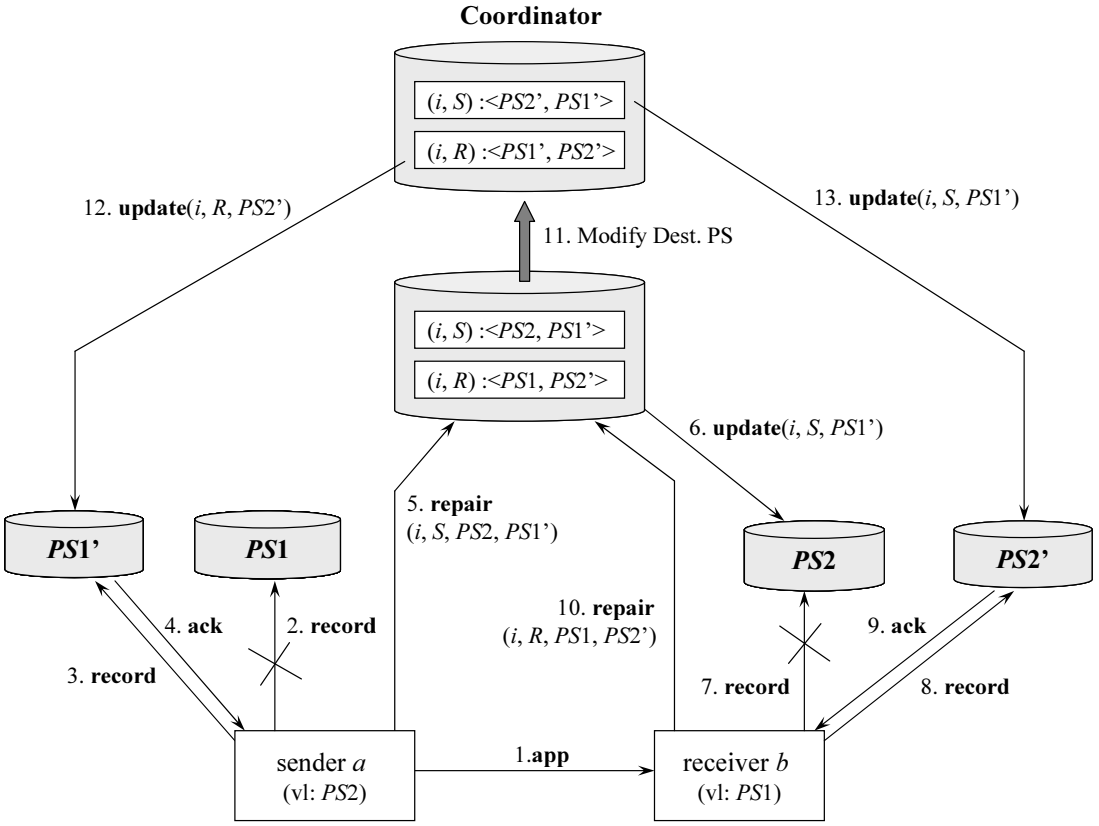


FIGURE 3.10: The coordinator receiving two repair requests

3.5.4 Discussion

Two factors affect the accuracy of a causelink or viewlink:

Firstly, the asynchronous nature of the protocol allows an assertor to record documen-

tation in parallel with the application's execution. When an assertor is creating a relationship p-assertion embedding causelinks or is sending an application message with its ownlink (Step 1 in Figure 3.9), it uses the default link of the corresponding interaction. However, the default link may not point to the store which will successfully record the assertor's interaction record later.

Secondly, an assertor can use an alternative store to record its interaction record in the presence of failures (i.e., timeout events). Similarly to the coordinator retransmitting message update to a store, an assertor can keep resending record messages to a same provenance store without using alternative stores. But an assertor is not expected to do so since it may affect application performance if a crashed provenance store recovers after a long period of time. Hence, using an alternative store is one of our efforts to meet requirement *Efficient Recording*. The coordinator, on the other hand, does not affect the application when updating a viewlink in a provenance store. So it is acceptable that the update is delivered to a store after a while.

3.6 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 [122], a fault-tolerant directory service for mobile agents [120] and PReP [76].

The ASM characterises the behaviour of a distributed system consisting of assertors, provenance stores and the coordinator with respect to the messages the subcomponents 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. It is also systematic and can easily be encoded in a mechanical prover (as illustrated by other proofs [119, 123, 121] successfully encoded in Coq [36]).

Our approach to formalising F-PReP follows two steps. Firstly, we model F-PReP's behaviour in a failure-free environment (without considering Assumptions 1 and 3) in Sections 3.6.1 to 3.6.5. Secondly, we discuss failures by refining the failure-free specification in Section 3.6.6.

In Section 3.6.1, we begin by describing the state space of the ASM, and then proceed to discuss its transitions. Sections 3.6.2, 3.6.3, 3.6.4 and 3.6.5 detail the behaviour of each kind of actors. In Section 3.6.6, we consider failures by extending the system state space and adding transitions to the ASM. Chapter 6 will introduce how we implement the protocol based on the formalisation.

3.6.1 System State Space

Figure 3.11 shows the system state space in a failure-free environment. We identify specific subsets of actors in the system, namely, the assertors (senders and receivers), provenance stores, and update coordinators. In the rest of this dissertation, we assume *a single coordinator is used in the system (*$|\text{COID}| = 1$*) and it is known by all assertors.* We will evaluate if one coordinator leads to a performance bottleneck in Chapter 6.

The set of each protocol message 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 sets 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. The power set notation ($\mathbb{P}$) denotes that there can be more than one of a given element. Messages are exchanged over a set of communication channels, $\mathcal{K}$. Since no assumption is made about message order in communication channels and the fact that message retransmissions can lead to duplicate messages in communication channels, $\mathcal{K}$ is represented as bags[5] of messages between pairs of actors.

To ensure the uniqueness of an interaction key, we use the sender and receiver's identifiers and a natural number to model the interaction key IK. Since the interaction key is created by the sender of an interaction, the sender needs to ensure that the natural number is locally unique on the sender side of each interaction. In terms of VK, we use S and R to denote the sender and receiver's viewkind, respectively.

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

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

[5] In mathematics, a bag is a generalisation of a set. A member of a bag can have more than one membership, whilst each member of a set has only one membership.

$$
\begin{array}{rcll}
\mathrm{A} & = & \{a_1, \ldots, a_n\} & \text{(Set of Actor Identities)} \\
\mathrm{AID} & \subseteq & \mathrm{A} & \text{(Assertor Identities)} \\
\mathrm{SID} & \subseteq & \mathrm{AID} & \text{(Sender Identities)} \\
\mathrm{RID} & \subseteq & \mathrm{AID} & \text{(Receiver Identities)} \\
\mathrm{PID} & \subseteq & \mathrm{A} & \text{(Provenance Store Identities)} \\
\mathrm{COID} & \subseteq & \mathrm{A} & \text{(Coordinator Identities)} \\
|\mathrm{COID}| & = & 1 & \\
 & & & \\
\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{AID} \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} & = & \mathrm{A} \times \mathrm{A} \to \mathit{Bag}(\mathcal{M}) & \text{(Set of Channels)} \\
 & & & \\
\mathrm{IR} & = & \{m \in \mathcal{M} \mid m = \mathsf{record}(\kappa, v, a, vl, pas)\} & \text{(Set of Interaction Records)} \\
\mathrm{IK} & = & \mathrm{SID} \times \mathrm{RID} \times \mathbb{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{(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{ASSERTOR} & = & \mathrm{AID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{STR}_\perp \times \mathrm{DL}_\perp \times \mathrm{OL}_\perp \times \mathrm{VL}_\perp \times \mathbb{P}(\mathrm{PA}) & \text{(Set of Assertors)} \\
\mathrm{STR} & = & \{\mathsf{INIT}, \mathsf{READY}, \mathsf{SEND}, \mathsf{SENT}, \mathsf{ACKED}, \mathsf{OK}\} & \text{(States of Interaction Record)} \\
\mathrm{DL} & = & \mathrm{PID} & \text{(Set of Defaultlinks)} \\
\mathrm{LOG} & = & \mathrm{AID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{CHANGED}_\perp \times \mathrm{APS}_\perp & \text{(Set of Log Tables)} \\
\mathrm{CHANGED} & = & \mathit{Bool} & \text{(Flags of Using Alt. PS))} \\
\mathrm{APS} & = & \mathrm{PID} & \text{(Set of Used Alt. Stores)} \\
\mathrm{QUEUE} & = & \mathrm{AID} \to \mathit{Queue}(\mathrm{IR}) & \text{(Set of Record Queues)} \\
\mathrm{LC} & = & \mathrm{SID} \to \mathbb{N} & \text{(Sender's Local Counters)} \\
\mathrm{PSLIST} & = & \mathrm{AID} \to \mathbb{P}(\mathrm{PID}) & \text{(Set of Alt. Store Lists)} \\
 & & & \\
\mathrm{TIMER} & = & \mathrm{A} \to \mathrm{IK} \times \mathrm{VK} \times \mathrm{PID} \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{RC} & = & \mathrm{A} \to \mathrm{IK} \times \mathrm{VK} \to \mathbb{N} & \text{(Set of Retry Counters)} \\
 & & & \\
\mathrm{PS} & = & \mathrm{PID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{AID}_\perp \times \mathrm{VL}_\perp \times \mathbb{P}(\mathrm{PA}) & \text{(Set of Provenance Stores)} \\
\mathrm{VLST} & = & \mathrm{PID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{VLS}_\perp & \text{(ViewLink State Tables)} \\
\mathrm{VLS} & = & \{\mathsf{DEFAULT}, \mathsf{UPDATED}\} & \text{(Set of ViewLink States)} \\
 & & & \\
\mathrm{COORD} & = & \mathrm{COID} \to \mathrm{IK} \times \mathrm{VK} \to \mathrm{DESTPS}_\perp \times \mathrm{OL}_\perp & \text{(Set of Coordinators)} \\
\mathrm{UPDATE} & = & \mathrm{COID} \to \mathrm{IK} \times \mathrm{VK} \times \mathrm{PID} \to \mathrm{STATE}_\perp & \text{(Set of Update Tables)} \\
\mathrm{STATE} & = & \{\mathsf{UPDATE}, \mathsf{WAIT}, \mathsf{UPDATED}\} & \text{(Set of Update States)} \\
 & & & \\
\mathrm{C} & = & \mathrm{ASSERTOR} \times \mathrm{LOG} \times \mathrm{QUEUE} \times \mathrm{LC} \times \mathrm{TIMER} \times & \\
 & & \mathrm{RC} \times \mathrm{PS} \times \mathrm{VLST} \times \mathrm{COORD} \times \mathrm{UPDATE} \times \mathcal{K} & \text{(Set of Configurations)}
\end{array}
$$

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

Initial State of Configuration:
$c_i = \langle assertor\_T_i, log\_T_i, queue\_T_i, lc_i, timer\_T_i, rc_i, store\_T_i, vlstate\_T_i, coord\_T_i, update\_T_i, k_i \rangle$
where:

$$
\begin{array}{lll}
assertor\_T_i = \lambda a \kappa v \cdot \langle \perp, \perp, \perp, \perp, \emptyset \rangle, & log\_T_i = \lambda a \kappa v \cdot \langle \perp, \perp \rangle, & queue\_T_i = \lambda a \cdot \emptyset, \\
lc_i = \lambda a \cdot 0, & timer\_T_i = \lambda a \kappa v a_{ps} \cdot \langle \perp, 0 \rangle, & rc_i = \lambda a \kappa v \cdot N, \\
store\_T_i = \lambda a_{ps} \kappa v \cdot \langle \perp, \perp, \emptyset \rangle, & vlstate\_T_i = \lambda a_{ps} \kappa v \cdot \perp, & coord\_T_i = \lambda a_c \kappa v \cdot \langle \perp, \perp \rangle, \\
update\_T_i = \lambda a_c \kappa v a_{ps} \cdot \perp, & k_i = \lambda a a \cdot \emptyset &
\end{array}
$$

FIGURE 3.11: System state space

3.6.1.1 Sender and Receiver State Space

An assertor ($a \in$ AID) uses various tables ($assertor\_T \in$ ASSERTOR, $log\_T \in$ LOG, $queue\_T \in$ QUEUE, $lc \in$ LC, $timer\_T \in$ TIMER and $rc \in$ RC) to record record messages into a provenance store.

A table maps a key to a tuple. Since an assertor may be involved in different interactions where it can be a sender or a receiver, table $assertor\_T$ maps an interaction key ($\kappa \in$ IK) and the assertor's view kind ($v \in$ VK) to a tuple of five elements: the state ($str \in$ STR) of an interaction record message, the assertor's defaultlink ($dl \in$ DL), ownlink ($ol \in$ OL), viewlink ($vl \in$ VL) in that interaction, and the p-assertions created about the interaction ($pas \in \mathbb{P}(\text{PA})$). We use $\text{STR}_\perp$ to denote that the initial state of STR is $\perp$. We note that table $assertor\_T$ is only used for maintaining the assertor's state to facilitate our reasoning and proof; it is not implemented.

We use DL to denote the set of default links. In the example of Figure 3.9, the default link of assertor a and b refers to $PS1$ and $PS2$, respectively. Hence, $PS1$ and $PS2$ are the assertors' default stores in that interaction.

A Log table ($log\_T \in$ LOG) maintains information about an assertor's use of alternative stores to facilitate causelink update. A flag ($changed \in$ CHANGED) is set to TRUE if an alternative store was used. The identifier of the used alternative store is kept in a field ($aps \in$ APS).

After creating interaction records, an assertor accumulates them in a local queue ($queue\_T \in$ QUEUE) before shipping them to their respective provenance store. The FIFO property of the queue guarantees successful causelink update. The notation LC defines a function mapping a sender's identifier to a natural number, used to distinguish interactions which may occur between the same sender and receiver. The sender needs to ensure that the natural number is locally unique. The list of alternative provenance stores are modelled by table $psList \in$ PSLIST, mapping an assertor's identity to a set of store identities.

The timer table ($timer\_T \in$ TIMER) models the timer, which is set by an assertor and the coordinator when waiting for an acknowledgement from a provenance store regarding a specific interaction. The key used by a timer table includes a store's identity ($a_{ps} \in PID$) as well as an interaction key ($\kappa \in$ IK) and a view kind ($v \in$ VK). 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 keeps decreasing after the timer is enabled.

In order to prevent infinitely resending messages, an application usually limits the number of retry attempts. We formalise this with a retry counter ($rc \in$ RC), which has an initial value indicating the max number of resubmissions. The counter decreases by one

after the assertor attempts a retry.

3.6.1.2 PS and Coordinator State Space

The set PS models provenance stores, each containing a table ($store\_T \in$ PS) indexed by a provenance store's identity. The table maps an interaction key and the view kind of the assertor that created and recorded p-assertions in the interaction to a tuple: the identity of the assertor, a viewlink and the set of p-assertions documenting the interaction.

Since the viewlink in a provenance store may be updated, we use a table ($vlstate\_T \in$ VLST) to keep the status of a viewlink. If the state is DEFAULT the viewlink is provided by an assertor without being updated by the coordinator. If it is UPDATED the viewlink has been updated by the coordinator. We note that this table is also solely for the purpose of reasoning and proof of the protocol's properties; it is not implemented.

A coordinator maintains repair request information in a table ($coord\_T \in$ COORD) and the states of updating a viewlink in another table ($update\_T \in$ UPDATE). The key used by table $update\_T$ includes a store's identity ($a_{ps} \in PID$), which refers to the destination store to be updated by the coordinator.

We will further detail these tables when describing the rules of provenance store and update coordinator.

3.6.1.3 State Machine Rules

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 in order to achieve another configuration. Figure 3.11 contains the initial state ($c_i \in$ C), which can be summarised as empty channels, empty tables and any 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.

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.

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 table update pseudo-statements. The table update operation puts a message into a table or changes the content of the table. It can assign a value to a field of a table, or assign a tuple to a table. We use notation $table\_T$ to refer to any table in the system state space, formally:

- 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 $table\_T'(\ldots).y := V$.
- If $table\_T(\ldots)$ has one field x, then the expression $table\_T(\ldots) := V$ denotes $table\_T(\ldots).x := V$.
- If $table\_T(\ldots)$ has fields $x_1, \ldots, x_n$ and V = $\langle v_1, \ldots, v_n \rangle$, then the expression $table\_T(\ldots) := V$ denotes for $m = 1, \ldots, n$, $table\_T(\ldots).x_m := v_m$ if $v_m \neq *$.

In the following example, the fourth field of $assertor\_T(a, \kappa, v)$ (i.e., viewlink vl) is not affected when $*$ is present.

$$\text{assertor\_T(a, } \kappa, v) := \langle \mathsf{OK}, PS_1, PS_2, *, pas \rangle \quad \equiv \quad \begin{cases} \text{assertor\_T(a, } \kappa, v).str := \mathsf{OK} \\ \text{assertor\_T(a, } \kappa, v).dl := PS_1 \\ \text{assertor\_T(a, } \kappa, v).ol := PS_2 \\ \text{assertor\_T(a, } \kappa, v).pas := pas \end{cases}$$

To manipulate an assertor's queue, which is used for accumulating interaction records, we define the following operations: $head(queue\_T(a))$, $enqueue(m, queue\_T(a))$, $dequeue(queue\_T(a))$ and $replaceHead(queue\_T(a), m)$.

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

- The expression $replaceHead(queue\_T(a), m)$ denotes $queue\_T(a) := m \,\|tail(queue\_T(a))$, which means the head of queue $queue\_T(a)$ is replaced by m.

We use *send* and *receive* 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. Formally, *send*, *receive* 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[6] $k'(a_1, a_2) = k(a_1, a_2) \oplus \{m\}$ 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 $k'(a_i, a_j) = k(a_i, a_j), \forall (a_i, a_j) \neq (a_1, a_2)$.

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

3.6.2 Assertor Rules in Exchanging phase

An assertor's behaviour can be summarised in two phases Exchanging and Recording, which are described in this section and Section 3.6.3, respectively.

The sender and receiver in an interaction have different rules in the Exchanging phase (Figure 3.12 and Figure 3.13). The sender sends to the receiver an application message app including application data (d), an interaction key (κ) and the sender's ownlink (a_{ps}).

Both actors document the exchange of app by producing a record, which is accumulated in the queue ($queue\_T$). This buffering of interaction records is designed to meet *Efficient Recording*. It reduces the performance penalty upon the application by allowing an actor to send interaction records when convenient.

We note that in order to fire transition *prepare_record*, the sender's viewlink must be equal to the receiver's defaultlink. This has been assumed when we explained Figure 3.9. The viewlink can be built into the sender at deployment time, or transferred to the sender in a response message from the receiver.

In this phase, an assertor also initialises several tables. For example, when an assertor's default link is initialised in transitions *send_app* and *receive_app*, it is equal to the assertor's ownlink.

$send\_app(a_s, a_r, a_{ps}, d, r)$:
//triggered when d, produced by a function
//described by r, is to be sent by a_s to a_r
$\rightarrow$ {
$\kappa \leftarrow newIdentifier(a_s, a_r)$;
$send(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
$pas \leftarrow createPA(a_s, \kappa, d, r)$;
$assertor\_T(a_s, \kappa, S) := \langle \mathsf{INIT}, a_{ps}, a_{ps}, \perp, pas \rangle$;
$log\_T(a_s, \kappa, S) := \langle \mathsf{FALSE}, \perp \rangle$;
}

$prepare\_record(a_s, a_r, \kappa)$:
$assertor\_T(a_s, \kappa, S).str = \mathsf{INIT} \wedge$
$assertor\_T(a_s, \kappa, S).vl = assertor\_T(a_r, \kappa, R).dl$
$\rightarrow$ {
$l \leftarrow assertor\_T(a_s, \kappa, S).vl$;
$pas \leftarrow assertor\_T(a_s, \kappa, S).pas$;
$enqueue(\mathsf{record}(\kappa, S, a_s, l, pas), queue\_T(a_s))$;
$assertor\_T(a_s, \kappa, S).str := \mathsf{READY}$;
}

FIGURE 3.12: The Sender's rules in exchanging phase

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

FIGURE 3.13: The Receiver's rules in exchanging phase

Function $newIdentifier(a_s, a_r)$ creates a globally unique interaction key. This function takes the identities of the sender and the receiver as inputs and returns a new interaction key represented by a tuple, consisting of assertor identities and a locally unique counter. Since it is the sender that creates a new interaction key, the local counter is maintained by the sender of an interaction.

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

Function $createPA(a, \kappa, d, r)$ specifies how p-assertions are produced. It takes an assertor identity (a), an interaction key (κ), application data (d), and a business logic description (r) as input and returns a set of p-assertions documenting the interaction (κ) in which d is transferred.

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

Definition
$createPA : \mathrm{A} \times \mathrm{IK} \times \mathrm{DATA} \times \mathrm{REL} \rightarrow \mathbb{P}(\mathrm{PA})$
$createPA(a, \kappa, d, r)$:
$pas \leftarrow$ *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 $cids = \{\langle cl, \kappa', R \rangle \mid \kappa' \in cause(a, \kappa, r)$ *and* $cl = assertor\_T(a, \kappa', R).ol\}$;
return pas;

In $createPA(a, \kappa, d, r)$, the created p-assertions 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 according to transition *send_app*, the sender must make a relationship p-assertion to capture the causal connections between these messages. A function $cause(a, \kappa, r)$ is used to find the interaction keys of cause interactions when creating a relationship p-assertion. An assertor may create other application dependent p-assertions, which are not shown in the definition.

Definition
$cause : \mathrm{A} \times \mathrm{IK} \times \mathrm{REL} \rightarrow \mathbb{P}(\mathrm{IK})$
$cause(a, \kappa, r)$:
$causeIK \subseteq \mathrm{IK}$;
//This function is application specific.
//Let κ *represent the effect interaction and* $causeIK$ *represent the set of cause*
//interactions, which are all related to a same relationship described by r *at*
//assertor a.
return $causeIK$;

3.6.3 Assertor Rules in Recording phase

In Recording phase, an assertor sends queued record messages to a provenance store and takes remedial actions in response to timeouts. To facilitate presentation, we assume each assertor employs a Recording Manager (RM), which monitors the assertor's queue and submits record messages to a provenance store. The behaviour of RM is specified in Figure 3.14.

- *Updating causelinks.* Transition *pre_check* checks the record message at the head of queue ($queue\_T(a)$) and updates causelinks in all relationship p-assertions included in that message. 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 a record message about a cause interaction

$pre\_check(a, \kappa, v, vl, pas)$:
$\quad queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge assertor\_T(a, \kappa, v).str = \mathsf{READY}$
$\rightarrow \{$
1 $\quad$ for *each* $pa \in pas$, *such that* $pa = \mathsf{rel\text{-}pa}(r', \langle \kappa, v \rangle, cids')$
2 $\quad\quad$ do for *each* $cid \in cids'$
3 $\quad\quad\quad$ do $\langle cl', \kappa', v' \rangle \leftarrow cid$;
4 $\quad\quad\quad\quad$ *if* $(log\_T(a, \kappa', v').changed)$, *then*
5 $\quad\quad\quad\quad\quad$ $cid' \leftarrow \langle log\_T(a, \kappa', v').aps, \kappa', v' \rangle$;
6 $\quad\quad\quad\quad\quad$ $cids' := cids' \ominus \{cid\} \oplus \{cid'\}$;
7 $\quad replaceHead(queue\_T(a), \mathsf{record}(\kappa, v, a, vl, pas))$;
8 $\quad assertor\_T(a, \kappa, v) := \langle \mathsf{SEND}, *, *, *, pas \rangle$;
$\quad \}$

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

$timer\_click(a, \kappa, v, a_{ps})$:
$\quad timer\_T(a, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa, v, a_{ps}).to > 0$
$\rightarrow \{$
$\quad\quad timer\_T(a, \kappa, v, a_{ps}).to := timer\_T(a, \kappa, v, a_{ps}).to - 1$;
$\quad \}$

$timeout\_ack(a, \kappa, v, a_{ps})$:
$\quad timer\_T(a, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa, v, a_{ps}).to = 0 \wedge rc(a, \kappa, v) > 0$
$\rightarrow \{$
$\quad\quad timer\_T(a, \kappa, v, a_{ps}) := \langle \mathsf{DISABLED}, 0 \rangle$;
$\quad\quad rc(a, \kappa, v) := rc(a, \kappa, v) - 1$;
$\quad\quad a'_{ps} \leftarrow random(psList(a))$;
$\quad\quad log\_T(a, \kappa, v) := \langle \mathsf{TRUE}, a'_{ps} \rangle$;
$\quad\quad assertor\_T(a, \kappa, v) := \langle \mathsf{SEND}, *, a'_{ps}, *, * \rangle$;
$\quad \}$

$receive\_ack(a, a_{ps}, \kappa, v)$:
$\quad \mathsf{ack}(\kappa, v) \in k(a_{ps}, a)$
$\rightarrow \{$
$\quad\quad receive(\mathsf{ack}(\kappa, v), a_{ps}, a)$;
$\quad\quad$ *if* $(timer\_T(a, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa, v, a_{ps}).to > 0)$, *then*
$\quad\quad\quad dequeue(queue\_T(a))$;
$\quad\quad\quad timer\_T(a, \kappa, v, a_{ps}) := \langle \mathsf{DISABLED}, 0 \rangle$;
$\quad\quad\quad assertor\_T(a, \kappa, v).str := \mathsf{ACKED}$;
$\quad \}$

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

FIGURE 3.14: Assertor's rules in recording phase

(i.e., $log\_T(a, \kappa', v').changed$ is true), then the corresponding causelink is updated (Line 6). Since the set of p-assertions in the record message may be altered with causelink updated, we need to update the current set of p-assertions maintained in $assertor\_T(a, \kappa, v)$ (Line 8) as well as the head of the queue (Line 7).

- *Submitting a record message.* RM sends a record message to a provenance store and sets timeout when waiting for an ack message (transition *send_record*).

- *Resubmitting a record message.* If RM does not receive an ack when the timeout expires (transition *timeout_ack*), then it infers that failures may have occurred. In this case, it uses an alternative store to resend the message. 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. A retry counter rc has been introduced to limit the number of reattempts. We will detail the use of the counter when we model failures in Section 3.6.6.

- *Updating log table.* If an alternative store was used to record a record message, RM sets $log\_T(a, \kappa, v).changed$ to TRUE and $log\_T(a, \kappa, v).aps$ to the identity of the alternative store (transition *timeout_ack*). This information is to be used for updating causelinks as described above.

- *Receiving acknowledgement.* If the timeout has not expired upon the receipt of ack message, then RM eliminates the acknowledged record from the queue (transition *receive_ack*).

- *Requesting to update viewlinks.* If an alternative store was used, RM needs to request the coordinator to update the opposite assertor's viewlink by sending a repair message (transition *post_check*).

We note that the FIFO property of the queue guarantees successful update of causelinks because transitions *send_app* and *receive_app* enforce that an assertor always makes p-assertions about a cause interaction, in which it receives a message, before an effect interaction, in which it sends another message as consequence of the received messages. This implies that the interaction record about a cause interaction is always placed into the queue before that about the effect interaction. Therefore, by checking log table log_T before sending record messages, causelinks can be updated successfully. Although current model implies that there is only one queue per assertor, it can be relaxed by adding a process identifier to $queue\_T(a)$. Then, each process that an assertor participates in utilises a queue to permit parallel recording, though sequential recording still remains for each process.

3.6.4 Provenance Store Rules

Figure 3.15 gives provenance store's rules. A provenance store replies an ack message only *after* it has processed a record message (transition *receive_record*). A store checks if it has recorded p-assertions about a given interaction before processing the record message. This prevents processing duplicate record messages.

The notation $\overline{v}$ in transition *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.

We note that table $vlstate\_T(a_{ps}, \kappa, v)$, indicating the state of a viewlink: DEFAULT or UPDATED, is only used to facilitate our proof in the next chapter.

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

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

FIGURE 3.15: Provenance store's rules

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

3.6.5 Coordinator Rules

Coordinator's rules are shown in Figure 3.16. Transition *receive_repair* is triggered when the coordinator receives a repair request. If there exists request information from the opposite view with regard to the same interaction, this means the coordinator has received another repair message (detailed in Section 3.5.3). In this case, the coordinator replaces one assertor's destination store with the other's ownlink (Lines 6 and 7 in transition *receive_repair*), thus making each assertor's destination store correct. Then the

coordinator is ready to dispatch two update messages to their respective new destination stores by setting update status to UPDATE.

$receive\_repair(a, a_{dps}, a_c, \kappa, v, \overline{v}, ol)$:
  $\mathsf{repair}(\kappa, v, a_{dps}, ol) \in k(a, a_c)$
$\rightarrow$ {
1 $receive(\mathsf{repair}(\kappa, v, a_{dps}, ol), a, a_c)$;
2 *if* $(coord\_T(a_c, \kappa, v) = \langle \bot, \bot \rangle)$, *then*
3 $coord\_T(a_c, \kappa, v) := \langle a_{dps}, ol \rangle$;
4 *if* $(coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle)$, *then*
5 $a'_{dps} \leftarrow coord\_T(a_c, \kappa, \overline{v}).ol$;
6 $coord\_T(a_c, \kappa, v) := \langle a'_{dps}, * \rangle$;
7 $coord\_T(a_c, \kappa, \overline{v}) := \langle ol, * \rangle$;
8 $update\_T(a_c, \kappa, \overline{v}, ol) := \mathsf{UPDATE}$;
9 $a_{ps} \leftarrow coord\_T(a_c, \kappa, v).a_{dps}$;
10 $update\_T(a_c, \kappa, v, a_{ps}) := \mathsf{UPDATE}$;
  }

$send\_update(a_c, \kappa, v, a_{ps}, t)$:
  $update\_T(a_c, \kappa, v, a_{ps}) = \mathsf{UPDATE}$
$\rightarrow$ {
  $ol \leftarrow coord\_T(a_c, \kappa, v).ol$;
  $send(\mathsf{update}(\kappa, v, ol), a_c, a_{ps})$;
  $update\_T(a_c, \kappa, v, a_{ps}) := \mathsf{WAIT}$;
  $timer\_T(a_c, \kappa, v, a_{ps}) := \langle \mathsf{ENABLED}, t \rangle$;
  }

$timer\_click(a_c, \kappa, v, a_{ps})$:
  $timer\_T(a_c, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a_c, \kappa, v, a_{ps}).to > 0$
$\rightarrow$ {
  $timer\_T(a_c, \kappa, v, a_{ps}).to := timer\_T(a_c, \kappa, v, a_{ps}).to - 1$;
  }

$timeout\_uack(a_c, \kappa, v, a_{ps})$:
  $timer\_T(a_c, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a_c, \kappa, v, a_{ps}).to = 0 \wedge rc(a_c, \kappa, v) > 0$
$\rightarrow$ {
  $update\_T(a_c, \kappa, v, a_{ps}) := \mathsf{UPDATE}$;
  $timer\_T(a_c, \kappa, v, a_{ps}) := \langle \mathsf{DISABLED}, 0 \rangle$;
  $rc(a_c, \kappa, v) := rc(a_c, \kappa, v) - 1$;
  }

$receive\_uack(a_{ps}, a_c, \kappa, v)$:
  $\mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c)$
$\rightarrow$ {
  $receive(\mathsf{uack}(\kappa, v), a_{ps}, a_c)$;
  if $(timer\_T(a_c, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a_c, \kappa, v, a_{ps}).to > 0)$, *then*
  $timer\_T(a_c, \kappa, v, a_{ps}) := \langle \mathsf{DISABLED}, 0 \rangle$;
  $update\_T(a_c, \kappa, v, a_{ps}) := \mathsf{UPDATED}$;
  }

FIGURE 3.16: Coordinator's rules

The coordinator sends an update message to a destination store and sets timeout when waiting for the acknowledgement uack (transition *send_update*). The coordinator also uses rc to count the number of retries to avoid infinite message retransmissions (transition *timeout_uack*). We will detail the use of the counter in Section 3.6.6.

In the current design, we do not remove request information maintained in the coordinator. Request information with regard to an interaction can only be eliminated *after* the coordinator has *successfully* updated the provenance store in *each view* of the interaction. If there exists request information about only one view, then the coordinator cannot delete it since it may receive another repair request from the other assertor later.

Given that an assertor sends a repair message within finite time (due to the use of timeouts), the coordinator can remove any request information with corresponding update status being UPDATED after a reasonably long period of time.

3.6.6 Modelling Failures

After formalising the protocol in a failure-free environment, this section considers Assumptions 1 and 3. We extend the system state space (Figure 3.17) and revise or add transitions to the ASM (Figures 3.18 to 3.20.). Section 2.8.1 has introduced the common approach to modelling failures, which inspires us. Our approach is also similar to [121], where an ASM is used to model crash failures and communication omissions.

We note that the protocol formalised in the previous sections can be used for implementation whilst the extended system state space and transitions introduced in this section will never be explicitly implemented; they are just a means to be able to evaluate protocol properties [70, 69], which will be done in the following chapters.

$$
\begin{array}{rcll}
\text{CRASH} & = & \text{PID} \rightarrow \mathit{Bool} & \text{(Set of PS Crash Tables)} \\
\text{LOST} & = & \text{A} \times \text{PID} \rightarrow \mathit{Bag}(\mathcal{M}) & \text{(Set of Message Loss Tables)} \\
\text{FC} & = & \text{IK} \rightarrow \mathbb{N} & \text{(Set of Global Failure Counters)} \\
 & & & \\
\text{C}' & = & \text{C} \times \text{CRASH} \times \text{LOST} \times \text{FC} & \text{(Set of Configurations)}
\end{array}
$$

Characteristic Variables:
$crash\_T \in \text{CRASH}$, $lost \in \text{LOST}$, $fc \in \text{FC}$, $c' \in \text{C}'$

Initial State of Configuration:
$c'_i = \langle c_i, crash\_T_i, lost_i, fc_i \rangle$
where:
$crash\_T_i = \lambda a_{ps} \cdot \mathsf{FALSE}, \quad lost_i = \lambda a a_{ps} \cdot \emptyset, \quad fc_i = \lambda \kappa \cdot 0$

FIGURE 3.17: Extended system state space

In Figure 3.17, we use PS Crash Table ($crash\_T \in \text{CRASH}$) to model the state of a provenance store: crashing (TRUE) or normal (FALSE). The table is initialised as FALSE.

A lost table ($lost \in \text{LOST}$) maintains messages that are omitted by communication channels. Since message retransmissions can lead to duplicate messages, which can also be omitted by channels, we represent the lost table as a bag of messages between a pair of actors. In order to simplify rules, $lost(a_1, a_2)$ is defined to be bidirectional, containing messages that are lost in either $k(a_1, a_2)$ or $k(a_2, a_1)$.

We use a Global Failure Counter ($fc \in \text{FC}$) to limit the number of failures (store crashes and channel omissions) that may occur regarding an interaction. Intuitively, it is realistic to experience a finite number of store crashes and message losses when an assertor is recording an interaction record and when the coordinator is updating a store. This counter is initialised with a natural number by the sender of an interaction (Figure 3.18). It is decreased by 1 when there is a failure event (transitions *crash*, *msg_loss_pstore*

and *msg_loss_channel* in Figures 3.19 and 3.20). This counter is crucial to the proof of the protocol's termination property in the next chapter.

$send\_app(a_s, a_r, a_{ps}, d, r, n)$:
//triggered when d, produced by a function
//described by r, is to be sent by a_s *to* a_r
$\rightarrow$ {
$\kappa \leftarrow newIdentifier(a_s, a_r)$;
$send(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
$pas \leftarrow createPA(a_s, \kappa, d, r)$;
$assertor\_T(a_s, \kappa, S) := \langle \mathsf{INIT}, a_{ps}, a_{ps}, \bot, pas \rangle$;
$log\_T(a_s, \kappa, S) := \langle \mathsf{FALSE}, \bot \rangle$;
$fc(\kappa) = n$;
}

FIGURE 3.18: Extended sender's rules

$receive\_record(a, a_{ps}, \kappa, v, vl, pas)$:
$\mathsf{record}(\kappa, v, a, vl, pas) \in k(a, a_{ps}) \wedge \neg crash\_T(a_{ps})$
$\rightarrow$ {
$receive(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$;
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*
$store\_T(a_{ps}, \kappa, v).vl := vl$;
$vlstate\_T(a_{ps}, \kappa, v) := \mathsf{DEFAULT}$;
$send(\mathsf{ack}(\kappa, v), a_{ps}, a)$;
}

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

$crash(\kappa, a_{ps})$:
$fc(\kappa) > 0 \wedge \neg crash\_T(a_{ps})$
$\rightarrow$ {
$crash\_T(a_{ps}) := \mathsf{TRUE}$;
$fc(\kappa) := fc(\kappa) - 1$;
}

$restart(a_{ps})$:
$crash\_T(a_{ps}) = \mathsf{TRUE}$
$\rightarrow$ {
$crash\_T(a_{ps}) := \mathsf{FALSE}$;
}

FIGURE 3.19: Extended provenance store's rules

With the extended system state space, we revise provenance store rules, as shown in Figure 3.19. A new guard, $\neg crash\_T(a_{ps})$, is added to transitions *receive_record* and *receive_update* to ensure that they are only fired when the store is not crashing. Transitions *crash* and *restart* simulate store crash and restart events, respectively.

We have introduced a retry counter rc, whose value is decreased by 1 after each retry attempt in transitions *timeout_ack* (Figure 3.14) and *timeout_uack* (Figure 3.16). Since current formalisation does not consider the case where the counter reaches 0, we state

$msg\_loss\_pstore(a_{ps}, a, \kappa, m)$:
$fc(\kappa) > 0 \wedge crash\_T(a_{ps}) \wedge m \in k(a, a_{ps})$
$\rightarrow$ {
$k(a, a_{ps}) := k(a, a_{ps}) \ominus \{m\}$;
$lost(a, a_{ps}) := lost(a, a_{ps}) \oplus \{m\}$;
$fc(\kappa) := fc(\kappa) - 1$;
}

$msg\_loss\_channel(a_{ps}, a, \kappa, m)$:
$fc(\kappa) > 0 \wedge (m \in k(a, a_{ps}) \vee m \in k(a_{ps}, a))$
$\rightarrow$ {
if $(m \in k(a, a_{ps}))$, *then*
$k(a, a_{ps}) := k(a, a_{ps}) \ominus \{m\}$;
$lost(a, a_{ps}) := lost(a, a_{ps}) \oplus \{m\}$;
else
$k(a_{ps}, a) := k(a_{ps}, a) \ominus \{m\}$;
$lost(a, a_{ps}) := lost(a, a_{ps}) \oplus \{m\}$;
$fc(\kappa) := fc(\kappa) - 1$;
}

FIGURE 3.20: Communication channel's rules

Assumption 11 in order to preserve the protocol's correctness. Under Assumption 11, transitions *timeout_ack* or *timeout_uack* is always executed whenever there is a timeout event.

Assumption 11. *For any a, a_c, κ and v, $rc(a, \kappa, v)$ and $rc(a_c, \kappa, v)$ are always greater than 0.*

Figure 3.20 specifies rules for communication channels. As far as distributed system modelling is concerned, it is unrealistic to consider that messages in transit on a communication link remain present if the destination of the communication link exhibits a failure. Transition *msg_loss_pstore* models that any message sent to a crashing store is omitted in channel. Transition *msg_loss_channel* specifies that messages are lost in the channel to/from a provenance store. These transitions remove a message from channel, place it in the *lost* table and then decrease Global Failure Counter by 1.

3.7 Discussion

This section discusses several issues regarding the design and formalisation of F-PReP.

Since no system wide clock exists in distributed systems, F-PReP does not assume any global time; all ordering is based on local time, as perceived by a given component (an assertor, a provenance store or the coordinator) in the system. Each can cope with messages received from other sites in any order. In addition, there is an ordering requirement to enable causelink update: an assertor must enqueue the interaction record about a cause interaction before the interaction record about the effect interaction, as explained in Section 3.4.

F-PReP uses timeout to detect potential failures. An assertor selects an alternative store to record its interaction record if an acknowledgement is not received from the original store before a timeout. The presence of a timeout however cannot tell an assertor if the original store has processed a message. This implies that the original store may still receive and record the assertor's interaction record, leading to duplicate viewlinks or causelinks recorded in that store (as exemplified in Figures 3.21, 3.22). This would affect documentation retrievability, which we will discuss in Chapter 5.

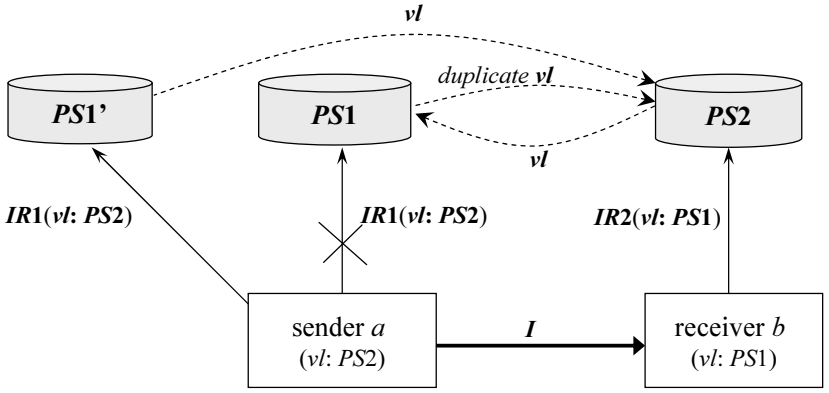


FIGURE 3.21: Duplicate viewlink in $PS1$

The formalisation of a timeout is designed to be an abstract way of detecting a potential failure in order to trigger the remedial actions. In practice, the triggering condition is not necessarily limited to the event of a timeout. For example, it can be a failure to connect to a provenance store and can also be the receipt of a response message with a fault code indicating any exception thrown in the provenance store. The occurrences of these events do not guarantee successful recording of documentation in a provenance store, therefore remedial actions should also be taken.

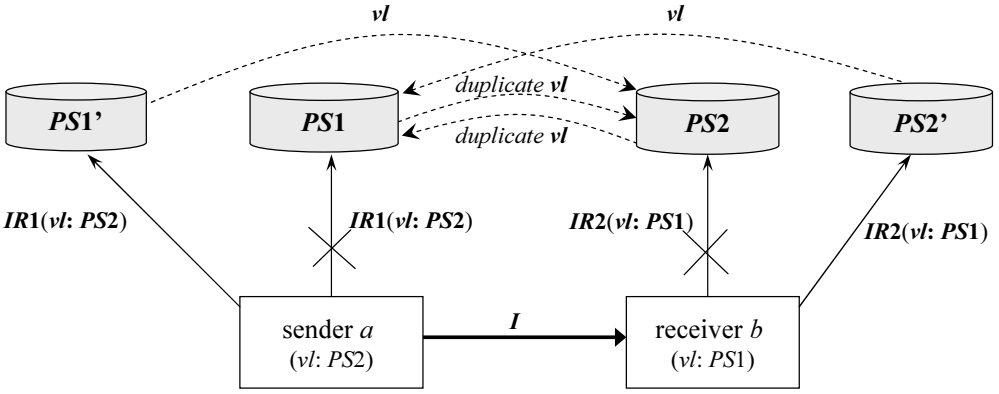


FIGURE 3.22: Duplicate viewlink in $PS1$ and $PS2$

Although an alternative store is always chosen in the presence of a timeout event in the formalisation, the retransmission policy can be configured in practice. For example, an interaction record can be resent to the same provenance store for certain times before using an alternative store.

In the formalisation, we assumed that a retry counter ($rc(a, \kappa, v)$ and $rc(a_c, \kappa, v)$) is always greater than 0 so that transitions *timeout_ack* and *timeout_uack* are always triggered whenever there is a timeout. We now discuss the case where the counter reaches 0. The solution is to employ a local store at the assertor and coordinator's site, which we will detail in Chapter 6. If $rc(a, \kappa, v)$ reaches 0 before the interaction record is successfully recorded in a provenance store, the assertor can checkpoint its recording state by storing all outstanding interaction records to its local store and then resubmit them at a later stage. If $rc(a_c, \kappa, v)$ reaches 0, the coordinator stops resending message update and persistently keeps the request information, i.e., the identity of the destination store and the requesting assertor's ownlink in the coordinator's local store. It can resend the update message later.

F-PReP is based on PASOA's approach to modelling process documentation and it extends PReP by adding fault-tolerant functionalities. The advantages of this add-on approach include separation of concerns and Agile style development, which enable us to separately develop and quickly test the core and fault-tolerant functionalities of a recording protocol. However, we have to be fully compatible with previous design of PReP. Instead, we can design our recording protocol that copes with failures from scratch. This requires an in-depth understanding of both process documentation and complex system behaviour that may appear in the presence of failures, which adds difficulty to our research given strict time constraints.

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 understand the complex behaviour of actors (assertors, provenance stores and the coordinator) in the presence of failures. With such a formal description, we have successfully identified several deficiencies in the early design of the protocol. For example, a deficiency in previous design was that the viewlink provided by an update message can be overwritten by the viewlink from a record message (i.e., we did not check if $store\_T(a_{ps}, \kappa, v).vl$ is $\perp$ in rule *receive_record*). 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.

3.8 Summary

We now summarise the contributions of this chapter:

Firstly, we described a generic protocol, F-PReP, for recording documentation (i.e., interaction records) in the presence of failures (provenance store crashes and communication omission failures). F-PReP also specifies remedial actions when a failure (i.e.,

a timeout event observed by an assertor) occurs. Remedial actions include basic fault-tolerant mechanisms such as message retransmission and the use of alternative provenance stores. F-PReP uses a local queue and log table to enable causelink update. To facilitate viewlink update, a coordinator is introduced to update incorrect viewlinks so that multiple provenance stores are still properly connected.

Secondly, we formalised F-PReP using an ASM approach. We began with a formalisation without consideration of failures. Then we modelled failures by extending system state space and adding extra transitions. With such a formal description, we have successfully identified several deficiencies in the early design of the protocol. Since code-like specification is independent of any given programming language or implementation, our protocol can be implemented using different languages and technologies.

Next chapter will formalise requirements Guaranteed Recording, Causelink Accuracy and Viewlink Accuracy as correctness properties and prove that F-PReP preserves these properties. Chapter 6 will introduce the implementation of F-PReP and evaluate its performance demonstrating that it meets requirement Efficient Recording.

3.9 Appendix: ASM Rules Summary

We have presented ASM rules in Sections 3.6.2, 3.6.3, 3.6.4, and 3.6.5. We also extended the ASM with additional rules in Section 3.6.6. We now summarise the complete ASM rules in Figures 3.23, 3.24, 3.26, 3.27 and 3.28, which will be referred to when we prove the correctness of the protocol in the next chapter. Numbers are annotated in rules, which will be used when we prove the protocol's termination property in Section 4.1.

$send\_app(a_s, a_r, a_{ps}, d, r, n)$:
//triggered when d, produced by a function
//described by r, is to be sent by a_s to a_r
$\rightarrow$ {
$\kappa \leftarrow newIdentifier(a_s, a_r)$;
$send(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
$pas \leftarrow createPA(a_s, \kappa, d, r)$;
$assertor\_T(a_s, \kappa, S) := \langle \mathsf{INIT}, a_{ps}, a_{ps}, \bot, pas \rangle$;
$log\_T(a_s, \kappa, S) := \langle \mathsf{FALSE}, \bot \rangle$;
$fc(\kappa) = n$;
}
annotation not applicable

$prepare\_record(a_s, a_r, \kappa)$:
$assertor\_T(a_s, \kappa, S).str = \mathsf{INIT} \wedge$
$assertor\_T(a_s, \kappa, S).vl = assertor\_T(a_r, \kappa, R).dl$
$\rightarrow$ {
$l \leftarrow assertor\_T(a_s, \kappa, S).vl$;
$pas \leftarrow assertor\_T(a_s, \kappa, S).pas$;
$enqueue(\mathsf{record}(\kappa, S, a_s, l, pas), queue\_T(a_s))$;
$assertor\_T(a_s, \kappa, S).str := \mathsf{READY}$;
-4
}
overall: -4

FIGURE 3.23: The Sender's rules in exchanging phase

$receive\_app(a_s, a_r, a_{ps}, d, \kappa, vl)$:
$\mathsf{app}(d, \kappa, vl) \in k(a_s, a_r) \wedge assertor\_T(a_r, \kappa, R).str = \bot$
$\rightarrow$ {
$receive(\mathsf{app}(d, \kappa, vl), a_s, a_r)$;
$pas \leftarrow createPA(a_r, \kappa, d, \bot)$;
$enqueue(\mathsf{record}(\kappa, R, a_r, vl, pas), queue\_T(a_r))$;
$assertor\_T(a_r, \kappa, R) := \langle \mathsf{READY}, a_{ps}, a_{ps}, vl, pas \rangle$;
$log\_T(a_r, \kappa, R) := \langle \mathsf{FALSE}, \bot \rangle$;
// business logic
}
annotation not applicable

FIGURE 3.24: The Receiver's rules in exchanging phase

$pre\_check(a, \kappa, v, vl, pas)$:
$queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge assertor\_T(a, \kappa, v).str = \mathsf{READY}$
$\rightarrow$ {
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*
$cid' \leftarrow \langle log\_T(a, \kappa', v').aps, \kappa', v' \rangle$;
$cids' := cids' \ominus \{cid\} \oplus \{cid'\}$;
$replaceHead(queue\_T(a), \mathsf{record}(\kappa, v, a, vl, pas))$;
$assertor\_T(a, \kappa, v) := \langle \mathsf{SEND}, *, *, *, pas \rangle$; -4
} overall: -4

$send\_record(a, \kappa, v, vl, pas, t)$:
$queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge assertor\_T(a, \kappa, v).str = \mathsf{SEND}$
$\rightarrow$ {
$a_{ps} \leftarrow assertor\_T(a, \kappa, v).ol$;
$send(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$; +4
$timer\_T(a, \kappa, v, a_{ps}) := \langle \mathsf{ENABLED}, t \rangle$; +1
$assertor\_T(a, \kappa, v).str := \mathsf{SENT}$; -8
} overall: -3

$timer\_click(a, \kappa, v, a_{ps})$:
$timer\_T(a, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa, v, a_{ps}).to > 0$
$\rightarrow$ {
$timer\_T(a, \kappa, v, a_{ps}).to := timer\_T(a, \kappa, v, a_{ps}).to - 1$; -1/t
} overall: -1/t

$timeout\_ack(a, \kappa, v, a_{ps})$:
$timer\_T(a, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge\ timer\_T(a, \kappa, v, a_{ps}).to = 0 \wedge\ rc(a, \kappa, v) > 0$
$\rightarrow$ {
$timer\_T(a, \kappa, v, a_{ps}) := \langle \mathsf{DISABLED}, 0 \rangle$;
$rc(a, \kappa, v) := rc(a, \kappa, v) - 1$; -10
$a'_{ps} \leftarrow random(psList(a))$;
$log\_T(a, \kappa, v) := \langle \mathsf{TRUE}, a'_{ps} \rangle$;
$assertor\_T(a, \kappa, v) := \langle \mathsf{SEND}, *, a'_{ps}, *, * \rangle$; //assert: was SENT +8
} overall: -2

$receive\_ack(a, a_{ps}, \kappa, v)$:
$\mathsf{ack}(\kappa, v) \in k(a_{ps}, a)$
$\rightarrow$ {
$receive(\mathsf{ack}(\kappa, v), a_{ps}, a)$; -2
if $(timer\_T(a, \kappa, v, a_{ps}).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa, v, a_{ps}).to > 0)$, *then*
$dequeue(queue\_T(a))$;
$timer\_T(a, \kappa, v, a_{ps}) := \langle \mathsf{DISABLED}, 0 \rangle$; -n
$assertor\_T(a, \kappa, v).str := \mathsf{ACKED}$; //assert: was SENT -2
} overall: -4-n or -2

$post\_check(a, a_c, \kappa, v)$:
$assertor\_T(a, \kappa, v).str = \mathsf{ACKED}$
$\rightarrow$ {
if $(log\_T(a, \kappa, v).changed = \mathsf{TRUE})$, *then*
$a_{ps} \leftarrow assertor\_T(a, \kappa, v).ol$;
$a_{dps} \leftarrow assertor\_T(a, \kappa, v).vl$;
$send(\mathsf{repair}(\kappa, v, a_{dps}, a_{ps}), a, a_c)$; +18
$assertor\_T(a, \kappa, v).str := \mathsf{OK}$; -22
} overall: -22 or -4

FIGURE 3.25: Assertor's rules in recording phase

$receive\_record(a, a_{ps}, \kappa, v, vl, pas)$:
  $\mathsf{record}(\kappa, v, a, vl, pas) \in k(a, a_{ps}) \wedge \neg crash\_T(a_{ps})$
$\rightarrow$ {
  $receive(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$; -4
  if $(store\_T(a_{ps}, \kappa, v).pas = \emptyset)$, *then*
    $store\_T(a_{ps}, \kappa, v) := \langle a, *, pas \rangle$;
  if $(store\_T(a_{ps}, \kappa, v).vl = \perp)$, *then*
    $store\_T(a_{ps}, \kappa, v).vl := vl$;
    $vlstate\_T(a_{ps}, \kappa, v) := \mathsf{DEFAULT}$;
  $send(\mathsf{ack}(\kappa, v), a_{ps}, a)$; +2
  } overall: -2

$receive\_update(a_{ps}, a_c, \kappa, v, \overline{v}, ol)$:
  $\mathsf{update}(\kappa, \overline{v}, ol) \in k(a_c, a_{ps}) \wedge \neg crash\_T(a_{ps})$
$\rightarrow$ {
  $receive(\mathsf{update}(\kappa, \overline{v}, ol), a_c, a_{ps})$; -4
  $store\_T(a_{ps}, \kappa, v).vl := ol$;
  $vlstate\_T(a_{ps}, \kappa, v) := \mathsf{UPDATED}$;
  $send(\mathsf{uack}(\kappa, \overline{v}), a_{ps}, a_c)$; +2
  } overall: -2

$crash(\kappa, a_{ps})$:
  $fc(\kappa) > 0 \wedge \neg crash\_T(a_{ps})$
$\rightarrow$ {
  $crash\_T(a_{ps}) := \mathsf{TRUE}$; +1
  $fc(\kappa) := fc(\kappa) - 1$; -2
  } overall: -1

$restart(a_{ps})$:
  $crash\_T(a_{ps}) = \mathsf{TRUE}$
$\rightarrow$ {
  $crash\_T(a_{ps}) := \mathsf{FALSE}$; -1
  } overall: -1

FIGURE 3.26: Provenance store's rules

$msg\_loss\_pstore(a_{ps}, a, \kappa, m)$:
  $fc(\kappa) > 0 \wedge crash\_T(a_{ps}) \wedge m \in k(a, a_{ps})$
$\rightarrow$ {
  $k(a, a_{ps}) := k(a, a_{ps}) \ominus \{m\}$; - msg_measure(m)
  $lost(a, a_{ps}) := lost(a, a_{ps}) \oplus \{m\}$; +2
  $fc(\kappa) := fc(\kappa) - 1$; -2
  } overall: - msg_measure(m)

$msg\_loss\_channel(a_{ps}, a, \kappa, m)$:
  $fc(\kappa) > 0 \wedge (m \in k(a, a_{ps}) \vee m \in k(a_{ps}, a))$
$\rightarrow$ {
  if $(m \in k(a, a_{ps}))$, *then*
    $k(a, a_{ps}) := k(a, a_{ps}) \ominus \{m\}$; - msg_measure(m)
    $lost(a, a_{ps}) := lost(a, a_{ps}) \oplus \{m\}$; +2
  else
    $k(a_{ps}, a) := k(a_{ps}, a) \ominus \{m\}$; - msg_measure(m)
    $lost(a, a_{ps}) := lost(a, a_{ps}) \oplus \{m\}$; +2
  $fc(\kappa) := fc(\kappa) - 1$; -2
  } overall: - msg_measure(m)

FIGURE 3.27: Communication channel's rules

$receive\_repair(a, a_{dps}, a_c, \kappa, v, \overline{v}, ol)$:
  $\mathsf{repair}(\kappa, v, a_{dps}, ol) \in k(a, a_c)$
→ {
  $receive(\mathsf{repair}(\kappa, v, a_{dps}, ol), a, a_c)$; -18
  if $(coord\_T(a_c, \kappa, v) = \langle \bot, \bot \rangle)$, *then*
    $coord\_T(a_c, \kappa, v) := \langle a_{dps}, ol \rangle$;
    if $(coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle)$, *then*
      $a'_{dps} \leftarrow coord\_T(a_c, \kappa, \overline{v}).ol$;
      $coord\_T(a_c, \kappa, v) := \langle a'_{dps}, * \rangle$;
      $coord\_T(a_c, \kappa, \overline{v}) := \langle ol, * \rangle$;
      $update\_T(a_c, \kappa, \overline{v}, ol) :=$ UPDATE;//assert: was UPDATE, WAIT or UPDATED +0, +6 or +8
    $a_{ps} \leftarrow coord\_T(a_c, \kappa, v).a_{dps}$;
    $update\_T(a_c, \kappa, v, a_{ps}) :=$ UPDATE; +8
  } overall: -18, -10, -4 or -2

$send\_update(a_c, \kappa, v, a_{ps}, t)$:
  $update\_T(a_c, \kappa, v, a_{ps}) =$ UPDATE
→ {
  $ol \leftarrow coord\_T(a_c, \kappa, v).ol$;
  $send(\mathsf{update}(\kappa, v, ol), a_c, a_{ps})$; +4
  $update\_T(a_c, \kappa, v, a_{ps}) :=$ WAIT; -6
  $timer_T(a_c, \kappa, v, a_{ps}) := \langle$ENABLED$, t \rangle$; +1
  } overall: -1

$timer\_click(a_c, \kappa, v, a_{ps})$:
  $timer\_T(a_c, \kappa, v, a_{ps}).status =$ ENABLED $\wedge\ timer\_T(a_c, \kappa, v, a_{ps}).to > 0$
→ {
  $timer\_T(a_c, \kappa, v, a_{ps}).to := timer\_T(a_c, \kappa, v, a_{ps}).to - 1$; -1/$t$
  } overall: -1/t

$timeout\_uack(a_c, \kappa, v, a_{ps})$:
  $timer\_T(a_c, \kappa, v, a_{ps}).status =$ ENABLED $\wedge\ \ timer\_T(a_c, \kappa, v, a_{ps}).to = 0 \wedge\ \ rc(a_c, \kappa, v) > 0$
→ {
  $update\_T(a_c, \kappa, v, a_{ps}) :=$ UPDATE; //assert: was WAIT +6
  $timer_T(a_c, \kappa, v, a_{ps}) := \langle$DISABLED$, 0 \rangle$;
  $rc(a_c, \kappa, v) := rc(a_c, \kappa, v) - 1$; -10
  } overall: -4

$receive\_uack(a_{ps}, a_c, \kappa, v)$:
  $\mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c)$
→ {
  $receive(\mathsf{uack}(\kappa, v), a_{ps}, a_c)$; -2
  if $(timer\_T(a_c, \kappa, v, a_{ps}).status =$ ENABLED $\wedge\ timer\_T(a_c, \kappa, v, a_{ps}).to > 0)$, *then*
    $timer_T(a_c, \kappa, v, a_{ps}) := \langle$DISABLED$, 0 \rangle$; -n
    $update\_T(a_c, \kappa, v, a_{ps}) :=$ UPDATED; //assert: was WAIT -2
  } overall: -4-n or -2

FIGURE 3.28: Coordinator's rules

Chapter 4

Protocol Analysis

In the previous chapter, we presented F-PReP and formalised it using an Abstract State Machine (ASM). The documentation recorded using F-PReP is expected to meet a number of requirements: Guaranteed Recording, Causelink Accuracy and Viewlink Accuracy.

The contribution of this chapter is therefore the formal proof of the protocol's correctness demonstrating that F-PReP meets these requirements. Specifically, we show that the sender and receiver's interaction records are guaranteed to be recorded in provenance stores and the links in their interaction records are accurate. We also prove the termination of the protocol, which means the ASM executes a finite number of transitions. The properties established in this chapter will be used as building blocks when we investigate the entire retrievability of process documentation in the next chapter.

Recall that we have used tables to maintain link information, e.g., $store\_\mathcal{T}(a_{ps}, \kappa, v).vl$, $assertor\_\mathcal{T}(a, \kappa, v).ol$ and $log\_\mathcal{T}(a, \kappa, v).aps$. To demonstrate Causelink Accuracy and Viewlink Accuracy, we derive a number of equations between these tables.

In the proof, we establish various lemmas and invariants to facilitate the proof of a property. Given an arbitrary valid configuration of the ASM, the 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. This kind of proof has the advantages of being systematic and not prone to error.

The rest of the chapter is organised as follows. We show the ASM terminates in Section 4.1. We formalise requirements Guaranteed Recording, Causelink Accuracy and Viewlink Accuracy as properties and provide proof for them in Sections 4.2, 4.3 and 4.4. Finally, Section 4.5 summarises the proof and concludes this chapter.

We note that all properties established in this chapter are preserved under Assumption 11 (*the retry counter of the assertor and coordinator is always greater than 0*).

This assumption ensures that transitions *timeout_ack* and *timeout_uack* are always fired whenever there is a timeout event.

4.1 Termination

According to the definition of interaction, an interaction involves the exchange of an application message. After firing *send_app* and *receive_app* rules (Figures 3.23, 3.24), the ASM generates a number of transitions to record documentation about the interaction. We demonstrate that the ASM terminates with regard to one interaction. We define the termination property as follows.

Definition 12 (Termination)**.** *Termination is defined as the execution of a finite number of ASM transitions (excluding send_app and receive_app) until there is no longer any enabled transition.*□

A configuration c is said to be reachable if there is a sequence of transitions $t_1, t_2, ..., t_n$ from the initial configuration: $c_i \mapsto^{t_1} c_1 \mapsto^{t_2} c_2 ... \mapsto^{t_n} c$.

In order to prove the termination property, we introduce a system measure that gives an indication of how far the ASM is from completing its transitions. The system measure is defined as follows.

Definition 13 (System Measure)**.** *For any reachable configuration, c, the system measure of a configuration c is the sum of the table measures and message measures in the system:*

$$
\begin{aligned}
&system\_measure(c)\\
&= table\_measure(assertor\_T)\\
&+ table\_measure(update\_T)\\
&+ table\_measure(timer\_T)\\
&+ table\_measure(crash\_T)\\
&+ table\_measure(lost)\\
&+ table\_measure(rc)\\
&+ table\_measure(fc)\\
&+ \textstyle\sum_{a_i \in \mathrm{A}} \sum_{a_j \in \mathrm{A}} \sum_{m \in k(a_i,a_j)} msg\_measure(m)
\end{aligned}
$$

with
$table\_measure(assertor\_T) = \sum_{a \in \mathrm{A}} \sum_{\kappa \in \mathrm{IK}} \sum_{v \in \mathrm{VK}} measure(assertor\_T(a, \kappa, v).str)$
such that
$measure(\mathsf{INIT}) = 40,\ measure(\mathsf{READY}) = 36,\ measure(\mathsf{SEND}) = 32,$
$measure(\mathsf{SENT}) = 24,\ measure(\mathsf{ACKED}) = 22,\ measure(\mathsf{OK}) = 0$

and
$table\_measure(update\_T) = \sum_{\kappa \in \mathrm{IK}} \sum_{v \in \mathrm{VK}} \sum_{a_{ps} \in PID} measure(update\_T(a_c, \kappa, v, a_{ps}))$

such that
$measure(\mathsf{UPDATE}) = 8,\ measure(\mathsf{WAIT}) = 2,\ measure(\mathsf{UPDATED}) = 0$

and
$table\_measure(timer\_T)\ =$
$\sum_{a\in \mathrm{A}} \sum_{\kappa\in \mathrm{IK}} \sum_{v\in \mathrm{VK}} \sum_{a_{ps}\in PID} timer\_T(a,\kappa,v,a_{ps}).to / timer\_T(a,\kappa,v,a_{ps}).to_i$[1]

and
$table\_measure(crash\_T)\ = \sum_{a_{ps}\in PID} measure(crash\_T(a_{ps}))$
such that
$measure(\mathsf{TRUE}) = 1,\ measure(\mathsf{FALSE}) = 0$

and
$table\_measure(lost)\ = \sum_{a_i\in \mathrm{A}} \sum_{a_j\in \mathrm{A}} |lost(a_i,a_j)| * 2$

and
$table\_measure(rc)\ = \sum_{a\in \mathrm{A}} \sum_{\kappa\in \mathrm{IK}} \sum_{v\in \mathrm{VK}} rc(a,\kappa,v) * 10$

and
$table\_measure(fc)\ = \sum_{\kappa\in \mathrm{IK}} fc(\kappa) * 2$

and
$msg\_measure(\mathsf{record}) = 4,\ msg\_measure(\mathsf{ack}) = 2,$
$msg\_measure(\mathsf{repair}) = 18,\ msg\_measure(\mathsf{update}) = 4,\ msg\_measure(\mathsf{uack}) = 2$

□

Intuitively, the processing of a message can update a table, which in turn may trigger the creation of new messages. The system measure of a configuration accounts for messages and the contents of tables. The values for the component measures are chosen such that the system measure of a configuration is always greater than that of any successor configuration via transitions excluding *send_app* and *receive_app*. Pseudo statements in rules are annotated with the change in system measure they cause in Figures 3.23, 3.24, 3.26, 3.27 and 3.28.

Lemma 14. *For any reachable configurations c, c' and for any transition t, such that t leads from c to c' and t is not send_app or receive_app, the following inequality holds:*

$$0 \leq system\_measure(c') < system\_measure(c).$$

□

Proof. First, we note that the system measure of a configuration is always positive or null (Definition 13). Second, the proof proceeds by an analysis of the different possible

[1] $timer\_T(a,\kappa,v,a_{ps}).to_i$ is the initial value of $timer\_T(a,\kappa,v,a_{ps}).to$.

cases for transition t. We consider here the transition *receive_update*. We compute the system measure of the configuration after transition:
(1) An update message is consumed, hence the measure is decreased by 4;
(2) A uack message is inserted into channel, hence the measure is increased by 2.
As a result, after transition, the measure is decreased by 2, which proves the lemma.

Similar reasoning shows that the other transitions except *send_app* or *receive_app* also cause the system measure to decrease strictly. The decrease has been clearly denoted by the change in measures in Figures 3.23, 3.24, 3.26, 3.27 and 3.28.

□

Theorem 15 (Termination)**.** *For any reachable configuration, all transition paths excluding send_app and receive_app terminate.* □

Proof. We proceed with the following reasoning. Let us define a *successor* relation on the set of configurations; c_2 is a successor of c_1 if c_2 is obtained from c_1 by a transition that differs from *send_app* and *receive_app*. With Lemma 14, the system measure of c_1 is always larger than that of its successor configuration c_2, which has a lower bound 0. Therefore, we can conclude that there exists a successor configuration c_k that does not have a successor by executing a transition other than *send_app* or *receive_app*; c_k is a fixed point of the successor relation. Given the fixed point and the fact that every transition excluding *send_app* and *receive_app* decreases the system measure (Lemma 14), we know the ASM can execute a finite number of transitions apart from *send_app* and *receive_app* in all transition paths until it has not any enabled transition. Therefore, by Definition 12, all transition paths that do not use *send_app* and *receive_app* terminate.

□

In the following sections, we will demonstrate that the protocol preserves properties Guaranteed Recording, Causelink Accuracy and Viewlink Accuracy.

In our proof, we rely on case analysis either on its own or in the context of a proof by induction to establish properties. Essentially, the rules of the ASM are analysed to show that after any number of transitions the particular property still holds for the resulting configuration of the state machine. The base case of the induction also requires us to derive a lemma in the initial configuration. For simplification, we omit the parameters of transitions except $timer\_click(a, \kappa, v, a_{ps})$ and $timer\_click(a_c, \kappa, v, a_{ps})$, which have the same transition name and thus require the parameters to distinguish them.

4.2 Guaranteed Recording

We now establish that the protocol meets requirement GUARANTEED RECORDING, formally expressed the following property:

Property 16 (GUARANTEED RECORDING)**.** *For any a, κ and v, the following implication holds when the ASM terminates at final configuration:*
If $assertor\_T(a, \kappa, v).str \neq \bot$, then

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

where $a_{ps} = assertor\_T(a, \kappa, v).ol$ and $vl \neq \bot$.
□

Property 16 shows that an assertor's documentation about an interaction, i.e., assertor's identity (a), viewlink (vl) and p-assertions ($assertor\_T(a, \kappa, v).pas$) will eventually be recorded in a provenance store, i.e., $store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$. The precondition $assertor\_T(a, \kappa, v).str \neq \bot$ means that the assertor a has created an interaction record about interaction κ, where it is in the view of v. Thus, when F-PReP finishes recording the interaction record, the documentation ends up in a provenance store.

To prove Property 16, we need to establish several relationships between the state of an assertor ($assertor\_T(a, \kappa, v).str$) and other system components (from Lemma 17 to Lemma 21). For example, we show that when the protocol terminates, $assertor\_T(a, \kappa, v).str$ is always in the state OK. We also show that if $assertor\_T(a, \kappa, v).str$ is ACKED or OK, then $store\_T(a_{ps}, \kappa, v)$ has a record $\langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$.

We begin with the relationship between $assertor\_T(a, \kappa, v).str$ and the state of protocol messages.

Lemma 17. *For any reachable configuration and for any a, κ and v, the following implication holds:*

If

$$\bigvee \begin{cases} \mathsf{record}(\kappa, v, a, vl, pas) \in k(a, a_{ps}) \\ \mathsf{record}(\kappa, v, a, vl, pas) \in lost(a, a_{ps}) \\ \mathsf{ack}(a, \kappa, v) \in k(a_{ps}, a) \\ \mathsf{ack}(a, \kappa, v) \in lost(a, a_{ps}) \end{cases}$$

then

$$assertor\_T(a, \kappa, v).str = \mathsf{SENT},$$

where $vl = assertor\_T(a, \kappa, v).vl$, $pas = assertor\_T(a, \kappa, v).pas$ and $a_{ps} = assertor\_T(a, \kappa, v).ol$.

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
The statement is preserved by this transition. After this transition, $assertor\_\mathcal{T}(a, \kappa, v).str$ is set to INIT from $\perp$. Therefore, the statement was preserved before the transition and remains valid after the transition.

prepare_record:
The statement is preserved by this transition. After this transition, $assertor\_\mathcal{T}(a, \kappa, v).str$ is set to READY from INIT. Therefore, the statement was preserved before the transition and remains valid after the transition. In addition, $\mathsf{record}(\kappa, v, a, vl, pas)$ is inserted into $queue\_\mathcal{T}(a)$, where $vl = assertor\_\mathcal{T}(a, \kappa, v).vl$ and $pas = assertor\_\mathcal{T}(a, \kappa, v).pas$.

receive_app:
The statement is preserved by this transition. After this transition, $assertor\_\mathcal{T}(a, \kappa, v).str$ is set to READY from $\perp$. Therefore, the statement was preserved before the transition and remains valid after the transition. In addition, $\mathsf{record}(\kappa, v, a, vl, pas)$ is inserted into $queue\_\mathcal{T}(a)$, where $vl = assertor\_\mathcal{T}(a, \kappa, v).vl$ and $pas = assertor\_\mathcal{T}(a, \kappa, v).pas$.

pre_check:
The statement is preserved by this transition. After this transition, $assertor\_\mathcal{T}(a, \kappa, v).str$ is set to SEND from READY. Therefore, the statement was preserved before the transition and remains valid after the transition.

send_record:
The statement is preserved by this transition. After this transition, $\mathsf{record}(\kappa, v, a, vl, pas)$ is inserted into $k(a, a_{ps})$, where $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).ol$. In addition, $assertor\_\mathcal{T}(a, \kappa, v).str$ is set to SENT.

timeout_ack:
The statement is preserved by this transition. After this transition, an alternative store a'_{ps} is selected, where $a'_{ps} = assertor\_\mathcal{T}(a, \kappa, v).ol$. Therefore, $assertor\_\mathcal{T}(a, \kappa, v).ol$ is changed, which makes the antecedent become false.

receive_ack:
The statement is preserved by this transition. After this transition, $\mathsf{ack}(a, \kappa, v)$ is removed from $k(a_{ps}, a)$, which preserves the statement.

post_check:
The statement is preserved by this transition. After this transition, $assertor\_\mathcal{T}(a, \kappa, v).str$

is set from ACKED to OK. Therefore, the statement was preserved before the transition and remains valid after the transition.

receive_record:
The statement is preserved by this transition. After this transition, $\mathsf{record}(\kappa, v, a, vl, pas)$ is removed from $k(a, a_{ps})$ and $\mathsf{ack}(a, \kappa, v)$ is inserted to $k(a_{ps}, a)$.

message_loss_pstore:
message_loss_channel:
The statement is preserved by these transitions. After these transitions,
$\mathsf{record}(\kappa, v, a, vl, pas)$ or $\mathsf{ack}(a, \kappa, v)$ is removed from $k(a, a_{ps})$ or $k(a_{ps}, a)$ and placed in $lost(a, a_{ps})$.

□

After establishing the relationship between $assertor\_T(a, \kappa, v).str$ and protocol messages, we now investigate the relationship between $assertor\_T(a, \kappa, v).str$ and the state of an assertor's timer.

Lemma 18. *For any reachable configuration and for any a, κ and v, the following implication holds:*

$$\bigwedge \begin{cases} timer\_T(a,\ \kappa, v, a_{ps}).status = \mathit{ENABLED} \\ timer\_T(a,\ \kappa, v, a_{ps}).to \geq 0 \end{cases}$$

$$\mathit{iff}$$

$$assertor\_T(a, \kappa, v).str = \mathit{SENT},$$

where $a_{ps} = assertor\_T(a, \kappa, v).ol$. □

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
prepare_record:
receive_app:
pre_check:
The statement is preserved by these transitions. After these transitions,
$timer\_T(a, \kappa, v, a_{ps})$ is not affected and $assertor\_T(a, \kappa, v).str$ is switched among states $\perp$, INIT, READY and SEND. Therefore, the statement was preserved before these transitions and remains valid after them.

send_record:
The statement is preserved by this transition. After this transition, $timer\_T(a, \kappa, v, a_{ps}).status$ is set to ENABLED, $timer\_T(a, \kappa, v, a_{ps}).to$ is initialised, where $a_{ps} = assertor\_T(a, \kappa, v).ol$. In addition, $assertor\_T(a, \kappa, v).str$ is set to SENT.

$timer\_click(a, \kappa, v, a_{ps})$:
The statement is preserved by this transition. After this transition, $timer\_T(a, \kappa, v, a_{ps}).to$ is greater or equal to 0 and $assertor\_T(a, \kappa, v).str$ is not affected.

timeout_ack:
receive_ack:
The statement is preserved by these transitions. After these transitions, $timer\_T(a, \kappa, v, a_{ps}).status$ is set to DISABLED and $timer\_T(a, \kappa, v, a_{ps}).to$ becomes 0. Meanwhile, $assertor\_T(a, \kappa, v).str$ is set to SEND or ACKED.

post_check:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to OK from ACKED. Therefore, the statement was preserved before this transition and remains valid after the transition.

□

Now we reveal the relationship between $assertor\_T(a, \kappa, v).str$ and the assertor's local queue.

Lemma 19. *For any reachable configuration and for any a, κ and v, the following implication holds:*

$assertor\_T(a, \kappa, v).str = \mathit{READY}, \mathit{SEND}$ *or* SENT*, iff*

$$\mathit{record}(\kappa, v, a, vl, pas) \in queue\_T(a),$$

where $vl = assertor\_T(a, \kappa, v).vl$ *and* $pas = assertor\_T(a, \kappa, v).pas$. □

Proof.

We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $assertor\_T(a, \kappa, v)$ and $queue\_T(a)$ are both empty. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to INIT and no message is enqueued.

prepare_record:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$

is set to READY from INIT, and record(κ, v, a, vl, pas) is inserted into $queue\_T(a)$, where $vl = assertor\_T(a, \kappa, v).vl$ and $pas = assertor\_T(a, \kappa, v).pas$.

receive_app:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to READY from $\perp$, and record(κ, v, a, vl, pas) is inserted into $queue\_T(a)$, where $vl = assertor\_T(a, \kappa, v).vl$ and $pas = assertor\_T(a, \kappa, v).pas$.

pre_check:
After this transition, $assertor\_T(a, \kappa, v).str$ is set to SEND from READY, and record(κ, v, a, vl, pas) still remains in $queue\_T(a)$. If pas is modified to update causelinks, same changes apply to $assertor\_T(a, \kappa, v).pas$. Therefore, pas is still equal to $assertor\_T(a, \kappa, v).pas$.

send_record:
The statement is preserved by these transitions. After these transitions, the first part of the statement remains true and record(κ, v, a, vl, pas) is not affected. Therefore, the statement is still preserved.

timeout_ack:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to SEND from SENT (Lemma 18) and record(κ, v, a, vl, pas) is not affected. Therefore, the statement is still preserved.

receive_ack:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to ACKED from SENT (Lemma 17) and record(κ, v, a, vl, pas) is dequeued. Therefore, the statement is still preserved.

post_check:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to OK from ACKED. Therefore, the statement was preserved before this transition and remains valid after the transition.

□

Lemma 20 states the final state of an assertor after finishing recording its interaction record.

Lemma 20. *For any a, κ and v, the following implication holds when the ASM terminates at final configuration:*

If $assertor\_T(a, \kappa, v).str \neq \perp$, then

$$assertor\_T(a, \kappa, v).str = OK.$$

□

PROOF.

When the ASM terminates, no more transition can be fired. This implies that:

- $assertor\_T(a, \kappa, v).str$ cannot be INIT. Otherwise, transition *prepare_record* can be fired.
- $queue\_T(a)$ is empty. Otherwise, assuming $\mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a))$, by Lemma 19, $assertor\_T(a, \kappa, v).str$ can be READY, SEND or SENT. If it is READY or SEND, transitions *pre_check* or *send_record* will be fired. If it is SENT, by Lemma 18, transitions $timer\_click(a, \kappa, v, a_{ps})$ or *timeout_ack* will be fired.
- $assertor\_T(a, \kappa, v).str$ cannot be ACKED. Otherwise, transition *post_check* will be fired.

Therefore, we can conclude that when the ASM terminates, $assertor\_T(a, \kappa, v).str$ is OK.

□

We now establish the relationship between $assertor\_T(a, \kappa, v).str$ and the content of a provenance store. Lemma 21 shows that the receipt of an acknowledgement message is crucial to GUARANTEED RECORDING property.

Lemma 21. *For any reachable configuration and for any a, κ and v, then the following holds:*

If $\mathsf{ack}(\kappa, v) \in k(a_{ps}, a) \vee assertor\_T(a, \kappa, v).str = \mathit{ACKED}$ *or* OK*, then*

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

where $a_{ps} = assertor\_T(a, \kappa, v).ol$ *and* $vl \neq \perp$.

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty and no message is in channel. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
prepare_record:
receive_app:
pre_check:
send_record:
timeout_ack:

The statement is preserved by these transitions. After these transitions, the antecedent is false.

receive_ack:
The statement is preserved by this transition. After this transition, an $\mathsf{ack}(\kappa, v)$ is removed from $k(a_{ps}, a)$ and $assertor\_T(a, \kappa, v).str$ may become ACKED.

post_check:
The statement is preserved by this transition. After this transition, $assertor\_T(a, \kappa, v).str$ is set to OK from ACKED. Therefore, the antecedent remains true, which still preserves the statement.

receive_record:
The statement is preserved by this transition. After this transition, a $\mathsf{record}(\kappa, v, a, vl, pas)$ is received and removed from channel $k(a, a_{ps})$. By Lemma 17, $vl = assertor\_T(a, \kappa, v).vl$, $pas = assertor\_T(a, \kappa, v).pas$ and $a_{ps} = assertor\_T(a, \kappa, v).ol$. Therefore, after this transition, $store\_T(a_{ps}, \kappa, v)$ becomes $\langle a, vl, assertor\_T(a, \kappa, v).pas\rangle$, where $a_{ps} = assertor\_T(a, \kappa, v).ol$ and $vl \neq \perp$. In addition, an $\mathsf{ack}(\kappa, v)$ is inserted into channel $k(a_{ps}, a)$. Hence, the implication holds.

receive_update:
The statement is preserved by this transition. After this transition, the antecedent remains unchanged and $store\_T(a_{ps}, \kappa, v).vl$ is not empty.

msg_loss_pstore:
msg_loss_channel:
After these transitions, $\mathsf{ack}(\kappa, v)$ is removed from $k(a_{ps}, a)$. Therefore, the antecedent may remain unchanged or become false. In either case, the statement is sill valid.

□

Theorem 22 (Guaranteed Recording)**.** *F-PReP preserves Guaranteed Recording property (Property 16).* □

Proof. By Lemma 20, we know that when the ASM terminates, $assertor\_T(a, \kappa, v).str$ is OK. Therefore, by Lemma 21, $store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas\rangle$, where $a_{ps} = assertor\_T(a, \kappa, v).ol$. Hence, when the ASM terminates, the implication is preserved.

□

Theorem 22 shows that once documentation about an interaction is created, it will end up in the assertor's provenance store.

4.3 Causelink Accuracy

Recall the description of relationship p-assertion: a relationship p-assertion made by an assertor (i.e., the sender of an effect interaction) defines the causal relationship between the effect interaction and several cause interactions (where the same assertor is the receiver of a cause interaction). Only in the effect interaction (in which the relationship p-assertion is produced) does the assertor need to update causelinks of the relationship p-assertion to preserve Causelink Accuracy.

Property 23 (Causelink Accuracy)**.** *For any a, κ and vl, let S and R be the sender and receiver's viewkind, respectively, then the following implication holds when the ASM terminates at final configuration:*

If $assertor\_T(a, \kappa, S).str \neq \perp$, *then*

$$\begin{array}{l} for\ any\ pa \in store\_T(a_{ps}, \kappa, S).pas,\ such\ that\ pa = \mathsf{rel\text{-}pa}(rel, \langle \kappa, S \rangle, cids), \\ \quad for\ any\ c \in cids,\ let\ \langle cl', \kappa', R \rangle = c, \\ \quad\quad cl' = assertor\_T(a, \kappa', R).ol\ \wedge \\ \quad\quad store\_T(cl', \kappa', R) = \langle a, vl, assertor\_T(a, \kappa', R).pas \rangle \end{array}$$

where $a_{ps} = assertor\_T(a, \kappa, S).ol$.

□

According to system state space (Section 3.6.1), this property specifies that when the ASM terminates, any relationship p-assertion ($\mathsf{rel\text{-}pa}(rel, \langle \kappa, S \rangle, cids)$) recorded in the sender's provenance store has a set of records ($cids$) about cause interactions and each record has an accurate causelink ($cl' = assertor\_T(a, \kappa', R).ol$), pointing to the store ($store\_T(cl', \kappa', R)$) which successfully recorded documentation about the corresponding cause interaction.

According to transition *pre_check* in Figure 3.25, causelink update takes place when the assertor is ready to record an interaction record to a provenance store. Therefore, we begin our proof by establishing that causelinks are already accurate in an interaction record when being recorded to a provenance store (Lemmas 24 to 26).

Firstly, we show that whenever an assertor uses an alternative store, $log\_T(a, \kappa, v).aps$ keeps the alternative store's address, i.e., the current value of $assertor\_T(a, \kappa, v).ol$.

Lemma 24. *For any reachable configuration and for any a, κ and v, the following implication holds:*

If $log\_T(a, \kappa, v).changed = \mathsf{TRUE}$, *then*

$$log\_T(a, \kappa, v).aps = assertor\_T(a, \kappa, v).ol.$$

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since $log\_T(a, \kappa, v)$ is empty. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
receive_app:
After these transitions, $log\_T(a, \kappa, v).changed$ is set to FALSE. Therefore, the implication is preserved.

timeout_ack:
The statement is preserved by this transition. After this transition, $log\_T(a, \kappa, v).changed$ is set to TRUE and $log\_T(a, \kappa, v).aps$ records the selected alternative store's address, which is referred by $assertor\_T(a, \kappa, v).ol$.

□

Next, we reveal the relationship between $assertor\_T(a, \kappa, S).str$ and $assertor\_T(a, \kappa', R).str$, where κ and κ' represent the respective effect and cause interaction that relate to one relationship.

Lemma 25. *For any reachable configuration and for any a and κ, let S, R be the sender and receiver's viewkind, respectively, the following implication holds:*

If $assertor\_T(a, \kappa, S).str \neq \bot$, INIT *or* READY, *then*

$$for\ any\ pa \in assertor\_T(a, \kappa, S).pas,\ such\ that\ pa = \textsf{rel-pa}(rel, \langle \kappa, S \rangle, cids),$$
$$for\ any\ c \in cids,\ let\ \langle cl', \kappa', R \rangle = c,$$
$$assertor\_T(a,\ \kappa', R).str = \textsf{ACKED}\ or\ \textsf{OK}.$$

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since $assertor\_T(a, \kappa, S).str$ is $\bot$. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
After these transitions, $assertor\_T(a, \kappa, S).str$ is set to INIT from $\bot$. Therefore, the implication is preserved.

prepare_record:
receive_app:
After these transitions, $assertor\_T(a, \kappa, S).str$ is set to READY from $\bot$ or INIT. Therefore, the implication is preserved.

pre_check:
After this transition, $assertor\_T(a, \kappa, S).str$ is set to SEND from READY and $\mathsf{record}(\kappa, S, a, vl, pas)$ is the head of $queue\_T(a)$. According to FIFO nature of a queue, we can imply that $\mathsf{record}(\kappa', R, a, vl', pas')$ has been dequeued, where vl' and pas' are the viewlink and p-assertions with regard to the cause interaction. By Lemma 19, $assertor\_T(a, \kappa', R).str \neq \mathsf{READY}, \mathsf{SEND}$ *and* SENT. Therefore, we can conclude that $assertor\_T(a, \kappa', R).str =$ ACKED *or* OK.

send_record:
timeout_ack:
receive_ack:
post_check:
After these transitions, $assertor\_T(a, \kappa, S).str$ is switched among states SEND, SENT, ACKED and OK. Therefore, the antecedent remains true, which preserves the statement.

□

One important property (Lemma 26) to be established is that if $assertor\_T(a, \kappa, S).str \neq \perp, \mathsf{INIT}$ *or* READY, then any interaction record about a cause interaction has been successfully recorded in a store (i.e., $store\_T(cl', \kappa', R) = \langle a, vl, assertor\_T(a, \kappa', R).pas \rangle$).

Lemma 26. *For any reachable configuration and for any a, κ and vl, let S, R be the sender and receiver's viewkind, respectively, the following implication holds:*

If $assertor\_T(a, \kappa, S).str \neq \perp$, INIT *or* READY*, then*

$$\begin{array}{l} for\ any\ pa \in assertor\_T(a, \kappa, S).pas,\ such\ that\ pa = \mathsf{rel\text{-}pa}(rel, \langle \kappa, S \rangle, cids), \\ \quad for\ any\ c \in cids,\ let\ \langle cl', \kappa', R \rangle = c, \\ \quad\quad cl' = assertor\_T(a, \kappa', R).ol \wedge \\ \quad\quad store\_T(cl', \kappa', R) = \langle a, vl, assertor\_T(a, \kappa', R).pas \rangle \end{array}$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since $assertor\_T(a, \kappa, S).str$ is $\perp$. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
After these transitions, $assertor\_T(a, \kappa, S).str$ is set to INIT from $\perp$. Therefore, the implication is preserved.

prepare_record:
receive_app:

After these transitions, $assertor\_\mathcal{T}(a, \kappa, S).str$ is set to READY from $\perp$ or INIT. Therefore, the implication is preserved.

pre_check:
After this transition, $assertor\_\mathcal{T}(a, \kappa, S).str$ is set to SEND from READY and cl' may be set to $log\_\mathcal{T}(a, \kappa', R).aps$ depending on $log\_\mathcal{T}(a, \kappa', R).changed$. We now analyse $log\_\mathcal{T}(a, \kappa', R).changed$.

- $log\_\mathcal{T}(a, \kappa', R).changed = \mathsf{TRUE}$:
  By Lemma 24, $log\_\mathcal{T}(a, \kappa', R).aps$ is equal to $assertor\_\mathcal{T}(a, \kappa', R).ol$.
- $log\_\mathcal{T}(a, \kappa', R).changed = \mathsf{FALSE}$:
  According to function $createPA$ (defined in Section 3.6.2), when relationship p-assertion $\mathsf{rel\text{-}pa}(rel, \langle \kappa, S \rangle, cids)$ is created, cl' is set to $assertor\_\mathcal{T}(a, \kappa', R).ol$.

Based on the above analysis, we have $cl' = assertor\_\mathcal{T}(a, \kappa', R).ol$. By Lemma 25, $assertor\_\mathcal{T}(a, \kappa', R).str =$ ACKED *or* OK. Hence with Lemma 21, $store\_\mathcal{T}(cl', \kappa', R) = \langle a, vl, assertor\_\mathcal{T}(a, \kappa', R).pas \rangle$. Therefore, the statement is preserved.

send_record:
timeout_ack:
receive_ack:
post_check:
After these transitions, $assertor\_\mathcal{T}(a, \kappa, S).str$ is switched among states SEND, SENT, ACKED and OK. Therefore, the antecedent remains true, which preserves the statement.

□

We are now ready to prove Causelink Accuracy property.

Theorem 27 (Causelink Accuracy)**.** *F-PReP preserves Causelink Accuracy property (Property 23).*
□

Proof. Lemma 20 shows that when the ASM terminates, $assertor\_\mathcal{T}(a, \kappa, S).str$ is OK. By Lemma 26, all causelinks in $assertor\_\mathcal{T}(a, \kappa, S).pas$ are accurate when being recorded in a provenance store. By Property 16, p-assertions in $assertor\_\mathcal{T}(a, \kappa, S).pas$ are eventually recorded in $store_\mathcal{T}(a_{ps}, \kappa, S).pas$where $a_{ps} = assertor\_\mathcal{T}(a, \kappa, S).ol$. Therefore, this implication is preserved. □

After establishing properties Guaranteed Recording and Causelink Accuracy, we show that F-PReP preserves property Viewlink Accuracy in the next section.

4.4 Viewlink Accuracy

Property 16 specifies that when the protocol terminates, an assertor's documentation about an interaction is guaranteed to be recorded in its provenance store. Property 28 requires that a viewlink recorded in the store must be accurate, pointing to the store where the other assertor in the same interaction recorded documentation about that interaction.

Recall that notation $\overline{v}$ 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.

Property 28 (Viewlink Accuracy)**.** *For any a, κ and v, and for some a' and vl', the following implication must hold when the ASM terminates at final configuration:*

If $assertor\_T(a, \kappa, v).str \neq \perp$, then

$$\bigwedge \begin{cases} store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ store\_T(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

where $a_{ps} = assertor\_T(a, \kappa, v).ol$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).ol$.
□

Assertors a and a' are involved in the same interaction but in different views. Therefore, for any a in an interaction, there is some a' in that interaction.

When the ASM terminates, any viewlink in an assertor's provenance store ($store\_T(a_{ps}, \kappa, v)$, such that $a_{ps} = assertor\_T(a, \kappa, v).ol$) points to a correct location ($store\_T(a'_{ps}, \kappa, \overline{v})$, such that $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).ol$), where the other assertor in the same interaction recorded p-assertions.

Our proof is divided into two parts. First, we establish that when the ASM terminates, any viewlink in an assertor's provenance store points to the same location as referred by the opposite assertor's ownlink (i.e., $store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol$). Then we use Property 16 to show that $store\_T(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle$, such that $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).ol$.

Now we outline the proof of the first part. According to F-PReP, an assertor may send a repair request to the coordinator to update the other assertor's viewlink. The coordinator records the request information in table $coord\_T$ and maintains the update state in table $update\_T$, indicating if the update has been successful. Since each interaction involves two views and the assertor in each view may send one request to the coordinator, there are three cases that we need to consider.

1. $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$:
   This is the case where the coordinator did not receive any repair message with

regard to an interaction when the ASM terminates. We will establish the first part of Property 28 for this case in Lemma 42.

2. $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$:
   This is the case where the coordinator received one repair message from the assertor of each view. We will establish the first part of Property 28 for this case in Lemma 46.

3. $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$[2]:
   This is the case where the coordinator received only one repair message. We will establish the first part of Property 28 for this case in Lemma 48.

In order to prove Property 28 in the first case, we firstly establish several lemmas. Lemma 29 show that if the sender did not use any alternative store when recording its interaction record, the receiver's viewlink remains accurate (i.e., $assertor\_T(a_r, \kappa, R).vl = assertor\_T(a_s, \kappa, S).ol$).

Lemma 29. *For any reachable configuration and for any a_s, κ and for some a_r, let S, R be the sender and receiver's viewkind, respectively, then the following implication holds:*

If $assertor\_T(a_s, \kappa, S).str \neq \perp \wedge log\_T(a_s, \kappa, S).changed = \mathit{FALSE}$, *then*

$$\bigvee \left\{ \begin{array}{l} \mathit{app}(d,\ \kappa, a_{ps}) \in k(a_s, a_r) \\ assertor\_T(a_r, \kappa, R).vl = assertor\_T(a_s, \kappa, S).ol \end{array} \right.$$

where $a_{ps} = assertor\_T(a_s, \kappa, S).ol$.

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since $assertor\_T(a, \kappa, v).str$ is $\perp$. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to INIT and $log\_T(a_s, \kappa, S).changed$ is set to FALSE. Meanwhile, $app(d, \kappa, a_{ps})$ is inserted in $k(a_s, a_r)$, where a_{ps} is equal to $assertor\_T(a_s, \kappa, S).ol$. Therefore, the implication is preserved.

prepare_record:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to READY from INIT. Therefore, the precondition of the statement remains true, which preserves the implication.

[2]Due to the symmetric nature, this case is equal to $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$.

receive_app:
After this transition, $app(d, \kappa, a_{ps})$ is removed from $k(a_s, a_r)$ and $assertor\_T(a_r, \kappa, R).vl$ is set to a_{ps}. Since a_{ps} is equal to $assertor\_T(a_s, \kappa, S).ol$ (according to *send_app*), the implication holds.

pre_check:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to READY from SEND. Therefore, the precondition of the statement remains true, which preserves the implication.

send_record:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to SENT from SEND. Therefore, the precondition of the statement remains true, which preserves the implication.

timeout_ack:
After this transition, $log\_T(a_s, \kappa, S).changed$ is set to TRUE. Hence, the precondition of the statement becomes false. Therefore, the implication is still preserved.

receive_ack:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to ACKED from SENT (Lemma 17). Therefore, the precondition of the statement remains true, which preserves the implication.

post_check:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to OK from ACKED. Therefore, the precondition of the statement remains true, which preserves the implication.

□

Recall the definitions in Section 3.5.1, a default link refers to the provenance store that an assertor initially used when recording an interaction record.

Lemma 30 shows that $log\_T(a, \kappa, v).changed$ indicates if an assertor has used an alternative store. If not, the assertor's ownlink remains equal to its default link.

Lemma 30. *For any reachable configuration and for any a, κ and v, the following implication holds:*

$$log\_T(a, \kappa, v).changed = FALSE$$

iff

$$assertor\_T(a, \kappa, v).ol = assertor\_T(a, \kappa, v).dl.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial

configuration since tables are empty. We consider only those transitions that may have an effect on terms in the implication.

send_app:
receive_app:
After this transition, $log\_T(a, \kappa, v).changed$ is set to FALSE and $assertor\_T(a, \kappa, v).ol$ equals to $assertor\_T(a, \kappa, v).dl$. Therefore, the statement is preserved.

timeout_ack:
After this transition, $log\_T(a, \kappa, v).changed$ is set to TRUE and $assertor\_T(a, \kappa, v).ol$ refers to an alternative store. Therefore, the statement is still preserved.

□

Lemma 29 has shown that if the sender of an interaction did not use alternative stores when recording its interaction record, the receiver's viewlink remains correct. In Lemma 31, we show that if the receiver did not use any alternative store, then the sender's viewlink is also correct.

Lemma 31. *For any reachable configuration, for any a_s and κ, and for some a_r, let S, R be the sender and receiver's viewkind, respectively, the following implication holds:*

If $(assertor\_T(a_s, \kappa, S).str \neq (\bot \vee \mathit{INIT})) \wedge log\_T(a_r, \kappa, R).changed = \mathit{FALSE}$, *then*

$$assertor\_T(a_s, \kappa, S).vl = assertor\_T(a_r, \kappa, R).ol$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration, since $assertor\_T(a, \kappa, v).str$ is $\bot$. We now consider only those transitions that may have an effect on terms in the implication.

send_app:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to INIT. Therefore, the precondition is false, which preserves the implication.

prepare_record:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to READY and $assertor\_T(a_s, \kappa, S).vl$ is equal to $assertor\_T(a_r, \kappa, R).dl$. If $log\_T(a_r, \kappa, R).changed$ is FALSE, $assertor\_T(a_r, \kappa, R).dl$ is equal to $assertor\_T(a_r, \kappa, R).ol$ (Lemma 30). Therefore, $assertor\_T(a_s, \kappa, S).vl$ is equal to $assertor\_T(a_r, \kappa, R).ol$, which preserves the statement.

receive_app:
After this transition, $log\_T(a_r, \kappa, R).changed$ is set to FALSE. Since the guard to fire this transition is $\mathsf{app}(d, \kappa, vl)$ in transit. By rule *send_app*, $assertor\_T(a_s, \kappa, S).str$ is

set to INIT when sending $\mathsf{app}(d, \kappa, vl)$. Therefore, the precondition of the statement is false, which preserves the implication.

pre_check:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to SEND from READY. Therefore, the precondition of the statement remains true, which preserves the implication.

send_record:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to SENT from SEND. Therefore, the precondition of the statement remains true, which preserves the implication.

timeout_ack:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to SEND from SENT. Therefore, the precondition of the statement remains true, which preserves the implication.

receive_ack:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to ACKED from SENT. Therefore, the precondition of the statement remains true, which preserves the implication.

post_check:
After this transition, $assertor\_T(a_s, \kappa, S).str$ is set to OK from ACKED. Therefore, the precondition of the statement remains true, which preserves the implication.

□

Lemma 32. *For any a, κ, v and for some a', then the following implication holds when the ASM terminates at final configuration:*

If $log\_T(a, \kappa, v).changed = FALSE$, then

$$assertor\_T(a', \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).ol$$

□

Proof. When the ASM terminates, $assertor\_T(a, \kappa, v).str$ is OK (Lemma 20) and no app message is in transit. Therefore, by Lemma 29 and Lemma 31, the implication is preserved for both views.

□

Lemma 33 connects the state of an assertor and the state of the coordinator. It shows that if an assertor has finished recording and used an alternative store during recording, then it has sent a repair request to a coordinator. On the other hand, if the coordinator has received a repair request, then we can imply that the requesting assertor has recorded its interaction record in an alternative store.

We note that since there is only one coordinator in the system, the coordinator's identity a_c is known.

Lemma 33. *For any reachable configuration and for any a, κ and v, the following equality holds:*

$$assertor\_T(a,\kappa,v).str = \mathsf{OK} \wedge log\_T(a,\kappa,v).changed$$

$$= \bigvee \left\{ \begin{array}{l} \mathsf{repair}(\kappa, v, a'_{ps}, a_{ps}) \in k(a, a_c) \\ coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle \end{array} \right.$$

where $a'_{ps} = assertor\_T(a,\kappa,v).vl$ and $a_{ps} = assertor\_T(a,\kappa,v).ol$. □

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $coord\_T(a_c,\kappa,v)$ is empty and no message is in transit. We now consider only those transitions that may have an effect on terms in the equality.

send_app:
prepare_record:
receive_app:
pre_check:
send_record:
receive_ack:
timeout_ack:
After these transitions, $assertor\_T(a,\kappa,v).str$ is switched among states $\bot$, INIT, READY, SEND, SENT and ACKED. Therefore, the first part of the statement remains false. In addition, these transitions do not affect the right part of the statement. Therefore, the statement is preserved.

post_check:
After this transition, $assertor\_T(a,\kappa,v).str$ is set to OK from ACKED. If $log\_T(a,\kappa,v).changed$ is TRUE, then $\mathsf{repair}(\kappa, v, a'_{ps}, a_{ps})$ is inserted in $k(a,a_c)$, where $a'_{ps} = assertor\_T(a,\kappa,v).vl$ and $a_{ps} = assertor\_T(a,\kappa,v).ol$. Therefore, the statement is preserved.

msg_loss_pstore:
msg_loss_channel:
The two transitions do not remove $\mathsf{repair}(\kappa, v, a'_{ps}, a_{ps})$ from $k(a,a_c)$, since we have assumed that only messages to/from a provenance store can get lost (Assumption 3), $\mathsf{repair}(\kappa, v, a'_{ps}, a_{ps})$ is not lost. Therefore, the statement is preserved.

receive_repair:
After this transition, $\mathsf{repair}(\kappa, v, a'_{ps}, a_{ps})$ is removed from $k(a,a_c)$ and $coord\_T(a_c,\kappa,v)$ becomes not empty.

send_update:

timeout_uack:
receive_uack:
The statement is preserved by these transitions. After these transitions, $coord\_T(a_c, \kappa, v)$ remains unaffected.

□

Lemma 34 and Lemma 35 show how the coordinator updates a destination provenance store by sending an update message.

Lemma 34. *For any reachable configuration and for any κ, v and a_{ps}, the following implication holds:*

If $\mathsf{update}(\kappa, v, ol) \in k(a_c, a_{ps})$, *then*

$$ol = coord\_T(a_c, \kappa, v).ol.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
The statement is preserved by this transition. This transition does not alter $coord\_T(a_c, \kappa, v).ol$ if $coord\_T(a_c, \kappa, v).ol \neq \perp$ before the transition.

send_update:
The statement is preserved by this transition. After this transition, $\mathsf{update}(\kappa, v, ol) \in k(a_c, a_{ps})$ is inserted to $k(a_c, a_{ps})$, such that $ol = coord\_T(a_c, \kappa, v).ol$.

receive_update:
The statement is preserved by this transition. After this transition, $\mathsf{update}(\kappa, v, ol)$ is removed from $k(a_c, a_{ps})$. Therefore, the antecedent becomes false, which preserves the statement.

□

Lemma 35. *For any reachable configuration and for any κ, v and a_{ps}, the following implication holds:*

If $\mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c)$, *then*

$$store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol.$$

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_update:
After this transition, $\mathsf{update}(\kappa, v, ol)$ is removed from $k(a_c, a_{ps})$ and $\mathsf{uack}(\kappa, v)$ is inserted to $k(a_{ps}, a_c)$. In addition, $store\_T(a_{ps}, \kappa, \overline{v}).vl$ is set to ol, which is provided by $\mathsf{update}(\kappa, v, ol)$. By Lemma 34, ol is equal to $coord\_T(a_c, \kappa, v).ol$. Therefore, we have $store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol$.

receive_uack:
The statement is preserved by this transition. After this transition, $\mathsf{uack}(\kappa, v)$ is removed from $k(a_{ps}, a_c)$. Therefore, the antecedent becomes false, which preserves the statement.

□

Lemma 36 shows the three possible states of a provenance store regarding an interaction: initial state, possessing a viewlink and possessing an interaction record.

Lemma 36 (INVARIANT)**.** *For any reachable configuration, for any a_{ps}, κ and v, and for some a, the following statement holds:*

$$\bigvee \begin{cases} store\_T(a_{ps}, \kappa, v) = \langle \bot, \bot, \emptyset \rangle \\ store\_T(a_{ps}, \kappa, v) = \langle \bot, vl, \emptyset \rangle \\ store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \end{cases}$$

where $vl \neq \bot$. □

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $store\_T(a_{ps}, \kappa, v)$ is $\langle \bot, \bot, \emptyset \rangle$. We now consider only those transitions that may have an effect on terms in the equality.

receive_record:
After this transition, $store\_T(a_{ps}, \kappa, v)$ becomes $\langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$, where $vl \neq \bot$.

receive_update:
After this transition, $store\_T(a_{ps}, \kappa, v).vl$ is not $\bot$. If $store\_T(a_{ps}, \kappa, v)$ was $\langle \bot, \bot, \emptyset \rangle$ before the transition, then it becomes $\langle \bot, vl, \emptyset \rangle$, such that $vl \neq \bot$. If $store\_T(a_{ps}, \kappa, v)$ was $\langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$, it remains as $\langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$ with $store\_T(a_{ps}, \kappa, v).vl$ updated.

send_app:
receive_app:

The statement is preserved by these transitions. After these transitions, a new interaction key κ is created. Therefore, $store\_T(a_{ps}, \kappa, v)$ is in the initial state.

□

Lemma 37 gives the relationship among the coordinator's timer, protocol messages and the destination provenance store to be updated.

Lemma 37. *For any reachable configuration and for any κ, v and a_{ps}, the following implication holds:*

If

$$\bigwedge \begin{cases} timer\_T(a_c, \kappa, v, a_{ps}).status = \mathit{ENABLED} \\ timer\_T(a_c, \kappa, v, a_{ps}).to \geq 0 \end{cases}$$

then

$$\bigvee \begin{cases} \mathsf{update}(\kappa, v, ol) \in k(a_c, a_{ps}) \\ \mathsf{update}(\kappa, v, ol) \in lost(a_c, a_{ps}) \\ store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol \\ \mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c) \\ \mathsf{uack}(\kappa, v) \in lost(a_c, a_{ps}) \end{cases}$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
The statement is preserved by this transition. After this transition, $timer\_T(a_c, \kappa, v, a_{ps})$ remains in the initial state and no message is inserted to channels.

send_update:
The statement is preserved by this transition. After this transition, $\mathsf{update}(\kappa, v, ol)$ is inserted to $k(a_c, a_{ps})$. In addition, $timer\_T(a_c, \kappa, v, a_{ps}).status$ is set to ENABLED and $timer\_T(a_c, \kappa, v, a_{ps}).to$ is initialised.

$timer\_click(a_c, \kappa, v, a_{ps})$:
After this transition, $timer\_T(a_c, \kappa, v, a_{ps}).to$ remains greater than or equal to 0. Therefore, the statement is preserved by this transition.

timeout_uack:
After this transition, $timer\_T(a_c, \kappa, v, a_{ps}).status$ is set to DISABLED and

$timer\_T(a_c, \kappa, v, a_{ps}).to$ is set to 0. Therefore, the antecedent is false, which preserves the implication.

receive_uack:
The statement is preserved by this transition. After this transition, $\mathsf{uack}(\kappa, v)$ is removed from $k(a_{ps}, a_c)$. By Lemma 35, $store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol$. Therefore, the implication still holds after the transition.

msg_loss_pstore:
msg_loss_channel:
The statement is preserved by these transitions. After these transitions, $\mathsf{update}(\kappa, v, ol)$ or $\mathsf{uack}(\kappa, v)$ is removed from $k(a_c, a_{ps})$ or $k(a_{ps}, a_c)$ and placed in $lost(a_c, a_{ps})$. Therefore, the implication still holds after the transition.

receive_update:
The statement is preserved by this transition. After this transition, $\mathsf{update}(\kappa, v, ol)$ is removed from $k(a_c, a_{ps})$. In addition, $store\_T(a_{ps}, \kappa, \overline{v}).vl$ is set to ol, which is provided by $\mathsf{update}(\kappa, v, ol)$. By Lemma 34, ol is equal to $coord\_T(a_c, \kappa, v).ol$. Therefore, we have $store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol$, which preserves the statement.

□

Lemma 38 connects the state of the coordinator and the state of a provenance store. It shows how the ASM evolves after the coordinator has received a repair request. Table ($vlstate\_T \in$ VLST) keeps the status of a viewlink. If the state is DEFAULT the viewlink is provided by an assertor without being updated by the coordinator. If it is UPDATED the viewlink has been updated by the coordinator.

Lemma 38. *For any reachable configuration and for any κ and v, then the following equality holds:*

$$coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle$$

$$= \bigvee \begin{cases} \mathsf{update}(\kappa, v, ol) \in k(a_c, a_{ps}) \\ \mathsf{update}(\kappa, v, ol) \in lost(a_c, a_{ps}) \\ vlstate\_T(a_{ps}, \kappa, \overline{v}) = \mathit{UPDATED} \\ store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol \\ \mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c) \\ \mathsf{uack}(\kappa, v) \in lost(a_c, a_{ps}) \\ update\_T(a_c, \kappa, v, a_{ps}) = \mathit{UPDATE} \\ update\_T(a_c, \kappa, v, a_{ps}) = \mathit{UPDATED} \end{cases}$$

where $ol = coord\_T(a_c, \kappa, v).ol$, $a_{ps} = coord\_T(a_c, \kappa, v).a_{dps}$.

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $coord\_T(a_c, \kappa, v)$ is empty and no message is in transit. We now consider only those transitions that may have an effect on terms in the equality.

receive_repair:
The statement is preserved by this transition. After this transition, $coord\_T(a_c, \kappa, v)$ becomes not empty and $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE, where $a_{ps} = coord\_T(a_c, \kappa, v).a_{dps}$. Therefore, the statement is preserved.

send_update:
The statement is preserved by this transition. In order to fire this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is UPDATE. After this transition, $\mathsf{update}(\kappa, v, ol)$ is inserted in $k(a_c, a_{ps})$, where $ol = coord\_T(a_c, \kappa, v).ol$. Therefore, the statement was preserved before this transition and remains valid after it.

timeout_uack:
This transition does not affect $coord\_T(a_c, \kappa, v)$. In order to fire this transition, we have $timer\_T(a_c, \kappa, v, a_{ps}).status = \mathsf{ENABLED}$ and $timer\_T(a_c, \kappa, v, a_{ps}).to > 0$. By Lemma 37, the statement is preserved. After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE. Therefore, the statement was preserved before this transition and remains valid after it.

receive_uack:
The statement is preserved by this transition. After this transition, $\mathsf{uack}(\kappa, v)$ is removed from channel. By Lemma 35, $store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol$. Therefore, the statement is still preserved.

msg_loss_pstore:
msg_loss_channel:
The statement is preserved by these transitions. After these transitions, $\mathsf{update}(\kappa, v, ol)$ or $\mathsf{uack}(\kappa, v)$ is removed from $k(a_c, a_{ps})$ or $k(a_{ps}, a_c)$ and placed in $lost(a_c, a_{ps})$. Therefore, the implication still holds after the transition.

receive_update:
The statement is preserved by this transition. After this transition, $\mathsf{update}(\kappa, v, ol)$ is removed from $k(a_c, a_{ps})$ and $\mathsf{uack}(\kappa, v)$ is inserted in $k(a_{ps}, a_c)$. Besides, $vlstate\_T(a_{ps}, \kappa, \overline{v})$ is set to UPDATED and $store\_T(a_{ps}, \kappa, \overline{v}).vl$ is set to ol, which is from $\mathsf{update}(\kappa, v, ol)$. Since ol is equal to $coord\_T(a_c, \kappa, v).ol$ (Lemma 34), $store\_T(a_{ps}, \kappa, \overline{v}).vl$ is equal to $coord\_T(a_c, \kappa, v).ol$.

□

Lemma 39, Lemma 40 and Lemma 41 establish that if the coordinator did not update a provenance store, then any viewlink recorded in the provenance store is provided by

an assertor.

Lemma 39. *For any reachable configuration, for any a_{ps}, κ, v, and for some a, then the following implication holds:*

If $vlstate\_T(a_{ps}, \kappa, v) = \mathit{DEFAULT}$, then

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

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_record:
After this transition, if $vlstate\_T(a_{ps}, \kappa, v)$ is set to DEFAULT, then $store\_T(a_{ps}, \kappa, v).vl$ is set to vl, which comes from $\mathsf{record}(\kappa, v, a, vl, pas)$ produced by an assertor a. By Lemma 17, vl is equal to $assertor\_T(a, \kappa, v).vl$. Therefore, the implication is preserved.

receive_update:
After this transition, $vlstate\_T(a_{ps}, \kappa, v)$ is set to UPDATED. Therefore, the antecedent is false, which preserves the implication.

□

Lemma 40. *For any reachable configuration and for any a_{ps}, κ and v, then the following implication holds:*

If $store\_T(a_{ps}, \kappa, v).vl \neq \bot$, then

$$vlstate\_T(a_{ps}, \kappa, v) = \mathit{DEFAULT}\ \mathit{or}\ \mathit{UPDATED}.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $store\_T(a_{ps}, \kappa, v)$ is empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_record:
After this transition, if $vlstate\_T(a_{ps}, \kappa, v)$ is set to DEFAULT, then $store\_T(a_{ps}, \kappa, v).vl$ is set to vl, which comes from $\mathsf{record}(\kappa, v, a, vl, pas)$. Therefore, the implication is preserved.

receive_update:
After this transition, if $vlstate\_T(a_{ps}, \kappa, v)$ is set to UPDATED, then $store\_T(a_{ps}, \kappa, v).vl$ is set to ol, which comes from $\mathsf{update}(\kappa, \overline{v}, ol)$. Therefore, the implication is preserved.

□

Lemma 41. *For any reachable configuration and for any a, κ and v, then the following implication holds:*

If $assertor\_T(a, \kappa, v).str = \mathit{OK} \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$, *then*

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

where $a_{ps} = assertor\_T(a, \kappa, v).ol$ □

Proof.

We proceed with the following reasoning. If $coord\_T(a_c, \kappa, \overline{v})$ is $\langle \perp, \perp \rangle$, then by Lemma 38, $vlstate\_T(a_{ps}, \kappa, v)$ is not UPDATED. If $assertor\_T(a, \kappa, v).str$ is OK, then by Lemma 21, $store\_T(a_{ps}, \kappa, v).vl$ is not $\perp$, where $a_{ps} = assertor\_T(a, \kappa, v).ol$. With the fact that $vlstate\_T(a_{ps}, \kappa, v)$ is not UPDATED and by Lemma 40, we can imply that $vlstate\_T(a_{ps}, \kappa, v)$ is DEFAULT. Therefore, with Lemma 39, the implication is preserved.

□

We are now ready to prove Property 28's first part for the first case: assertors' viewlink recorded in their respective provenance store is accurate if the update coordinator did not receive any repair message when the protocol terminates.

Lemma 42. *For any κ and v, and for some a and a', then the following implication holds when the ASM terminates at final configuration:*

If $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$, *then*

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

where $a_{ps} = assertor\_T(a, \kappa, v).ol$.

□

Proof. We proceed with the following reasoning. The statement holds in the initial configuration. When the protocol terminates, $assertor\_T(a, \kappa, v).str = \mathsf{OK}$ and $assertor\_T(a', \kappa, \overline{v}).str = \mathsf{OK}$ (Lemma 20). Since $coord\_T(a_c, \kappa, \overline{v})$ is $\langle \perp, \perp \rangle$, by Lemma 33, $log\_T(a', \kappa, \overline{v}).changed$ is FALSE. Therefore, by Lemma 32,

$$assertor\_T(a, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \tag{4.1}$$

and Lemma 41

$$store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a, \kappa, v).vl. \tag{4.2}$$

From (4.1) and (4.2),

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

such that

$$a_{ps} = assertor\_T(a, \kappa, v).ol.$$

Therefore, the implication is preserved.

□

We now consider the cases where the coordinator received one or two repair requests from assertors. Lemma 43 shows that after receiving a repair request, the coordinator knows the requesting assertor's ownlink, which the coordinator can later use to update another assertor's viewlink.

Lemma 43. *For any reachable configuration, for any κ and v, and for some a, then the following implication holds:*

If $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle$, then

$$coord\_T(a_c, \kappa, v).ol = assertor\_T(a, \kappa, v).ol.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $coord\_T(a_c, \kappa, v)$ is empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
After this transition, $coord\_T(a_c, \kappa, v)$ becomes not empty. Meanwhile, $coord\_T(a_c, \kappa, v).ol$ is set to a_{ps}, which is provided by $\mathsf{repair}(\kappa, v, a'_{ps}, a_{ps})$. By Lemma 33, a_{ps} is equal to $assertor\_T(a, \kappa, v).ol$. Therefore, the statement is preserved.

send_app:
receive_app:
timeout_ack:
The statement is preserved by these transitions. After them, $assertor\_T(a, \kappa, v).ol$

is assigned to a value and $assertor\_T(a, \kappa, v).str$ is set to READY. By Lemma 33, $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle$. Therefore, the antecedent is false, which preserves the statement.

□

Lemma 44 shows that after receiving two repair requests, the coordinator ensures that the destination store to be updated is correct so as to successfully update viewlink in that store.

Lemma 44. *For any reachable configuration and for any κ and v, then the following implication holds:*

If $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$, *then*

$$coord\_T(a_c, \kappa, v).a_{dps} = coord\_T(a_c, \kappa, \overline{v}).ol.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table $coord\_T(a_c, \kappa, v)$ is empty and no message is in transit. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
After this transition, $coord\_T(a_c, \kappa, v)$ is not $\langle \perp, \perp \rangle$. If $coord\_T(a_c, \kappa, \overline{v})$ is also not $\langle \perp, \perp \rangle$, then $coord\_T(a_c, \kappa, v).a_{dps}$ is set to $coord\_T(a_c, \kappa, \overline{v}).ol$. Therefore, the statement is preserved.

□

Lemma 45 shows the end states of the coordinator and the provenance store that has been updated when the ASM terminates.

Lemma 45. *For any κ and v, then the following implication must hold when the ASM terminates at final configuration:*

If $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle$, *then*

$$\bigwedge \begin{cases} vlstate\_T(a_{ps}, \kappa, v) = \mathit{UPDATED} \\ store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol \\ update\_T(a_c, \kappa, v, a_{ps}) = \mathit{UPDATED} \end{cases}$$

where $a_{ps} = coord\_T(a_c, \kappa, v).a_{dps}$.

□

PROOF. We proceed with the reasoning on Lemma 38. When the ASM terminates, no message is in transit. In addition, $update\_T(a_c, \kappa, v, a_{ps}) \neq \mathsf{UPDATE}$. Otherwise, *send_update* transition will be fired. Therefore, by Lemma 38, when the ASM terminates, the implication is preserved.

□

We are now ready to prove Property 28's first part for the second case, i,e., assertors' viewlink recorded in their respective provenance store is accurate if the update coordinator received a repair message from each assertor before the protocol terminates.

Lemma 46. *For any κ and v, and for some a and a', then the following implication holds when the ASM terminates at final configuration:*

If $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$, then

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

where $a_{ps} = assertor\_T(a, \kappa, v).ol$. □

PROOF. We proceed with the following reasoning. By Lemma 45, when the ASM terminates,

$$store\_T(a_{ps}, \kappa, v).vl = coord\_T(a_c, \kappa, \overline{v}).ol \tag{4.3}$$

such that

$$a_{ps} = coord\_T(a_c, \kappa, \overline{v}).a_{dps}. \tag{4.4}$$

With Lemma 44 and (4.4),

$$a_{ps} = coord\_T(a_c, \kappa, v).ol. \tag{4.5}$$

Since $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle$ and $coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$,
by Lemma 43,

$$coord\_T(a_c, \kappa, v).ol = assertor\_T(a, \kappa, v).ol \tag{4.6}$$

and

$$coord\_T(a_c, \kappa, \overline{v}).ol = assertor\_T(a', \kappa, \overline{v}).ol. \tag{4.7}$$

Hence, with (4.5) and (4.6)

$$a_{ps} = assertor\_T(a, \kappa, v).ol. \tag{4.8}$$

Therefore, from (4.3), (4.7) and (4.8), when the protocol terminates,

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

such that

$$a_{ps} = assertor\_T(a, \kappa, v).ol.$$

Therefore, the implication is preserved in the case.

□

After proving that Property 28's first part holds in the first two cases, we proceed to the third case where assertors' viewlink recorded in their respective provenance store is accurate if the update coordinator received only one repair message before the protocol terminates. We first need a lemma showing that if the coordinator receives one repair message, then the destination store to be updated is indicated by the viewlink of the requesting assertor.

Lemma 47. *For any reachable configuration, for any κ and v, and for some a, then the following implication holds:*

If $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$, *then*

$$coord\_T(a_c, \kappa, v).a_{dps} = assertor\_T(a, \kappa, v).vl.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
After this transition, $coord\_T(a_c, \kappa, v)$ is not empty. If $coord\_T(a_c, \kappa, \overline{v})$ is still empty, then $coord\_T(a_c, \kappa, v).a_{dps}$ becomes equal to a_{dps}, which is provided by repair(κ, v, a_{dps}, ol). By rule *post_check* (Figure 3.25), $a_{dps} = assertor\_T(a, \kappa, v).vl$. Therefore, the statement is preserved.

prepare_record:
receive_app:
The statement is preserved by these transitions. After them, $assertor\_T(a, \kappa, v).vl$ are initialised and $assertor\_T(a, \kappa, v).str$ is set to READY. By Lemma 33, $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle$. Therefore, the antecedent is false, which preserves the statement.

□

We now prove Property 28's first part for the third case.

Lemma 48. *For any κ and v, and for some a and a', then the following implication holds when the ASM terminates at final configuration:*

If $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \bot, \bot \rangle$*, then*

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

where $a_{ps} = assertor\_T(a, \kappa, v).ol$*.*

□

Proof. We proceed with the following reasoning. Since $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle$, by Lemma 45, when the ASM terminates,

$$store\_T(a'_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol, \tag{4.9}$$

such that

$$a'_{ps} = coord\_T(a_c, \kappa, v).a_{dps}. \tag{4.10}$$

By Lemma 47,

$$coord\_T(a_c, \kappa, v).a_{dps} = assertor\_T(a, \kappa, v).vl. \tag{4.11}$$

Since $coord\_T(a_c, \kappa, \overline{v})$ is $\langle \bot, \bot \rangle$, $log\_T(a', \kappa, \overline{v}).changed$ is FALSE (Lemma 33). Therefore, by Lemma 32,

$$assertor\_T(a, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol. \tag{4.12}$$

Therefore, with (4.10), (4.11) and (4.12),

$$a'_{ps} = assertor\_T(a', \kappa, \overline{v}).ol. \tag{4.13}$$

By Lemma 43, $coord\_T(a_c, \kappa, v).ol = assertor\_T(a, \kappa, v).ol$. Therefore, with (4.9) and (4.13), when the protocol terminates,

$$store\_T(a'_{ps}, \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).ol \tag{4.14}$$

such that

$$a'_{ps} = assertor\_T(a', \kappa, \overline{v}).ol.$$

Since $coord\_T(a_c, \kappa, \overline{v}) = \langle \bot, \bot \rangle$ and $assertor\_T(a, \kappa, v) =$ OK, when the ASM terminates, by Lemma 41,

$$store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a, \kappa, v).vl. \tag{4.15}$$

such that

$$a_{ps} = assertor\_T(a, \kappa, v).ol. \tag{4.16}$$

Therefore, with (4.12), (4.15) and (4.16)

$$store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol, \tag{4.17}$$

such that

$$a_{ps} = assertor\_T(a, \kappa, v).ol.$$

Therefore, from (4.14) and (4.17), $\forall v \in \text{VK}$, the implication is preserved.

□

Based on the above analysis, we are ready to prove Property 28, i,e., assertors' viewlink recorded in their respective provenance store is accurate when the protocol terminates, pointing to the store where the other assertor in the interaction recorded documentation about the same interaction.

Theorem 49 (Viewlink Accuracy)**.** *F-PReP preserves Viewlink Accuracy property (Property 28).* □

Proof. We proceed with the following reasoning. The statement holds in the initial configuration. We perform case analysis on $coord\_T(a_c, \kappa, v)$ and $coord\_T(a_c, \kappa, \overline{v})$.

(1) $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$:
This is the case where the update coordinator did not receive any repair message with regard to one interaction. By Lemma 42, the first part of the implication is preserved in this case.

(2) $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$:
This is the case where the update coordinator received two repair messages from both assertors. By Lemma 46, the first part of the implication is preserved in this case.

(3) $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \perp, \perp \rangle$:
This is the case where the update coordinator received only one repair message. By Lemma 48, the first part of the implication is preserved in this case.

By Property 16, $store\_T(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle$, such that $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).ol$. Hence, the second part of the implication is also preserved.

Therefore, the implication in Property 28 is preserved.

□

We note that all properties established in the rest of this chapter are preserved under Assumption 11 (*the retry counter of the assertor and coordinator is always greater than 0, i.e.,* $rc(a, \kappa, v) > 0$ *and* $rc(a_c, \kappa, v) > 0$). This assumption ensures that transitions *timeout_ack* and *timeout_uack* are always fired whenever there is a timeout event.

Hence, under this assumption, theorems 22, 27, 49 have established that for one interaction:

- Once documentation about the interaction is created, it will end up in a provenance store;
- The viewlinks and causelinks associated with the interaction are accurate in the provenance store.

4.5 Summary

In this chapter, we have shown that F-PReP preserves properties Termination, Guaranteed Recording, Causelink Accuracy and Viewlink Accuracy. These properties state that after F-PReP eventually finishes recording interaction records and taking remedial actions (Termination), these interaction records are guaranteed to be recorded in provenance stores (Guaranteed Recording) and the links in these interaction records are accurate (Causelink Accuracy and Viewlink Accuracy).

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 these properties. Previous experience has shown that the ASM formalism is suitable for mechanical proof derivations, and several algorithms [119, 123, 121], which are formalised using ASM, have been carried out using Coq [36].

Next chapter will make use of the properties established in this chapter to investigate the properties of process documentation, especially the retrievability of distributed documentation.

Chapter 5

Graph-based Analysis

This dissertation aims to address the problem of recording process documentation in the presence of failures while still ensuring its entire retrievability. We have designed and formalised a recording protocol F-PReP, and have proved that F-PReP has properties Guaranteed Recording, Causelink Accuracy and Viewlink Accuracy. This chapter investigates the properties of process documentation that is recorded by using F-PReP. Specifically, we show that the recorded process documentation is complete and still retrievable in its entirety.

Provenance stores used by an assertor can be classified as two types: the default store (i.e., the one that an assertor initially used when recording an interaction record) and alternative stores. When the protocol terminates, there is a final store that an assertor knows to have successfully recorded its interaction records. The final store can be the default store or an alternative store. All properties established in Chapter 4 are only concerned with the final store.

Recall that an assertor sets a timeout when submitting an interaction record to a store. Due to the impossibility to distinguish store crash from message loss in the event of a timeout, it may be the case that a provenance store has recorded an interaction record whilst the assertor sees a timeout event and has to choose an alternative store. Therefore, the default store and any of the attempted alternative stores may possess duplicate interaction records, which means there would be redundant viewlinks or causelinks recorded in multiple locations (as exemplified in Figures 3.21, 3.22). This however would affect documentation retrievability.

To investigate this issue, this chapter develops another set of properties regarding the topological relationship between an assertor's default store and alternative stores. By translating these properties to a notion of graph, we provide a global view of how distributed documentation is connected after the whole process completes its execution in the presence of failures. Such a graphical representation offers a more intuitive description of the content of provenance stores than the ASM-based formalism since the

graphical notation helps us to hide details of a provenance store and does not involve the coordinator. In addition, the graph-based analysis investigates properties regarding the retrieval of process documentation and facilitates the design of a new traversal function whist the ASM-based formalism does not deal with these concerns.

This chapter has two contributions:

Firstly, we define graph notions to represent the topology of distributed documentation. We also introduce a mapping function to convert an ASM configuration to a graph and perform an exhaustive analysis on the forms of graph. This analysis builds up our confidence in understanding what is actually recorded in provenance stores in the presence of failures and more importantly, it facilitates us to demonstrate the properties of process documentation in provenance stores.

Secondly, we introduce a new query function to retrieve process documentation. The new function, derived from the exhaustive analysis of graphs, searches any candidate provenance store and guarantees that any retrieved process documentation is in its entirety.

This chapter is organised as follows. Section 5.1 defines a notion of graph and a mapping function to convert an ASM configuration to a graph. Section 5.2 establishes several graph properties. In Section 5.3, we exhaustively study various forms of graphs regarding one interaction that may appear. Then we investigate properties for the whole process documentation in Section 5.4. Finally, Section 5.5 discusses several issues and Section 5.6 concludes this chapter. In the appendix (Section 5.7), we provide proofs for the properties used for the exhaustive analysis in Section 5.3.

5.1 Definitions

In this section, we introduce a notion of graph to represent the topology of distributed process documentation spanning across multiple provenance stores. We define and formalise graph elements and then introduce a mapping function to produce a graph from an ASM configuration.

5.1.1 Graph Definitions

To facilitate our discussion, we define a term *Interaction context*.

Definition 50 (Interaction context)**.** *An interaction context specifies the view of an interaction, consisting of an interaction key κ, and a viewkind v.*

There are two interaction contexts regarding one interaction: the Sender's and the Receiver's. An assertor records an interaction record to a provenance store about its interaction context.

The state of a provenance store is reflected by a node on the graph and the links recorded in provenance stores are captured by edges connecting nodes. Since a provenance store records an interaction record with regard to a specific interaction context, we associate each node with the interaction context. Therefore, given a node, we can locate an interaction record in a provenance store.

Definition 51 (Node). *A node contains a pointer to a provenance store that was used to record an interaction record, and also contains the associated interaction context. We use notation* $\langle a_{ps}, \kappa, v\rangle$ *to denote a node.*

By Lemma 36, a store has three states regarding an interaction context: $\langle \perp, vl, \emptyset\rangle$, $\langle a, vl, assertor\_T(a, \kappa, v).pas\rangle$ and $\langle \perp, \perp, \emptyset\rangle$. We define the term Node Kind to reflect the state of a store.

Definition 52 (Node Kind). *A node kind indicates the state of a node:* LINK, FULL, FINAL *and* NULL.

We now define the four kinds. In the rest of this chapter, given notations a, a', vl, vl' and pas, we mean that $a \neq \perp$, $a' \neq \perp$, $vl \neq \perp$, $vl' \neq \perp$ and $pas \neq \emptyset$. Otherwise, notations $\perp$ and $\emptyset$ are explicitly specified.

Definition 53 (Link Node). *A* LINK *node indicates a provenance store which recorded a viewlink about an interaction context, i.e., for some vl,*

$$node = \langle a_{ps}, \kappa, v\rangle, \ \textit{such that}\ store\_T(a_{ps}, \kappa, v) = \langle \perp, vl, \emptyset\rangle.$$

Definition 54 (Full Node). *A* FULL *node indicates a provenance store which recorded an interaction record about an interaction context, i.e., for some a and vl,*

$$node = \langle a_{ps}, \kappa, v\rangle, \ \textit{such that}\ store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas\rangle.$$

Definition 55 (Final Node). *A* FINAL *node is a* FULL *node and the corresponding assertor knows that the related provenance store recorded its interaction record. Formally, for some a and vl,*

$node = \langle a_{ps}, \kappa, v\rangle$, *such that*

$$\bigwedge \begin{cases} store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas\rangle \\ assertor\_T(a, \kappa, v).ol = a_{ps} \\ assertor\_T(a, \kappa, v).str = \textit{ACKED or OK} \end{cases}$$

According to the protocol, when $assertor\_T(a, \kappa, v).str$ becomes ACKED or OK, the assertor has received an acknowledgement from its provenance store. Hence it knows that the store has successfully recorded its interaction record.

Definition 56 (Null Node)**.** *A NULL node indicates a provenance store which did not record any information about an interaction context, i.e.,*

$$node = \langle a_{ps}, \kappa, v \rangle, \text{ such that } store\_T(a_{ps}, \kappa, v) = \langle \bot, \bot, \emptyset \rangle.$$

Definition 57 (Edge)**.** *An edge is unidirectional. It starts from a source node and ends at a destination node. There are two kinds of edges: viewlink edge and causelink edge.*

Viewlink and causelink edges reflect underlying viewlinks and causelinks recorded in provenance stores.

Definition 58 (Viewlink Edge)**.** *A viewlink edge starts from node $\langle a_{ps}, \kappa, v \rangle$ and ends at node $\langle a'_{ps}, \kappa', v' \rangle$, such that $a'_{ps} = store\_T(a_{ps}, \kappa, v).vl$.*

We will identify properties regarding the relationship between the two nodes connected via a viewlink edge in Section 5.2.

In order to define Causelink Edge, we firstly define a function $getCIDS$. This function retrieves all relationship p-assertions recorded in a provenance store about an interaction context and then returns a set of triples, each containing a cause interaction context ($\langle \kappa', v' \rangle$) and an associated causelink (a'_{ps}), obtained from those relationship p-assertions. We note that $getCIDS(a_{ps}, \kappa, R)$ returns empty since the receiver (R) of an interaction never records any relationship p-assertion.

Definition
$getCIDS : \text{PID} \times \text{IK} \times \text{VK} \to \mathbb{P}(\text{PID} \times \text{IK} \times \text{VK})$
$getCIDS(a_{ps}, \kappa, v) =$
$\{\langle a'_{ps}, \kappa', v' \rangle \mid \langle a'_{ps}, \kappa', v' \rangle \in cids, \forall \text{ rel-pa}(r, \langle \kappa, v \rangle, cids) \in store\_T(a_{ps}, \kappa, v).pas\}$

Definition 59 (Causelink Edge)**.** *A causelink edge starts from node $\langle a_{ps}, \kappa, v \rangle$ and ends at node $\langle a'_{ps}, \kappa', v' \rangle$, such that $\langle a'_{ps}, \kappa', v' \rangle \in getCIDS(a_{ps}, \kappa, v)$.*

We will identify properties regarding the relationship between the nodes connected via a causelink edge in Section 5.2.

Definition 60 (Graph)**.** *A graph, G, represents the topology of distributed process documentation spanning across interlinked provenance stores. It contains Nodes, Node Kinds, Viewlink Edges and Causelink Edges.*

A node locates an interaction record in a provenance store and the nodekind annotates what the provenance store actually recorded with regard to an interaction context. An edge tells us where to find another interaction record via a viewlink or causelink.

Figure 5.1 formalises the above terms.

$$
\begin{array}{rcll}
\text{NODE} & = & \text{PID} \times \text{IK} \times \text{VK} & \text{(Nodes)} \\
\text{PID} & : & \text{primitive set} & \text{(PS Identities)} \\
\text{IK} & : & \text{primitive set} & \text{(Interaction Keys)} \\
\text{VK} & = & \{\text{S}, \text{R}\} & \text{(View Kinds)} \\
\text{NK} & = & \text{NODE} \rightarrow \text{KIND} & \text{(Node Kinds)} \\
\text{KIND} & = & \{\mathsf{LINK}, \mathsf{FULL}, \mathsf{FINAL}, \mathsf{NULL}\} & \text{(Kinds)} \\
 & & & \\
\text{VE} & = & \text{NODE} \times \text{NODE} & \text{(ViewLink Edges)} \\
\text{CE} & = & \text{NODE} \times \text{NODE} & \text{(CauseLink Edges)} \\
 & & & \\
\text{GRAPH} & = & \mathbb{P}(\text{NODE}) \times \text{NK} \times \mathbb{P}(\text{VE}) \times \mathbb{P}(\text{CE}) & \text{(Graphs)}
\end{array}
$$

Characteristic Variables:
$n \in \text{NODE}$, $a_{ps} \in \text{PID}$, $\kappa \in \text{IK}$, $v \in \text{VK}$, $n \in \mathbb{P}(\text{NODE})$, $ve \in \mathbb{P}(\text{VE})$, $ce \in \mathbb{P}(\text{CE})$, $nk \in \text{NK}$, $g \in \text{GRAPH}$

Initial State of Configuration:
$g_i = \langle n_i, nk_i, ve_i, ce_i \rangle$
where:
$n_i = \emptyset, \quad nk_i = \lambda a_{ps} \kappa v \cdot \bot, \quad ve_i = \emptyset, \quad ce_i = \emptyset,$

FIGURE 5.1: Graph state space

5.1.2 Mapping Function

We define a graph mapping function T that translates an ASM configuration to a graph. Since this chapter is interested in the final graph reflecting the status of provenance stores after the protocol finishes recording the documentation of a process, this function is designed to convert a final ASM configuration to a final graph.

Definition
$T : \text{C} \rightarrow \text{GRAPH}$
$T(c)$:
initialize g;
for each a_{ps}, κ, v, such that $store\_T(a_{ps}, \kappa, v) \neq \langle \bot, \bot, \emptyset \rangle$
do
$node_1 \leftarrow \langle a_{ps}, \kappa, v \rangle$;
$g.n := g.n \cup \{node_1\}$;
if $store\_T(a_{ps}, \kappa, v) = \langle \bot, vl, \emptyset \rangle \wedge vl \neq \bot$
$g.nk(node_1) :=$ LINK;
elif $assertor\_T(a, \kappa, v).ol = a_{ps} \wedge assertor\_T(a, \kappa, v).str =$ ACKED *or* OK
$g.nk(node_1) :=$ FINAL;
else
$g.nk(node_1) :=$ FULL;
$a'_{ps} \leftarrow store\_T(a_{ps}, \kappa, v).vl$;

$node_2 \leftarrow \langle a'_{ps}, \kappa, \overline{v} \rangle$;
if $store\_T(a'_{ps}, \kappa, \overline{v}) = \langle \bot, \bot, \emptyset \rangle$
$g.n := g.n \cup \{node_2\}$;
$g.nk(node_2) :=$ NULL;
$g.ve := g.ve \cup \{\langle node_1, node_2 \rangle\}$;
for each $\langle a''_{ps}, \kappa', v' \rangle \in getCIDS(a_{ps}, \kappa, v)$
do
$node_3 \leftarrow \langle a''_{ps}, \kappa', v' \rangle$;
$g.ce := g.ce \cup \{\langle node_1, node_3 \rangle\}$;
return g;

The projecting function T takes a final system configuration, c, and produces a graph, g. The function begins with an empty graph and then looks up each entry of all provenance stores (i.e., for any a_{ps}, v and κ, it checks $store\_T(a_{ps}, \kappa, v)$). If the entry is not empty, then a node is produced and the node's kind is set. We note that according to the protocol, if $store\_T(a_{ps}, \kappa, v)$ is not empty, it can either be $\langle \bot, vl, \emptyset \rangle$ or $\langle a, vl, pas \rangle$ (Lemma 36). Therefore, the corresponding node kind cannot be NULL and can only be assigned once according to the definitions of node kind.

The function also adds viewlink and causelink edges to the current graph. If the destination node ($node_2$) of a viewlink edge does not exist (i.e., $store\_T(a'_{ps}, \kappa, \overline{v}) = \langle \bot, \bot, \emptyset \rangle$), then the node is added and its kind is set to NULL. Otherwise, $node_2$ will be added to the graph and assigned a nodekind when the function checks the corresponding entry. We also note that the destination node of a causelink edge is always FINAL in the final graph, to be demonstrated in Lemma 69 in Section 5.2. Therefore, $node_3$ does not need to be added to graph and its kind is not set in this iteration.

5.1.3 Graph Notations

The graph defined in Section 5.1.1 has captured sufficient detail that we need in order to systematically analyse the topology of interlinked documentation. However, to make our graph representation more intuitive and neat, we further classify nodes into three categories: Default Nodes, Intermediary Nodes and Alternative Final Nodes.

Provenance stores used by an assertor can be classified into two types: the default store (i.e., the one that an assertor initially used when recording an interaction record) and alternative stores. When the protocol terminates, there is a final store that an assertor knows to have successfully recorded its interaction records (i.e., the store referred by an assertor's ownlink when the protocol terminates.). The final store can be the default store or an alternative store. When using graph notations, we use a default node to refer to a default store, an intermediary node to refer to an alternative store which is not the

final store, and an alternative final node to indicate an alternative store that is the final store.

In the example of Figure 5.2, the assertor's default store is $PS1$, which failed to record the assertor's interaction record IR. Then the assertor used three alternative stores ($PS2$, $PS3$ and $PS4$) and finally recorded IR in $PS4$. Therefore, $PS4$ is the final store. In terms of nodes, $PS1$ is indicated by a default node, $PS2$ and $PS3$ are indicated by intermediary nodes, and $PS4$ is indicated by an alternative final node.

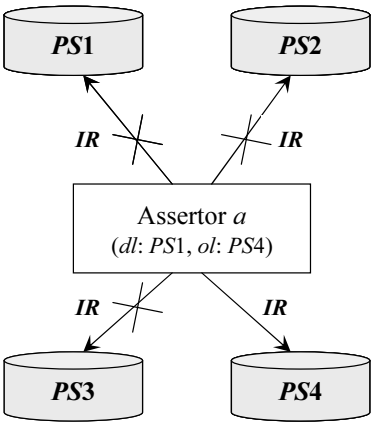


FIGURE 5.2: An example of links

Definition 61 (DEFAULT NODES)**.** $D\_Nodes(\kappa, v) =$
$\{\ \langle a_{ps}, \kappa, v\rangle \mid \exists a,\ a_{ps} = assertor\_T(a, \kappa, v).dl\ \}$

By Figure 3.11, $assertor\_T(a, \kappa, v).dl$ refers to the default link used by assertor a in the interaction context $\langle \kappa, v\rangle$.

To facilitate our discussion, we define a new set $PSSet(a, \kappa, v)$ to capture alternative provenance stores that are not the final store. The set $psList(a)$, defined in Figure 3.11, is the collection of all alternative stores to be used by assertor a[1]. When the ASM terminates, $assertor\_T(a, \kappa, v).str = \mathsf{OK}$ (Lemma 20).

Definition 62. $PSSet(a, \kappa, v) =$
$\{\ a_{ps} \mid a_{ps} \in psList(a) \wedge a_{ps} \neq assertor\_T(a, \kappa, v).ol \wedge assertor\_T(a, \kappa, v).str = \mathsf{OK}\ \}$

With $PSSet$, we define Intermediary Nodes.

Definition 63 (INTERMEDIARY NODES)**.** $I\_Nodes(\kappa, v) =$
$\{\ \langle a_{ps}, \kappa, v\rangle \mid \exists a,\ a_{ps} \in PSSet(a, \kappa, v)\ \}$

Definition 64 (ALTERNATIVE FINAL NODES)**.** $A\_Nodes(\kappa, v) =$
$\{\ \langle a_{ps}, \kappa, v\rangle \mid \exists a, a_{ps} \in psList(a) \wedge a_{ps} = assertor\_T(a, \kappa, v).ol \wedge assertor\_T(a, \kappa, v).str = \mathsf{OK}\ \}$

[1] We can imply that given $a_{ps} = assertor\_T(a, \kappa, v).dl$, $a_{ps} \notin psList(a)$.

We note that for any κ and v, there is one element in $D\_Nodes(\kappa, v)$ and $A\_Nodes(\kappa, v)$, and an alternative final node is always FINAL.

We now introduce a set of graph notations in Figure 5.3.

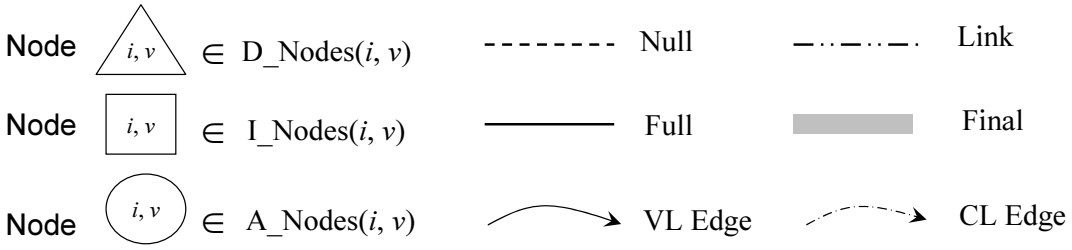


FIGURE 5.3: Graph notations

We use three shapes (triangle, square, circle) to represent nodes (Default Nodes, Intermediary Nodes and Alternative Final Nodes) and use three borders to denote three node kinds (NULL, LINK, FULL). In addition, we use grey colour to fill a FINAL node. Figure 5.4 gives several examples of nodes represented by the graph notations. A node's shape and kind are determined by case analysis in the following sections.

We now give an example demonstrating the graph of a process. We assume there is a process consisting of four interactions, as shown in Figure 5.5. The interaction $i3$ is the effect of interactions $i1$ and $i2$, while $i4$ is the effect of $i3$.

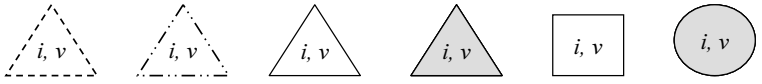


FIGURE 5.4: Examples of nodes

If there was no failure when the five assertors recorded their respective interaction record, a graph (Figure 5.6) is produced after F-PReP finishes recording. Since each assertor used its default store to record an interaction record, all the default nodes are FINAL in the graph, connected by viewlink and causelink edges. If failures happened, then the graph may vary. One possible graph is given in Figure 5.7. We do not explain why nodes are connected in this way. We will provide further explanation after establishing graph properties in the rest of this chapter.

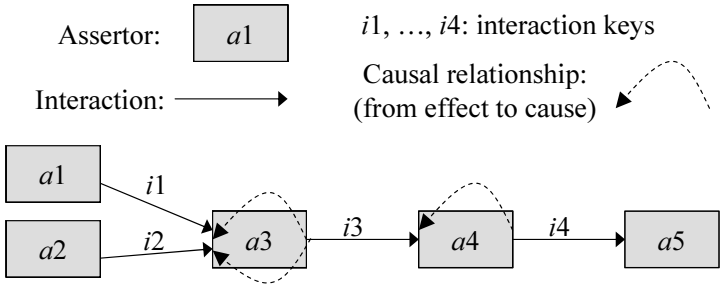


FIGURE 5.5: An example of process

Chapter 4 has proved properties to demonstrate that F-PReP meets requirements CAUSELINK

Accuracy and Viewlink Accuracy. These properties are only concerned with the final store. In order to provide the global topology of distributed documentation, we investigate a number of properties and graphical representations regarding the topological relationship between an assertor's default store and alternative stores in Sections 5.2 and 5.3.

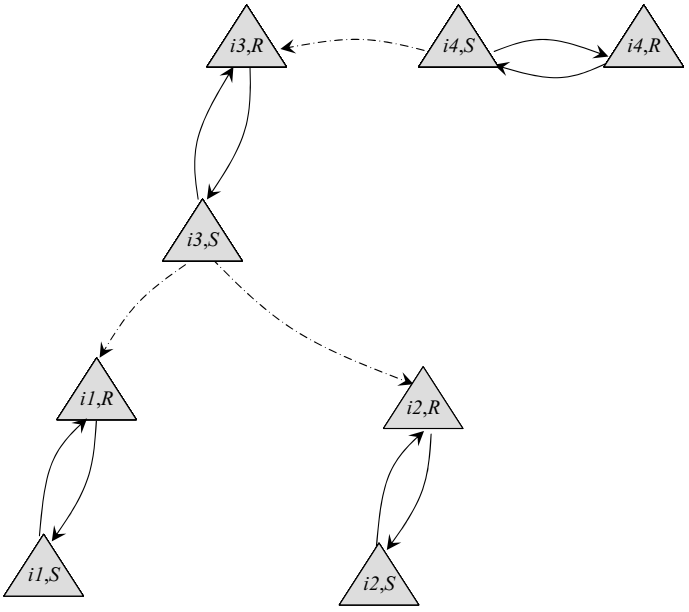


Figure 5.6: Graph produced after F-PReP recorded process documentation in a failure-free environment

5.2 Graph Properties

One purpose of this chapter is to investigate the topology of a final graph g_f, which is exhibited after F-PReP finishes recording process documentation in the presence of failures. Since a process consists of a set of interactions, this section analyses graph topology as a result of two assertors in one interaction finishing recording their interaction records. These properties will be used as building blocks when we investigate the graph properties of a whole process in Section 5.4.

Lemma 65 states the number of FINAL nodes in an interaction context.

Lemma 65. *At final graph g_f, given any interaction context $\langle \kappa, v \rangle$, there is only one FINAL node $\langle a_{ps}, \kappa, v \rangle$, where $a_{ps} = assertor\_T(a, \kappa, v).ol$.*

□

Proof

Given an interaction context $\langle \kappa, v \rangle$, by Guaranteed Recording (Theorem 22), $store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$, such that $a_{ps} = assertor\_T(a, \kappa, v).ol$.

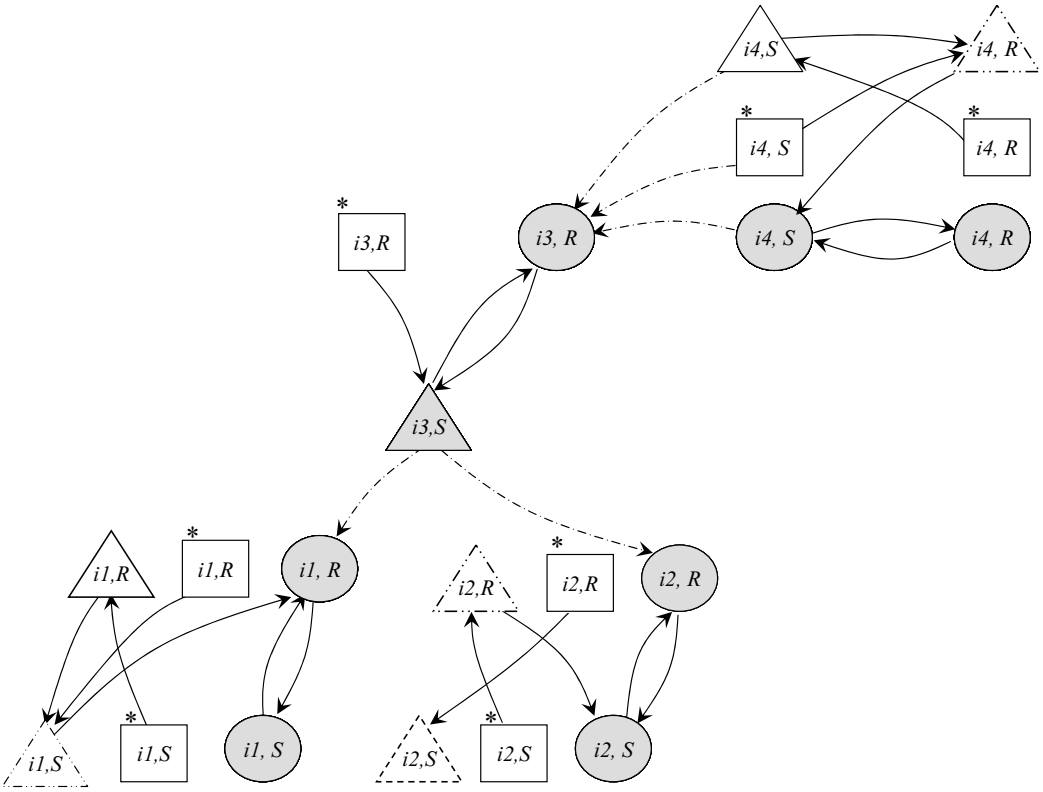


FIGURE 5.7: One possible graph produced after F-PReP finished recording in the presence of failures

By Lemma 20, when the ASM terminates, $assertor\_T(a, \kappa, v).str = \mathsf{OK}$.

Therefore, with function T, there is one final node associated with each interaction context $\langle \kappa, v \rangle$ on g_f.

Given that $assertor\_T(a, \kappa, v).ol$ refers to only one store, there is only one final node in each interaction context.

□

Lemma 66 states the relationship between the two nodes connected via a viewlink edge: these nodes are concerned with the two interaction contexts of the same interaction.

Lemma 66. *At final graph g_f, for any nodes n, $n' \in g_f.n$, such that $n = \langle a_{ps}, \kappa, v \rangle$ and $n' = \langle a'_{ps}, \kappa', v' \rangle$, the following statement holds:*

If $\langle n, n' \rangle \in g_f.ve$, then

$$\kappa = \kappa' \wedge v' = \overline{v}$$

□

PROOF. We proceed with the following reasoning. According to function T, when a viewlink edge is added into the graph, the two nodes connected by the viewlink edge

share the same interaction key and have opposite viewkinds. Therefore, the statement holds.

□

Lemma 67 specifies that the FINAL nodes related to one interaction are connected by two viewlink edges.

Lemma 67. *At final graph g_f, for any nodes n, $n' \in g_f.n$, such that $n = \langle a_{ps}, \kappa, v\rangle$ and $n' = \langle a'_{ps}, \kappa, \overline{v}\rangle$, the following statement holds*:

If $g_f.nk(n) = \mathit{FINAL} \wedge g_f.nk(n') = \mathit{FINAL}$, then

$$\langle n, n'\rangle \in g_f.ve \wedge \langle n', n\rangle \in g_f.ve$$

□

Proof. Given $g_f.nk(n) = \mathsf{FINAL}$ and $g_f.nk(n') = \mathsf{FINAL}$, by definition of FINAL node, there exist $store\_\mathcal{T}(a_{ps}, \kappa, v)$ and $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v})$, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).ol$, and $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol$.

In Section 4.4, we have established Viewlink Accuracy property (Theorem 49). This means that when the ASM terminates, $store\_\mathcal{T}(a_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol$ and $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).vl = assertor\_\mathcal{T}(a, \kappa, v).ol$. Therefore, $store\_\mathcal{T}(a_{ps}, \kappa, v).vl = a'_{ps}$ and $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).vl = a_{ps}$.

By function T, when checking $store\_\mathcal{T}(a_{ps}, \kappa, v)$, a viewlink edge $\langle n, n'\rangle$ is added into $g_f.ve$, such that $n = \langle a_{ps}, \kappa, v\rangle$ and $n' = \langle a'_{ps}, \kappa, \overline{v}\rangle$. Similarly, when checking $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v})$, a viewlink edge $\langle n', n\rangle$ is added into $g_f.ve$, such that $n' = \langle a'_{ps}, \kappa, \overline{v}\rangle$ and $n = \langle a_{ps}, \kappa, v\rangle$.

From the above reasoning, the statement holds.

□

By Lemmas 65, 66 and 67, g_f always contains two FINAL nodes connected via two viewlink edges for a given interaction i, as shown in the example of Figure 5.8. We note that Lemmas 65, 66 and 67 do not state the classification of a node. A node's classification is determined on a case by case basis, which we will discuss in the next section. In Figure 5.8, we assume both nodes are alternative final nodes.

Lemmas 68 and 69 specify properties related to causelink edges. Lemma 68 states the relationship between the nodes connected by a causelink edge.

Lemma 68. *At final graph g_f, for any nodes n, $n' \in g_f.n$, such that $n = \langle a_{ps}, \kappa, v\rangle$ and $n' = \langle a'_{ps}, \kappa', v'\rangle$, the following statements hold*:

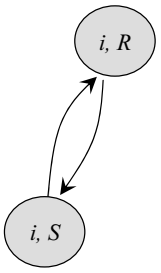


FIGURE 5.8: Interlinked FINAL nodes

If $\langle n, n' \rangle \in g_f.ce$, *then*

$$\kappa \neq \kappa' \wedge v = S \wedge v' = R$$

□

PROOF. We proceed with the following reasoning.

Recall function $getCIDS(a_{ps}, \kappa, v)$, κ' and v' indicate the interaction context in a cause interaction of a relationship p-assertion. Since each interaction key is globally unique, κ' is not equal to κ. A relationship p-assertion is created in function $createPA$, which we recall here:

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* $cl = assertor\_T(a, \kappa', R).ol\}$;
return pas;

According to the definition of $createPA$, the viewkind in the effect interaction of a relationship p-assertion is the sender (S) while the viewkind in a cause interaction is the receiver (R). Therefore, $v = S$ and $v' = R$.

□

Lemma 69 states that any causelink edge in the final graph ends at a FINAL node.

Lemma 69. *At final graph* g_f *and for any nodes* n, $n' \in g_f.n$, *the following statement holds:*

If $\langle n, n' \rangle \in g_f.ce$, *then*

$$g_f.nk(n') = \mathit{FINAL}.$$

□

PROOF. We proceed with the following reasoning. From Lemma 68, we know that if $\langle n, n' \rangle \in g_f.ce$, then n and n' are in contexts $\langle \kappa, S \rangle$ and $\langle \kappa', R \rangle$, respectively.

By CAUSELINK ACCURACY property (Theorem 49), any causelink is accurate when the ASM terminates, pointing to a provenance store that recorded the interaction record about cause interaction ($store\_T(a'_{ps}, \kappa', R) = \langle a, vl, assertor\_T(a, \kappa', R).pas \rangle$, such that $a'_{ps} = assertor\_T(a, \kappa', R).ol$). Since $assertor\_T(a, \kappa', R).str = \mathsf{OK}$ when the ASM terminates, n' is a FINAL node.

□

Lemma 70 gives the number of causelink edges that may appear in a graph.

Lemma 70. *At final graph g_f and for any node $n \in g_f.n$, $|getCIDS(n)| = |\{\langle n, n' \rangle | \langle n, n' \rangle \in g_f.ce, for\ some\ n'\}|$.* □

PROOF. We proceed with the following reasoning. By mapping function T, whenever a node n is added to $g_f.n$, $|getCIDS(n)|$ causelink edges starting from n are added into $g_f.ce$. Therefore, the statement holds. □

With Lemmas 68 and 69, any causelink edge starts from a node in the sender's interaction context and ends at the FINAL node in the receiver's context of another interaction. We now give an example of graph describing causelink edges in Figure 5.9. In Figure 5.9, there is one effect interaction i and three cause interactions j, k and l. Assume n is the node in the context $\langle i, S \rangle$, such that $|getCIDS(n)| = 3$. Therefore, there are three causelink edges in the graph, each pointing to a FINAL node in the corresponding cause interaction.

We note that Lemmas 68, 69 and 70 do not state the classification of a node nor the nodekind of the starting node n. This information is determined on a case by case basis, which we will discuss in the next section.

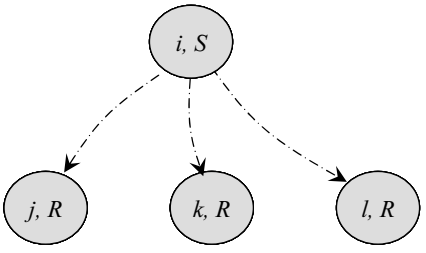


FIGURE 5.9: Nodes connected via causelink edges

Lemma 71 states that the destination node of a viewlink edge or causelink edge is always in the same graph as those edges.

Lemma 71. *At final graph g_f, for any nodes n and n', the following statement holds:*

If $\langle n, n' \rangle \in g_f.ve \vee \langle n, n' \rangle \in g_f.ce$, *then*

$$n' \in g_f.n.$$

□

Proof. We proceed with the following reasoning.

With function T, we consider a new viewlink edge $\langle n, n' \rangle$ added to $g_f.ve$, such that $n' = \langle a'_{ps}, \kappa', v' \rangle$. If $store\_T(a'_{ps}, \kappa', v')$ is $\langle \perp, \perp, \emptyset \rangle$, then n' is added to $g_f.n$. Otherwise, n' is added to $g_f.n$ when function T checks $store\_T(a'_{ps}, \kappa', v')$. Therefore, the lemma is preserved.

We now discuss a new causelink edge $\langle n, n' \rangle$ added to $g_f.ce$, such that $n' = \langle a'_{ps}, \kappa', v' \rangle$.

By Causelink Accuracy property (Theorem 49), any causelink is accurate when the ASM terminates, pointing to a provenance store that recorded the interaction record about a cause interaction ($store\_T(a'_{ps}, \kappa', v') = \langle a, vl, assertor\_T(a, \kappa', v').pas \rangle$, such that $a'_{ps} = assertor\_T(a, \kappa', v').ol$). Therefore, node n' is added to $g_f.n$ when function T checks $store\_T(a'_{ps}, \kappa', v')$.

Based on the above analysis, the lemma is preserved.

□

After identifying basic graph properties, we now perform systematic analysis on graph topologies, which is exhibited after F-PReP terminates for recording interaction records of the two assertors in one interaction. We will investigate graph properties for the whole process documentation in Section 5.4.

5.3 Exhaustive Analysis

In this section, we exhaustively analyse cases that affect the shape of a graph. Since a viewlink edge and a causelink edge connect two nodes concerned with the same and different interactions (Lemmas 66 and 68), respectively, we discuss graphs containing viewlink edges in Section 5.3.1 and then those related to causelink edges in Section 5.3.2 to separate concerns. The proof for several properties used in this section is provided as an appendix (Section 5.7) at the end of this chapter.

5.3.1 Viewlink Edges

To facilitate our discussion regarding the use of alternative stores, we begin our discussion with the case where only an assertor's default store records a duplicate interaction record (i.e., Definition 62, $PSSet(a, \kappa, v) = \emptyset$). The results are summarised in Figure 5.10 and will be explained in Section 5.3.1.1.

Next, we take $PSSet(a, \kappa, v) \neq \emptyset$ into consideration. We consider that any number of intermediary provenance stores may also duplicately record an assertor's interaction record. The results for this case are summarised in Figures 5.12 and 5.13 and will be explained in Section 5.3.1.2.

5.3.1.1 $PSSet(a, \kappa, v) = \emptyset$

We will have an exhaustive analysis on the topology of graphs generated after the two assertors in one interaction finish recording their respective interaction record in the event of failures. In terms of one assertor, there are only the following three cases.

We discuss two situations where an alternative store was/was not used during recording an interaction record (determined by $log\_T(a, \kappa, v).changed$). If an alternative store was used, we further discuss if the default store recorded any duplicate interaction record, which would result in redundant information and affect graph topology (determined by $store\_T(a_{ps}, \kappa, v)$).

(1) An assertor successfully recorded its interaction record to its default store. Recall that $log\_T(a, \kappa, v).changed$ indicates if an assertor has used any alternative store to record its interaction record (Lemma 30). We can formally express this case as follows:

When the ASM terminates at final configuration,

$$log\_T(a, \kappa, v).changed = \mathsf{FALSE}$$

(2) An assertor used an alternative store to record its interaction record in the case of failures and the default store did not record the interaction record.

Formally, when the ASM terminates at final configuration,

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T(a}_{ps}, \kappa, v) \neq \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \end{cases}$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$.

(3) An assertor used an alternative store to record its interaction record in the case of failures but the default store also duplicately recorded the interaction record.

Formally, when the ASM terminates at final configuration,

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T(a}_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \end{cases}$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$.

Given that there are two assertors (the sender and receiver) in one interaction and there are totally three cases with regard to each assertor, we have nine cases to consider. Since the role of the sender and receiver is interchangeable in the nine cases, we reduce the symmetric ones to the following six cases:

(a) Both the sender and receiver successfully recorded their interaction record to their respective default store.

Formally, for any a, κ and v, and for some a', when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{FALSE} \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{FALSE} \end{cases}$$

(b) One assertor successfully recorded its interaction record to its default store; the other assertor used an alternative store to record its interaction record in the case of failures and its default store did not record its interaction record.

Formally, for any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{FALSE} \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

(c) One assertor successfully recorded its interaction record to its default store; the other assertor used an alternative store to record its interaction record in the case of failures and its default store also duplicately recorded its interaction record.

Formally, for any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{FALSE} \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

(d) Both assertors used an alternative store to record their interaction record in the case of failures and their default stores did not record their interaction records.

Formally, for any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T(a}_{ps}, \kappa, v) \neq \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

(e) Both assertors used an alternative store to record their interaction record in the case of failures; one's default store did not record its interaction record but the other one's default store did.

Formally, for any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T(a}_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

(f) Both assertors used an alternative store to record their interaction record in the case of failures and their default stores also duplicately recorded their interaction records.

Formally, for any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T(a}_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

After identifying all possible cases, we show the corresponding graphs produced by each

case. The results are summarised in Figure 5.10. Since all our discussion in this section is concerned with one interaction, we omit a node's interaction key in our graphs for simplification. Due to the symmetric nature of these cases, we assume viewkinds v and $\overline{v}$ are roles of the receiver and sender of an interaction in our graphs.

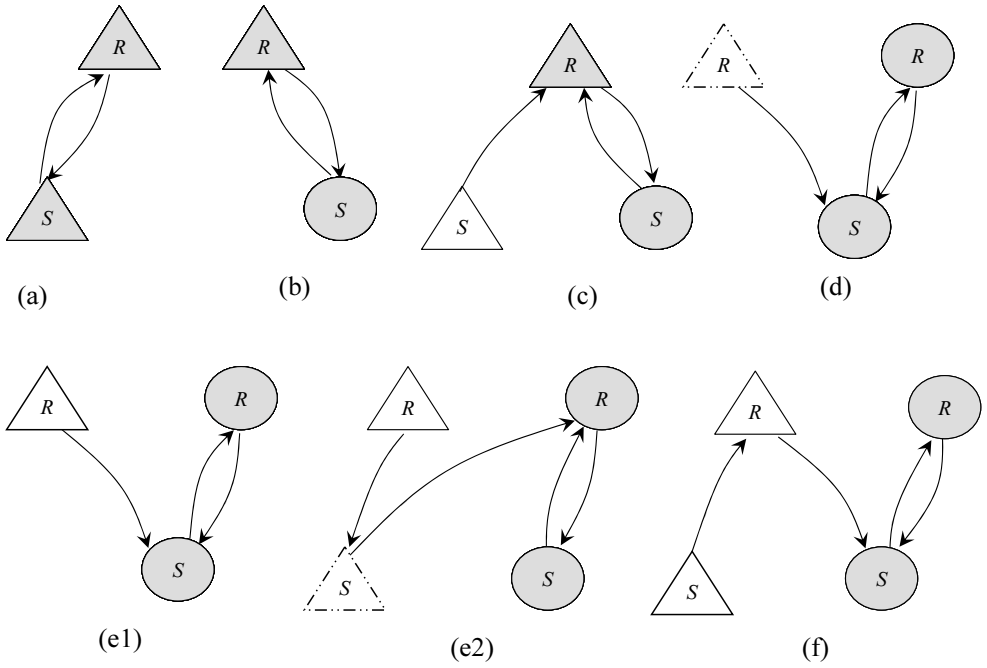


FIGURE 5.10: Graph summary ($PSSet(a, \kappa, v) = \emptyset$)

Given $PSSet(a, \kappa, v) = \emptyset$, only an assertor's default store may record duplicate interaction records. This means there is no intermediary node on graphs. We now discuss each of the six cases regarding the topology between default nodes and alternative final nodes.

Case (a):
For any a, κ and v, and for some a', when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{FALSE} \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{FALSE} \end{cases}$$

Figure 5.10(a) shows the corresponding graph. By Lemma 30, $assertor\_\mathcal{T}(a, \kappa, v).ol = assertor\_\mathcal{T}(a, \kappa, v).dl$ and $assertor\_T(a', \kappa, \overline{v}).ol = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$. Therefore, there is a default node and no alternative final node in either interaction context. By Lemmas 65 and 67, the two nodes are FINAL, connected via two viewlink edges.

Case (b):
For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$ and $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{FALSE} \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

Figure 5.10(b) shows the corresponding graph. By Lemma 30, $assertor\_T(a, \kappa, v).ol = assertor\_T(a, \kappa, v).dl$ and $assertor\_T(a', \kappa, \overline{v}).ol \neq assertor\_T(a', \kappa, \overline{v}).dl$. Therefore, there is one default node and no alternative final node in interaction context $\langle \kappa, v \rangle$ and there is one alternative final node in the other interaction context $\langle \kappa, \overline{v} \rangle$. We now analyse if there is a default node in interaction context $\langle \kappa, \overline{v} \rangle$ on the graph.

By Lemma 82 (proved in Section 5.7),

$$\bigwedge \begin{cases} \text{store\_T(a}_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}).vl = \bot \end{cases}$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$.

Since $store\_T(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle$, by Lemma 36, the fact that $store\_T(a'_{ps}, \kappa, \overline{v}).vl = \bot$ implies that $store\_T(a'_{ps}, \kappa, \overline{v}) = \langle \bot, \bot, \emptyset \rangle$. This means mapping function T (Section 5.1.2) does not add node $\langle a'_{ps}, \kappa, \overline{v} \rangle$ when checking $store\_T(a'_{ps}, \kappa, \overline{v})$. In addition, since $store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol$ and $assertor\_T(a', \kappa, \overline{v}).ol \neq assertor\_T(a', \kappa, \overline{v}).dl$, function T also does not add a default node for interaction context $\langle \kappa, \overline{v} \rangle$ when checking $store\_T(a_{ps}, \kappa, v)$. Therefore, there is no default node in interaction context $\langle \kappa, \overline{v} \rangle$.

With Lemmas 65 and 67, the default node in context $\langle \kappa, v \rangle$ and alternative final node in context $\langle \kappa, \overline{v} \rangle$ are FINAL, connected via two viewlink edges.

Case (c):
For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{FALSE} \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

Figure 5.10(c) shows the corresponding graph. By Lemma 30, $assertor\_T(a, \kappa, v).ol = assertor\_T(a, \kappa, v).dl$ and $assertor\_T(a', \kappa, \overline{v}).ol \neq assertor\_T(a', \kappa, \overline{v}).dl$. Therefore, there is one default node and no alternative final node in interaction context $\langle \kappa, v \rangle$ and there is one alternative final node in the other interaction context $\langle \kappa, \overline{v} \rangle$. Given $store\_T(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle$, where $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$,

there is a default node (which is FULL) in interaction context $\langle \kappa, \overline{v} \rangle$.

By Lemma 80 (proved in Section 5.7), $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).vl = assertor\_\mathcal{T}(a, \kappa, v).ol$, such that $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$. This means the default node in interaction context $\langle \kappa, \overline{v} \rangle$ is connected with the default node in context $\langle \kappa, v \rangle$ via a viewlink edge.

With Lemmas 65 and 67, we can express the corresponding graph in Figure 5.10(c).

Case (d):
For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$ and $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T(a}_{ps}, \kappa, v) \neq \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle \\ \text{log\_T(a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

Figure 5.10(d) shows the corresponding graph. By Lemma 30, $assertor\_\mathcal{T}(a, \kappa, v).ol \neq assertor\_\mathcal{T}(a, \kappa, v).dl$ and $assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \neq assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$. Therefore, there is an alternative final node in either interaction context. We now analyse if there is a default node in either interaction context on the graph.

By Lemma 87 (proved in Section 5.7),

$$\bigwedge \begin{cases} \text{store\_T(a}_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \\ \text{store\_T(a}'_{ps}, \kappa, \overline{v}).vl = \bot \end{cases}$$

such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$ and $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$.

Since $store\_\mathcal{T}(a_{ps}, \kappa, v) \neq \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle$, by Lemma 36, the fact $store\_\mathcal{T}(a_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).ol$ infers that $store\_\mathcal{T}(a_{ps}, \kappa, v) = \langle \bot, vl, \emptyset \rangle$, such that $vl = assertor\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).ol$. This means that there is a default node (which is LINK) in interaction context $\langle \kappa, v \rangle$ on the graph.

Since $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle$, by Lemma 36 and the fact $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).vl = \bot$, we can infer that $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) = \langle \bot, \bot, \emptyset \rangle$. This means that function T does not add node $\langle a'_{ps}, \kappa, \overline{v} \rangle$ when checking $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v})$. In addition, since $store\_\mathcal{T}(a_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol$ and $assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \neq assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, mapping function T does not add a default node for interaction context $\langle \kappa, \overline{v} \rangle$ when checking $store\_\mathcal{T}(a_{ps}, \kappa, v)$. Therefore, there is no default node in interaction context $\langle \kappa, \overline{v} \rangle$ on the graph.

With Lemmas 65 and 67, we can express the corresponding graph in Figure 5.10(d).

Case (e):

For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$ and $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T}(\text{a}_{ps}, \kappa, v) = \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle \\ \text{log\_T}(\text{a}', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T}(\text{a}'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

Figure 5.10(e1) shows the corresponding graph. By Lemma 30, $assertor\_\mathcal{T}(a, \kappa, v).ol \neq assertor\_\mathcal{T}(a, \kappa, v).dl$ and $assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \neq assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$. Therefore, there is an alternative final node in either interaction context. We now analyse if there is a default node in either interaction context on the graph.

By Lemma 88 (proved in Section 5.7), there are two subcases.
Case (e.1):

$$\bigwedge \begin{cases} \text{store\_T}(\text{a}_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \\ \text{store\_T}(\text{a}'_{ps}, \kappa, \overline{v}).vl = \bot \end{cases}$$

Given $store\_\mathcal{T}(a_{ps}, \kappa, v) = \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle$, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$, there is a default node (which is FULL) in interaction context $\langle \kappa, v \rangle$.

Since $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle$, by Lemma 36 and the fact that $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).vl = \bot$, we can infer that $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) = \langle \bot, \bot, \emptyset \rangle$. This means function T does not add node $\langle a'_{ps}, \kappa, \overline{v} \rangle$ when checking $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v})$. In addition, since $store\_\mathcal{T}(a_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol$ and $assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \neq assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, function T does not add a default node for interaction context $\langle \kappa, \overline{v} \rangle$ when checking $store\_\mathcal{T}(a_{ps}, \kappa, v)$. Therefore, there is no default node in interaction context $\langle \kappa, \overline{v} \rangle$ on the graph.

With Lemmas 65 and 67, we can express the corresponding graph in Figure 5.10(e1).

Case (e.2):

$$\bigwedge \begin{cases} \text{store\_T}(\text{a}_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl \\ \text{store\_T}(\text{a}'_{ps}, \kappa, \overline{v}).vl = assertor\_\mathcal{T}(a, \kappa, v).ol \end{cases}$$

Figure 5.10(e2) shows the corresponding graph. Given $store\_\mathcal{T}(a_{ps}, \kappa, v) = \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle$, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$, there is a default node (which is FULL) in interaction context $\langle \kappa, v \rangle$.

Since $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle$, by Lemma 36 and the fact

$store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}).vl = assertor\_\mathcal{T}(a, \kappa, v).ol$, we can infer $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) = \langle \bot, vl, \emptyset \rangle$, such that $vl = assertor\_\mathcal{T}(a, \kappa, v).ol$. This means there is a default node (which is LINK) in interaction context $\langle \kappa, \overline{v} \rangle$.

With Lemmas 65 and 67, we can express the corresponding graph in Figure 5.10(e2).

Case (f):
For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$ and $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, when the ASM terminates at final configuration:

$$\bigwedge \begin{cases} \text{log\_T(a, } \kappa, v).changed = \mathsf{TRUE} \\ \text{store\_T}(a_{ps}, \kappa, v) = \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle \\ \text{log\_T}(a', \kappa, \overline{v}).changed = \mathsf{TRUE} \\ \text{store\_T}(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

Figure 5.10(f) shows the corresponding graph. By Lemma 30, $assertor\_\mathcal{T}(a, \kappa, v).ol \neq assertor\_\mathcal{T}(a, \kappa, v).dl$ and $assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \neq assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$. Therefore, there is an alternative final node in either interaction context. We now analyse if there is a default node in either interaction context on the graph.

Given $store\_\mathcal{T}(a_{ps}, \kappa, v) = \langle a, vl, assertor\_\mathcal{T}(a, \kappa, v).pas \rangle$, such that $a_{ps} = assertor\_\mathcal{T}(a, \kappa, v).dl$, there is a default node (which is FULL) in interaction context $\langle \kappa, v \rangle$.

Given $store\_\mathcal{T}(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_\mathcal{T}(a', \kappa, \overline{v}).pas \rangle$, such that $a'_{ps} = assertor\_\mathcal{T}(a', \kappa, \overline{v}).dl$, there is a default node (which is FULL) in interaction context $\langle \kappa, \overline{v} \rangle$.

By Lemma 89 (proved in Section 5.7),

$$\bigwedge \begin{cases} \text{store\_T}(a_{ps}, \kappa, v).vl = assertor\_\mathcal{T}(a', \kappa, \overline{v}).ol \\ \text{store\_T}(a'_{ps}, \kappa, \overline{v}).vl = assertor\_\mathcal{T}(a, \kappa, v).dl \end{cases}$$

There are two viewlink edges each starting from a default node in the graph.

With Lemmas 65 and 67, we can express the corresponding graph in Figure 5.10(f).

5.3.1.2 $PSSet(a, \kappa, v) \neq \emptyset$

After discussing the case where only an assertor's default store may record a duplicate interaction record (i.e., $PSSet(a, \kappa, v) = \emptyset$), we now take $PSSet(a, \kappa, v) \neq \emptyset$ into consideration, where any number of intermediary provenance stores may also record a duplicate interaction record.

According to the definition of $PSSet(a, \kappa, v)$ (Definition 62), for any $a_{ps} \in PSSet(a, \kappa, v)$, a_{ps} is not the default store nor the final store used by assertor a in interaction context $\langle \kappa, v \rangle$. Therefore, when assertor a finishes recording its interaction record, the presence of $PSSet(a, \kappa, v) \neq \emptyset$ does not alter the existing topology of a graph as summarised in Figure 5.10.

If a provenance store referred by the element in $PSSet(a, \kappa, v)$ recorded any duplicate interaction record, an intermediary node and an associated viewlink edge would be added into the graphs in Figure 5.10. By Lemma 36 and Lemma 91 (proved in Section 5.7), we can infer that given $a_{ps} \in PSSet(a, \kappa, v)$ and $store\_T(a_{ps}, \kappa, v) \neq \langle \perp, \perp, \emptyset \rangle$, an intermediary node, $\langle a_{ps}, \kappa, v \rangle$, is a FULL node. Lemma 91 also specifies that a viewlink edge starting from $\langle a_{ps}, \kappa, v \rangle$ always ends at the default node of the opposite view in the same interaction.

Since there may be several intermediary nodes regarding the same interaction context, to facilitate our presentation, we use a star $*$ to indicate any number of intermediary nodes (num ≥ 0). In the example of Figure 5.11, there are several intermediary nodes in the sender's interaction context, each connected to the same node in the receiver's interaction context via a viewlink edge. This can be simplified by using a star $*$ on the graph.

We augment Figure 5.10 with any number of intermediary nodes and associated viewlink edges to have a generic representation, as shown in Figure 5.12. We note that Graph (A) remains the same because no failure occurred and hence no alternative store was used by each assertor. Since notation $*$ represents any number of nodes (num ≥ 0), Figure 5.12 includes the cases in Figure 5.10.

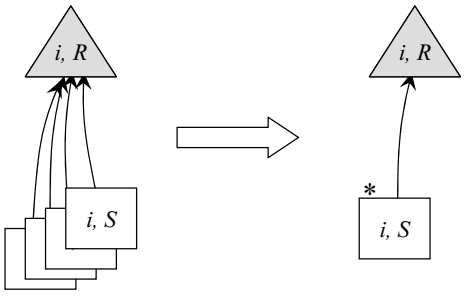


FIGURE 5.11: Grouping multiple intermediary nodes

We also note that if the destination node of a newly added viewlink edge does not exist, a NULL node is added to the graph according to the mapping function T. Graph (D) and Graph (E) in Figure 5.12 show the presence of NULL nodes.

By considering the symmetric cases of Figure 5.12, we have given in Figures 5.12 and 5.13 all possible graphs that can be produced after two assertors in an interaction finish recording their interaction record in the presence of failures. This summary will be used when we explore the properties of process documentation in the next section.

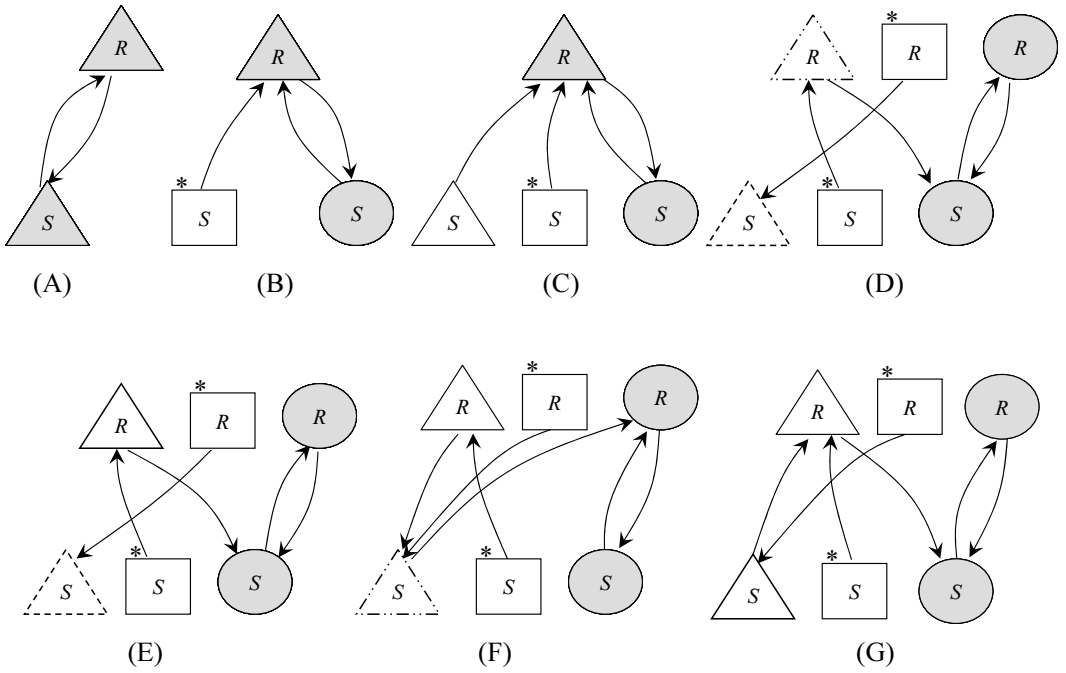


FIGURE 5.12: Graph summary ($PSSet(a, \kappa, v) \neq \emptyset$)

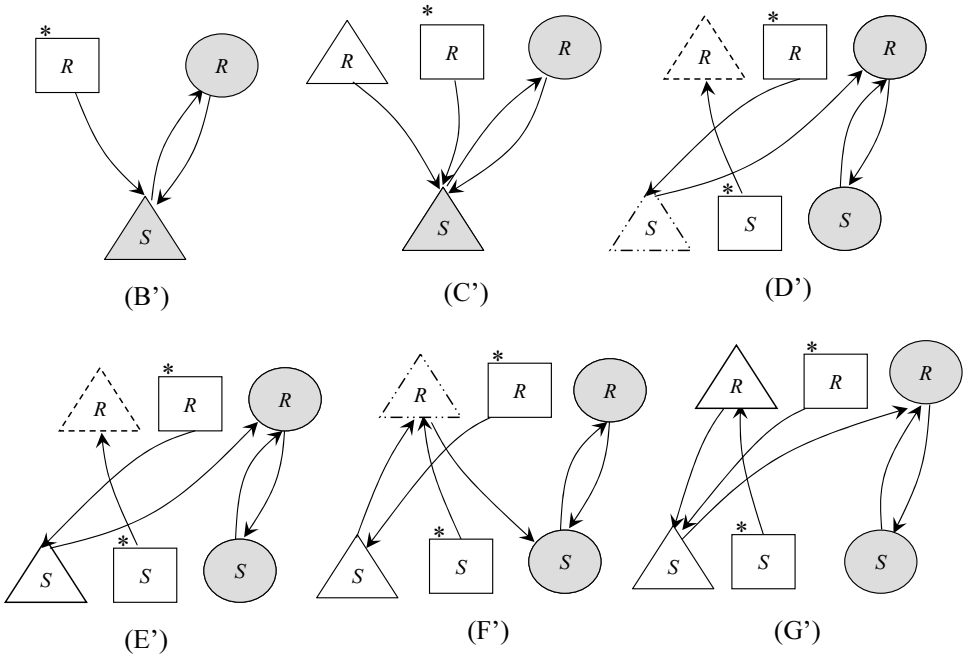


FIGURE 5.13: Graph summary (symmetric cases of Figure 5.12)

5.3.2 Causelink Edges

In this section, we analyse graphs containing causelink edges.

Figures 5.12 and 5.13 have summarised all possible graph topologies with regard to one interaction. By Lemmas 68 and 69, a causelink edge starts from a node in the sender's context and ends at the FINAL node in the receiver's context of another interaction.

We amend Figures 5.12 and 5.13 by adding causelink edges to all nodes (except LINK nodes[2]), which are in the sender's context[3]. By Lemma 92 (proved in Section 5.7), all causelink edges associated with the same cause interaction context $\langle \kappa, R \rangle$ share the same destination node.

We now illustrate the amending of Graph (D′) of Figure 5.13. We assume Graph (D′) is concerned with interaction j. For simplification, we also assume there is only one relationship p-assertion, which has only one cause interaction i and hence one causelink. Therefore, all the provenance stores duplicately recording the relationship p-assertion share a same causelink. The produced graph is shown in Figure 5.14, where there is only one causelink edge starting from any node (except the LINK node) in the sender's interaction context $\langle j, S \rangle$ and ending at a same destination node, which is a FINAL node in interaction context $\langle i, R \rangle$.

We have demonstrated the case where there is only one relationship p-assertion with only one cause interaction in an interaction record. We can easily extend the discussion to the case where there are multiple relationship p-assertions in an interaction record, each having multiple cause interactions.

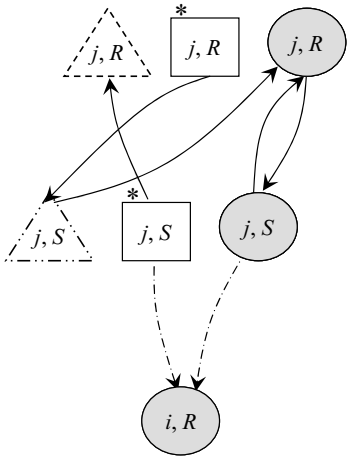


FIGURE 5.14: Graph (D′) in Figure 5.13 with causelink edges appended

Lemma 65 has stated that there is one FINAL node in each interaction context and Figures 5.12 and 5.13 have all graph topologies. Therefore, we can come to a general conclusion: *a causelink edge connects any two graphs in Figures 5.12 and 5.13, provided that the source node is either FULL or FINAL and the connection is compliant with Lemmas 68, 69 and 70.* Figure 5.15 gives an example.

[2] A LINK node reflects a provenance store which does not record any causelink.
[3] Only the sender of an interaction creates and records relationship p-assertions containing causelinks.

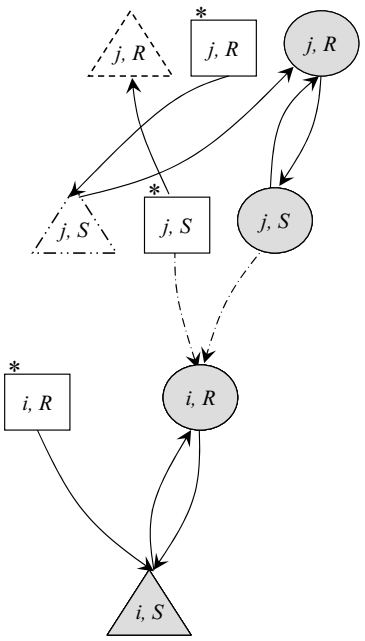


FIGURE 5.15: Graphs (B′) and (D′) in Figure 5.13 connected via causelink edges

5.4 Process Documentation Properties

In the previous sections, we discussed possible graph topologies after the two assertors in one interaction finish recording their interaction record in the presence of failures. This section investigates the properties of the whole process documentation.

Given that a process consists of a set of interactions, we use the induction on the interactions of a process to show the documentation of a whole process is guaranteed to be recorded in provenance stores and any links are accurate. In the inductive step, i.e., for each interaction of a process, we reuse the properties established for F-PReP.

5.4.1 Modeling Process

First of all, we extend the ASM to consider the execution of a process [76].

A process is a causally connected set of interactions between actors involved in that process.

To describe this execution formally, we also define the following state space [76] for the execution of the set of actors.

Since an actor behaves upon the receipt of an application message, we model the state of actor (AS) as data within received application messages, where messages are identified by interaction keys (IK). We also define a table AppS mapping actor identities (AID) to

$$
\begin{array}{rcll}
\text{AS} & = & \mathbb{P}(\text{DATA} \times \text{IK}) & \text{(Actor States)} \\
\textsc{AppS} & = & \text{AID} \rightarrow \text{AS} & \text{(Application States)} \\
\text{APC} & = & \textsc{AppS} \times \text{C} & \text{(Provenance-aware Application States)}
\end{array}
$$

Characteristic Variables:
$\langle d, \kappa \rangle \in \text{AS}$,
$as \in \textsc{AppS}$,
$apc \in \text{APC}$,
$c \in \text{C}$,
$\langle as, c \rangle = apc$

FIGURE 5.16: Extended ASM state space

actor states. The configuration of a provenance-aware application[4], modelled by APC, is defined by the combination of the state of all actors in the system combined with the configuration (C) of F-PReP[5]. The execution of actors following Figure 5.17 can be modeled by rules that express the transition of states when sending and receiving application messages[6].

$produce\_app(a_s, a_r, a_{ps}, d)$:
//triggered when d is to be sent by a_s to a_r.
$\rightarrow$ {
$\kappa \leftarrow newIdentifier(a_s, a_r)$;
$send(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
}

$consume\_app(a_s, a_r, a_{ps}, \kappa, d)$:
$\mathsf{app}(d, \kappa, a_{ps}) \in k(a_s, a_r)$
$\rightarrow$ {
$receive(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
$as(a_r) := as(a_r) \oplus \{\langle d, \kappa \rangle\}$;
// business logic
}

FIGURE 5.17: Application transitions

For simplification, we use *p_app* and *c_app* to denote transitions *produce_app* and *consume_app*, and use *s_app* and *r_app* to denote F-PReP's transitions *send_app* and *receive_app* (Figures 3.23 and 3.24).

With these definitions, the execution of an interaction can be denoted as

$$as_1 \longrightarrow_{p\_app} as_2 \longrightarrow_{c\_app} as_3.$$

[4]We term a provenance-aware application as an application that creates and records process documentation.

[5]This configuration is defined in the extended system state space in Figure 3.17.

[6]The application message contains an ownlink (a_{ps}), application data (d) and interaction key (κ) in order to be compatible with the definition in F-PReP.

For simplification, we express the above notation as

$$as_1 \longrightarrow_{(p\_app;\ c\_app)} as_3.$$

Given that a process consists of a set of interactions, the execution of a process[7] can be represented as

$$as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_f,$$

where as_0 and as_f are the initial state and final state of the assertors in the process, respectively.

We note that the sender in the first interaction of a process does not produce any relationship p-assertion since the first interaction is not caused by other interactions.

In a provenance-aware application, application transitions and F-PReP transitions are coupled together following Definition 72.

Definition 72. Let t be any application or F-PReP transition or a merged transition of both, for any configurations, $\langle as_1, c_1\rangle$ and $\langle as_2, c_2\rangle$, the following transition is allowed:

$$\langle as_1, c_1\rangle \longrightarrow_t \langle as_2, c_2\rangle$$

if one of the following conditions hold:

1. if $as_1 \longrightarrow_{p\_app} as_2$, then $c_1 \longrightarrow_{s\_app} c_2$.
2. if $as_1 \longrightarrow_{c\_app} as_2$, then $c_1 \longrightarrow_{r\_app} c_2$.
3. if $c_1 \longrightarrow_t c_2$, such that $t \neq s\_app$ and r_app, then $as_1 = as_2$.

□

When the *p_app* or *c_app* transition is fired in the application, the corresponding *s_app* or *r_app* transition is fired as well, which means the application and F-PReP transitions are merged together. We note that the pseudo-statements shared by the transitions only execute *once*.

Therefore, the execution of the application is coupled with the execution of F-PReP via corresponding rules for sending and receiving messages. Using this definition, Sections 5.4.2 and 5.4.3 analyse the properties of process documentation recorded after the provenance-aware application is complete.

[7] We assume that a process always starts from a transition *p_app*.

5.4.2 Guaranteed Recording, Viewlink Accuracy, Causelink Accuracy

In this section, we demonstrate that process documentation recorded using F-PReP is guaranteed to be recorded in provenance stores and we use graphical representation to show that all viewlinks and causelinks in the process documentation are accurate.

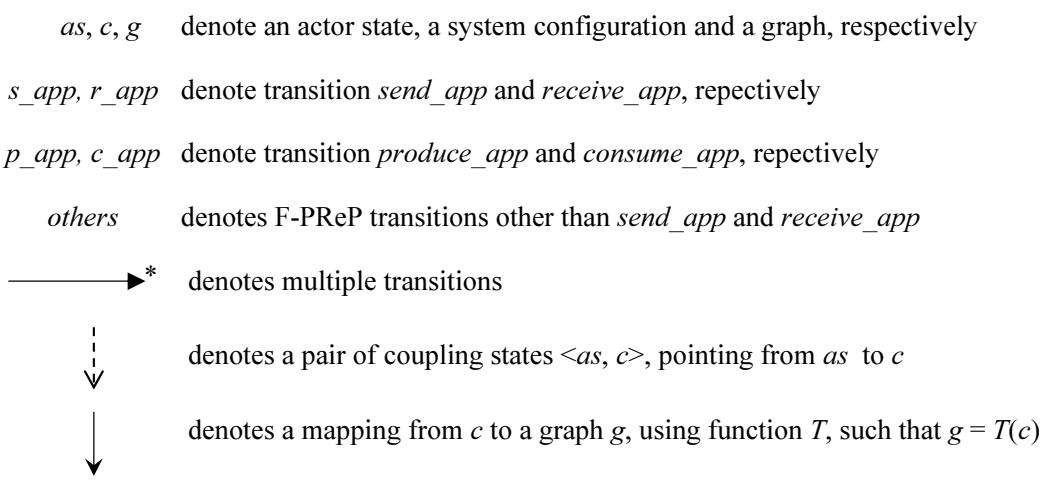


FIGURE 5.18: Transition notations

Our proof is based on the induction on the interactions of a process. To better illustrate the proof, we use three state transition diagrams shown in Figures 5.19, 5.21, and 5.22. These figures follow the notations defined in Figure 5.18.

Lemma 73. *As depicted in Figure 5.19, for any application state, as_j, reachable from initial state as_0, where $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$; for all apc_j reachable from apc_0: $apc_0 \longrightarrow^*_t apc_j$ with $apc_0 = \langle as_0, c_0\rangle$ and $apc_j = \langle as_j, c_j\rangle$; the following statements hold when using F-PReP to record process documentation in the presence of failures:*

1. *[Process Recording Termination] there exists a final configuration $apc_f = \langle as_j, c_f\rangle$, such that $apc_j \longrightarrow^*_t apc_f$ without application transitions;*
2. *[Guaranteed Recording] the documentation of the process, $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$, is recorded in provenance stores at c_f;*
3. *if graph g_f, produced by $T(c_f)$, is not empty, then g_f has the following properties:*

   (3.a) [Viewlink Accuracy] for any node $n \in g_f.n$, such that $n = \langle a_{ps}, \kappa, v\rangle$ and $g_f.nk(n) =$ FINAL, then there is only one viewlink edge $\langle n, n'\rangle \in g_f.ve$, where $n' = \langle a'_{ps}, \kappa, \overline{v}\rangle$ and $g_f.nk(n') =$ FINAL, for some a'_{ps}.

   (3.b) [Causelink Accuracy] for any node $n \in g_f.n$, such that $n = \langle a_{ps}, \kappa, S\rangle$ and $g_f.nk(n) =$ FINAL, then $|getCIDS(n)| = |\{\langle n, n'\rangle | \langle n, n'\rangle \in g_f.ce\}|$, where $n' = \langle a'_{ps}, \kappa', R\rangle$ and $g_f.nk(n') =$ FINAL and $\kappa' \neq \kappa$, for some a'_{ps}.

□

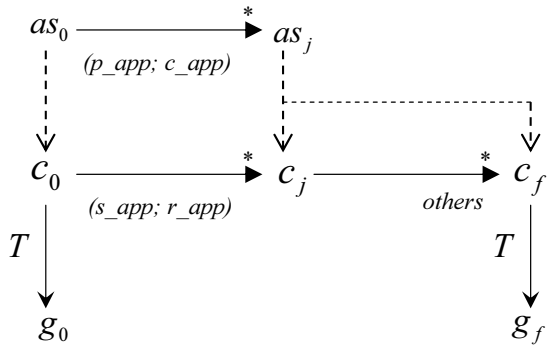
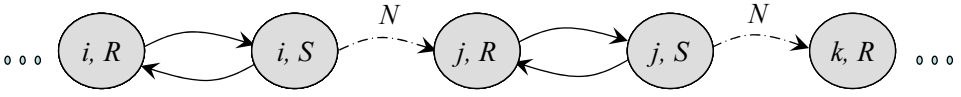


FIGURE 5.19: State transition diagram depicting Lemma 73

In Figure 5.19, the application proceeds from an initial state as_0 to some final state as_j after a serial of interactions. Because of system coupling, the provenance-aware application also proceeds from an initial state $\langle as_0, c_0\rangle$ to some state $\langle as_j, c_j\rangle$. However, there may be interaction records remaining to be recorded, thus the provenance-aware application finishes recording them without using application transitions (denoted by *others* in the figure). Finally, the provenance-aware application reaches final state $\langle as_j, c_f\rangle$. By $T(c_f)$, we can obtain graph g_f, which exhibits Properties (3.a) and (3.b).

Properties (3.a) and (3.b) state the topology of all FINAL nodes in g_f. Essentially, Properties (3.a) and (3.b) define the backbone of g_f, following a pattern as illustrated in Figure 5.20. In Figure 5.20, we assume each node is an alternative final node, which can be other type of node. We also use N to denote $|getCIDS(n)|$ and only show one destination node for N causelink edges to simplify the graph. In general, each causelink edge can have a different destination node.



i, *j*, *k*: interaction keys
N: the number of causelink edges, each pointing to another FINAL node.

For simplification, we omit the store's identity of a node and assume *N* causelink edges share a same destination node.

FIGURE 5.20: A pattern of FINAL nodes

PROOF. Our proof proceeds by induction on the length of the transition sequence from $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$.

In the base case, as_0 equals as_j, hence no interaction has taken place, and no application transition occurred, and process documentation is empty, and g_f is empty. Therefore, the lemma holds trivially.

In the inductive case, as shown in Figure 5.21, if $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_i$, then by Definition 72, we have $c_0 \longrightarrow^*_{(s\_app;\ r\_app)} c_i$. As inductive hypothesis, we assume that (1) there exists a final configuration $\langle as_i, c_x\rangle$, such that $\langle as_i, c_i\rangle \longrightarrow^*_t \langle as_i, c_x\rangle$ without application

transitions; (2) process documentation about $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_i$ was recorded in a set of provenance stores at c_x; (3) graph g_i, produced by $T(c_x)$, preserves the following properties:

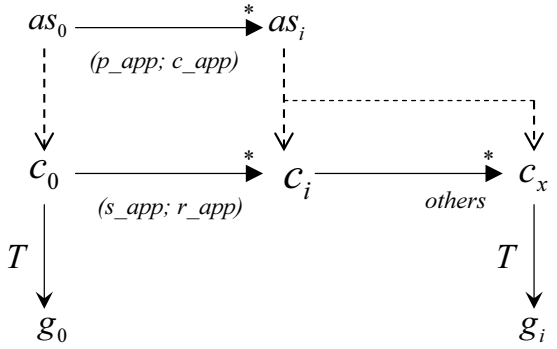


FIGURE 5.21: State transition diagram depicting the inductive hypothesis for proof of Lemma 73

- for any node $n \in g_i.n$, such that $n = \langle a_{ps}, \kappa, v \rangle$ and $g_i.nk(n) = \mathsf{FINAL}$, then there is only one viewlink edge $\langle n, n' \rangle \in g_i.ve$, where $n' = \langle a'_{ps}, \kappa, \overline{v} \rangle$ and $g_i.nk(n') = \mathsf{FINAL}$;
- for any node $n \in g_i.n$, such that $n = \langle a_{ps}, \kappa, S \rangle$ and $g_i.nk(n) = \mathsf{FINAL}$, there are $|getCIDS(n)|$ causelink edges $\langle n, n' \rangle \in g_i.ce$, where $n' = \langle a'_{ps}, \kappa', R \rangle$ and $g_i.nk(n') = \mathsf{FINAL}$ and $\kappa' \neq \kappa$.

We now consider the step $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$. By Definition 72, $c_x \longrightarrow_{(s\_app;\ r\_app)} c_j$. This inductive step is depicted in Figure 5.22. We note that this application transition can occur at any time after configuration $\langle as_i, c_i \rangle$. It does not have to wait for the recording of p-assertions to finish. We firstly establish that there exists a final configuration $\langle as_j, c_f \rangle$, where process documentation is recorded. Then we establish g_f's properties.

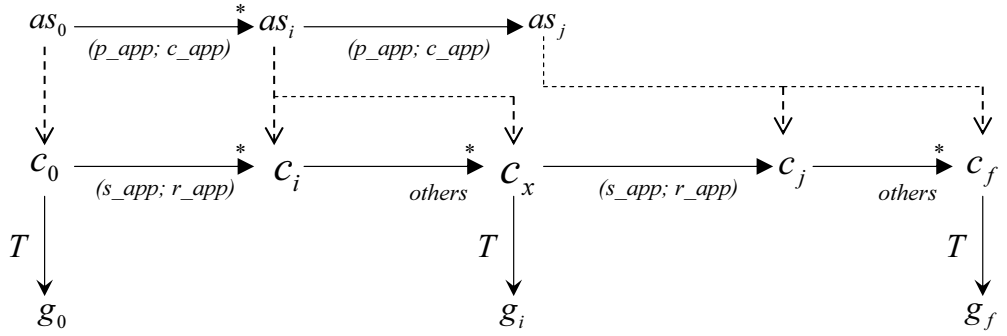


FIGURE 5.22: State transition diagram depicting the inductive step for proof of Lemma 73

(1) We establish the presence of a final configuration for the process. By inductive hypothesis, we know $\langle as_i, c_x \rangle$ is the final configuration for all prior interactions. Since $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$ is about one interaction, by the ASM's TERMINATION property (Theorem 15), there exists a final configuration for the interaction, where documentation

about this interaction is guaranteed to be recorded in provenance stores. This means that there are a finite number of transitions from c_j to a final system configuration c_f without using application transitions. Therefore, there exists a final configuration, $\langle as_j, c_f \rangle$ for the process $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$.

(2) By Theorem 22, the sender and receiver's interaction records about interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$ are guaranteed to be recorded at c_f. With inductive hypothesis, we know that documentation about prior interactions, $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_i$, has been recorded at c_x. Therefore, process documentation about $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$ is recorded at c_f.

(3) In the following properties, we establish the topology of distributed process documentation. Since $\langle as_i, c_x \rangle \longrightarrow^*_{paa} \langle as_j, c_f \rangle$ is about one interaction and the interaction key j is unique in the process, the subgraph g' with regard to this interaction is produced independently of the rest of the graph.

(3.a) By Lemma 65, g' contains two FINAL nodes, $n = \langle a_{ps}, j, v \rangle$ and $n' = \langle a'_{ps}, j, \overline{v} \rangle$, for some a_{ps} and a'_{ps}. By Lemma 67, $\langle n, n' \rangle \in g'.ve$ and $\langle n', n \rangle \in g'.ve$. Therefore, with inductive hypothesis, g_f preserves Property (3.a).

(3.b) By Lemma 65, one FINAL node n, $\langle a_{ps}, j, S \rangle$, is produced in the receiver's interaction context in g'. Since $g' \subseteq g_f$, we have $n \in g_f.n$. By Lemma 70, $|getCIDS(n)|$ causelink edges $\langle n, n' \rangle$ are added in $g_f.ce$, i.e., $|getCIDS(n)| = |\{\langle n, n' \rangle | \langle n, n' \rangle \in g_f.ce\}|$. By Lemmas 68 and 69, for each of the newly added causelink edges, we have $n' = \langle a'_{ps}, i, R \rangle$ and $g_f.nk(n') = \text{FINAL}$, such that $i \neq j$. Therefore, with inductive hypothesis, g_f preserves Property (3.b).

□

Essentially, (3.a) states that any two FINAL nodes in the opposite views with regard to a same interaction are interlinked via viewlink edges; (3.b) states that any FINAL node in the sender's context in an interaction connects to the FINAL node in the receiver's context of another interaction. By applying the two conclusions to all FINAL nodes in a graph, the pattern in Figure 5.20 appears.

This pattern has been exemplified by the graph in Figure 5.6 in Section 5.1.3, which is produced after F-PReP finishes recording process documentation in a failure-free environment. Recall that Section 5.3.2 has drawn a conclusion: *a causelink edge connects any two graphs in Figures 5.12 and 5.13, provided that the source node is either FULL or FINAL, and the connection is compliant with Lemmas 68, 69 and 70.* With this conclusion, we give an example of a graph (Figure 5.23) produced after F-PReP recorded process documentation in the presence of failures.

We note that there are many redundant nodes (FULL and LINK nodes) on the graph (Figure 5.23), indicating garbage information (i.e., duplicate or useless documentation)

recorded in provenance stores. As will be shown in the next section, redundant nodes are not on the path of retrieving process documentation. Therefore, the garbage information can be safely removed from stores, which we will discuss in Section 5.5.2.

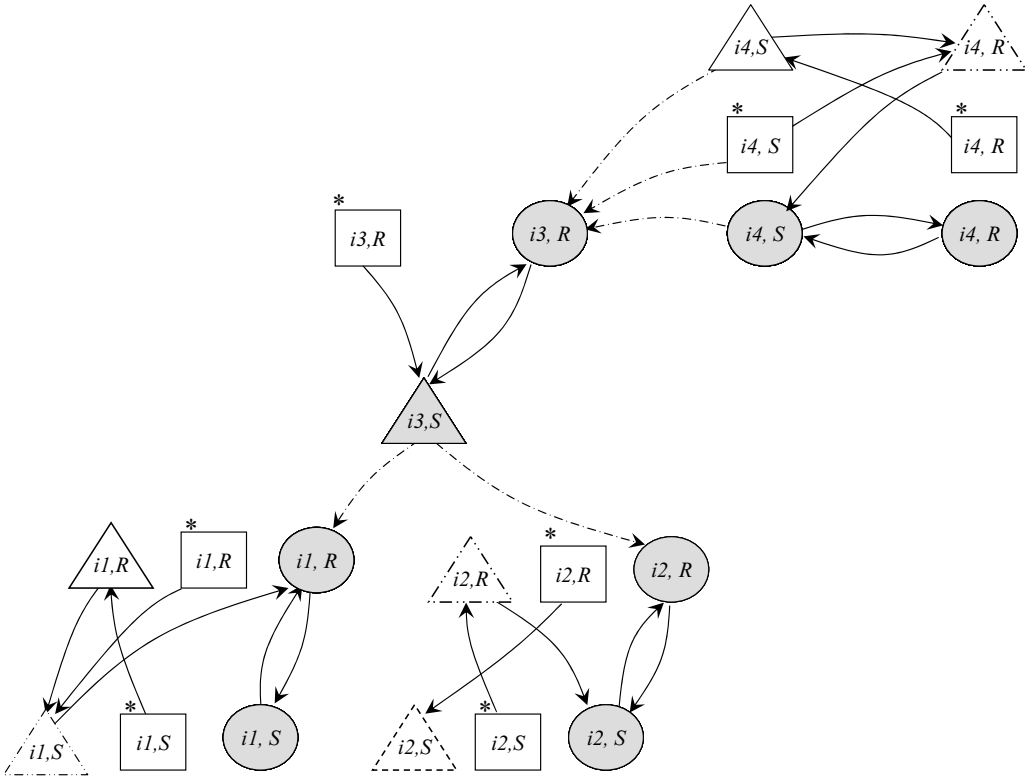


FIGURE 5.23: One possible graph produced after F-PReP finished recording in the presence of failures

After establishing that process documentation recorded using F-PReP is guaranteed to be recorded in provenance stores and all viewlinks and causelinks in the process documentation are accurate, we now demonstrate the entire retrievability of distributed documentation.

5.4.3 Documentation Retrievability

In this section, we refer to Figures 5.19, 5.21 and 5.22 when illustrating the proof.

5.4.3.1 Original Query Algorithm

A query algorithm has been developed [117] to retrieve process documentation which was recorded in a failure-free environment. We revise this algorithm in function *getProDoc*.

Function *getProDoc* takes a node n and a final graph g_f as input and returns the process documentation of a data item. We assume the data item is d_j, produced in process $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$ (Figure 5.19). We assume that *this function starts searching*

from the node in the receiver's interaction context $\langle j, R\rangle$ [8]. Therefore, a starting node n, $\langle a_{ps}, j, R\rangle$, is assumed as the initial input of the function, where a_{ps} is the default store used in interaction context $\langle j, R\rangle$.

Definition
$getProDoc : NODE \times GRAPH \rightarrow \mathbb{P}(RECORD)$
$getProDoc(n, g)$:
    initialize doc;
    $doc := doc \cup \{store\_T(n)\}$;
    if $(\langle n, n'\rangle \in g.ve)$
        $doc := doc \cup \{store\_T(n')\}$;
        for each $\langle n', n''\rangle \in g.ce$
            $doc := doc \cup getProDoc(n'', g)$;
    return doc;

Although the original query algorithm [117] does not involve g_f, we include it in order to simplify the presentation of *getProDoc*. For example, when dealing with causelinks, we do not need to express low level details such as the content of a relationship p-assertion, which has been captured by the definition of Causelink Edge.

Function *getProDoc* recursively retrieves interaction records from both views of each interaction of the process that led to d_j. The destination node of a viewlink edge or causelink edge indicates the next location to retrieve an interaction record. According to the definition of a node, notation $store\_T(n)$ is equivalent to $store\_T(a_{ps}, j, R)$, which simplifies our presentation.

To facilitate our further discussion, we define a term Directed Causal Path.

Definition 74 (Directed Causal Path)**.** *A directed causal path consists of a sequence of vertices (i.e., nodes),* n_1, n_2, …, n_m, *such that for* $i = 1, 2, \ldots, m-1$ *and for some* a_{ps} *and* κ, *the following conditions are satisfied:*

$$\bigvee \begin{cases} \langle n_i, n_{i+1}\rangle \in g.ce \\ \langle n_i, n_{i+1}\rangle \in g.ve \wedge n_i = \langle a_{ps}, \kappa, R\rangle \end{cases}$$

□

On a directed causal path, any two adjacent nodes are connected via a causelink edge or viewlink edge, representing links recorded in provenance stores. A causelink edge connects two nodes that can tell us where to find interaction records documenting an application's output data produced from an input data. A viewlink edge $\langle n_i, n_{i+1}\rangle \in g.ve$, where n_i is associated with a receiver's interaction context, connects two nodes that

[8] Similar discussion can be easily extended to the case of the sender, which we omit.

can help us to locate the interaction record documenting where a received input data was from, which was the output data produced by the sender in another interaction. Therefore, by following the directed causal paths, we can retrieve distributed process documentation describing how a data product was derived.

We note that by Figure 5.20, a viewlink edge $\langle n_{i+1}, n_i \rangle \in g.ve$, where $n_i = \langle a_{ps}, \kappa, R \rangle$, may exist but is not on the directed causal path. The presence of such a direction supports functionalities such as verifying two views' knowledge about the same interaction [76].

Essentially, *getProDoc* traverses by following any directed causal paths starting from a root node, say $\langle a_{ps}, j, R \rangle$. The retrieved process documentation has a form of tree structure with the root being the interaction record about interaction context $\langle j, R \rangle$, which is the first interaction record to be retrieved. In the example of Figure 5.5, let n be $\langle a_{ps}, i_4, R \rangle$, where a_{ps} is the default store used by assertor a_5 and g_f is the graph on Figure 5.6. Function $getProDoc(n, g_f)$ returns the entire process documentation and its retrieving path is shown on Figure 5.24. Since all default provenance stores successfully record interaction records in a failure-free environment, the nodes that $getProDoc(n, g_f)$ visits are the default nodes of each interaction, which are all FINAL.

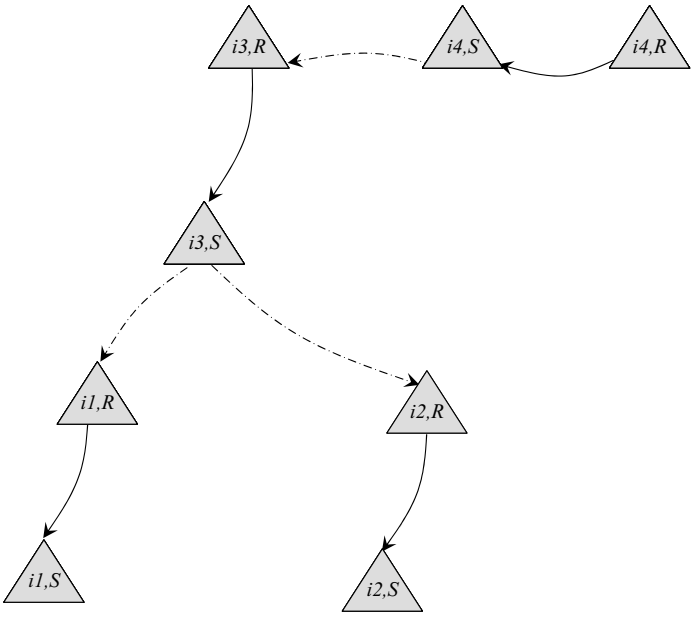


FIGURE 5.24: Retrieving path based on Figure 5.6

5.4.3.2 Retrieval Function Properties

The termination and completion properties of the original query algorithm [117] has not been formally established. We now establish them based on function *getProDoc*, which is crucial to establishing the retrievability of process documentation. We firstly identify Lemma 75 showing that the graph is acyclic.

Lemma 75 (Directed Acyclic Graph)**.** *Process graph has no directed causal path that starts and ends on the same node.* □

Proof. Our proof proceeds by induction on the length of the transition sequence from $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$, depicted in Figure 5.19. In the base case, as_0 equals as_j, hence no interaction has taken place, and g_f is empty. Therefore, the lemma holds trivially.

In the inductive case, as shown in Figure 5.21, if $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_i$, then the inductive hypothesis states that graph g_i has no directed causal path that starts and ends on the same node.

We now consider interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$ (Figure 5.22). Let n_j be any of the two nodes produced regarding interaction j. Since an interaction key is globally unique, j was not used in process $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_i$ and hence $n_j \notin g_i$.

We discuss the two nodes added into g_i to form g_f.

1. $n_j = \langle a_{ps}, j, S\rangle$:
The only permitted directed causal paths starts by following causelink edges from this node, which lead to nodes in g_i.

2. $n_j = \langle a_{ps}, j, R\rangle$:
The only permitted directed causal path starts by following a viewlink edge $\langle n_j, n'_j\rangle \in g_f.ve$, where $n'_j = \langle a'_{ps}, j, S\rangle$.

Finally, given $n_j \notin g_i$, there is no edge starting from a node in g_i and ending at n_j (Lemma 71). Therefore, for any node n_j regarding interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$, there is no directed causal path that starts and ends on the same n_j.

With inductive case, we have shown that g_f has no directed causal path that starts and ends on the same node, which preserves the statement.

□

Lemma 76 (Retrieval Termination)**.** *Function getProDoc terminates.* □

Proof. We proceed by the following reasoning. By definition, *getProDoc* retrieves interaction records by following directed causal paths from a starting node. With Lemma 75, *getProDoc* does not infinitely retrieves a same node. Since the first interaction in the process is not caused by other interactions, the recursion terminates when there is no further cause interaction. Therefore, function *getProDoc* terminates.

□

Next, we establish that function *getProDoc* returns entire process documentation.

Definition 77 (Entire Process Documentation)**.** *The entire documentation of a process consists of a set of interaction records about all interaction contexts of the process, and each interaction record is about a unique interaction context.*

Recall that an interaction context describes a view (sender or receiver) of an interaction and an interaction record is created by the sender or receiver describing its view of the interaction. Therefore, interaction records about all interaction contexts of a process are made by both the sender and receiver of each interaction of the whole process. If each interaction record is about a unique interaction context, then any two interaction records are not about the same interaction context, which means process documentation has no redundant interaction record and only contains the necessary information in order to determine the provenance of a data item.

In the example of Graph (E) in Figure 5.12, all the nodes in the receiver's context indicate an interaction record documenting the same interaction. So entire process documentation contains only one interaction record from the receiver's context of that interaction but the interaction record may be obtained from any node of the receiver's context.

Lemma 78 (Complete Retrieval)**.** *After F-PReP finishes recording process documentation in a failure-free environment, function getProDoc retrieves entire process documentation.* □

Proof. We proceed with the following reasoning. Since *getProDoc* retrieves process documentation recorded in a failure-free environment, the starting node it begins with is a default node, which is FINAL. By Lemma 65, there is only one FINAL node in each interaction context and by Lemma 76, the function terminates.

The following proof is based on the induction on the length of the transition sequence from $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$ in Figure 5.19.

In the base case, as_0 equals as_j, hence no interaction has taken place and process documentation is empty. Therefore, the lemma holds trivially. In the inductive case, the inductive hypothesis assumes that after F-PReP finishes recording process documentation about process $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_i$ (Figure 5.21) in a failure-free environment, $getProcDoc(n_i, g_i)$ returns entire process documentation about this process.

We now consider interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$ (Figure 5.22). Let n_j be the FINAL node in interaction context $\langle j, R\rangle$. After F-PReP finishes recording documentation for the process $as_0 \longrightarrow^*_{(p\_app;\ c\_app)} as_j$, we study $getProDoc(n_j, g_f)$. The interaction record in context $\langle j, R\rangle$ is first retrieved. Given $\langle n, n'\rangle \in g_f.ve$ and Lemma 73 (3.a), n' is the FINAL node in context $\langle j, S\rangle$. Hence, the function retrieves one interaction record from each interaction context of interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$.

By Lemma 73 (3.b), there are $|getCIDS(n')|$ causelink edges, each pointing to a destination node n'' that indicates the location where the interaction record of the receiver in the corresponding cause interaction is recorded. Assuming that the interaction key of a cause interaction is i, by Lemma 73 (3.b), we know that n'' is the FINAL node in interaction context $\langle i, R\rangle$ and $n'' \in g_i.n$. Therefore, according to the inductive case, $getProcDoc(\mathrm{n}'', \mathrm{g}_i)$ gives entire process documentation for previous interactions other than $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$. Since we have retrieved one interaction record from each interaction context of interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$, $getProcDoc(n_j, g_f)$ gives entire process documentation.

Based on the above proof, the implication is preserved.

□

5.4.3.3 New Query Function

The original query function *getProDoc* is used to retrieve process documentation recorded when no failure occurred. Hence it begins by searching an assertor's default store. However, it would retrieve incomplete process documentation if the documentation was recorded in the presence of failures, as illustrated in Figure 5.25.

Figure 5.25 shows *getProDoc*'s retrieving path based on the graph in Figure 5.23, where alternative stores were used to record the interaction records. It starts searching from the default node in interaction context $\langle i4, R\rangle$, which is LINK, referring to a provenance store which does not have an interaction record but only contains a viewlink for that context. Therefore, retrieving interaction records from the default store in this case cannot give entire process documentation.

To address this problem, we introduce a new query function $getProDoc\_New$ to retrieve process documentation recorded in the presence of failures.

Definition
$getProDoc\_New : \mathbb{P}(PID) \times IK \times GRAPH \rightarrow \mathbb{P}(RECORD)$
$getProDoc\_New(pslist, j, g_f)$:
  for *each* $a_{ps} \in pslist$
    do $n \leftarrow \langle a_{ps}, j, R\rangle$;
      if $(\langle n, n'\rangle \in g_f.ve \wedge store\_T(n).pas \neq \emptyset \wedge store\_T(n').pas \neq \emptyset)$
        return $getProDoc(n, g_f)$;
  return $\emptyset$;

Function $getProDoc\_New$ takes as input a list of provenance store identities (e.g., an assertor's alternative stores), an interaction key representing the interaction in which a

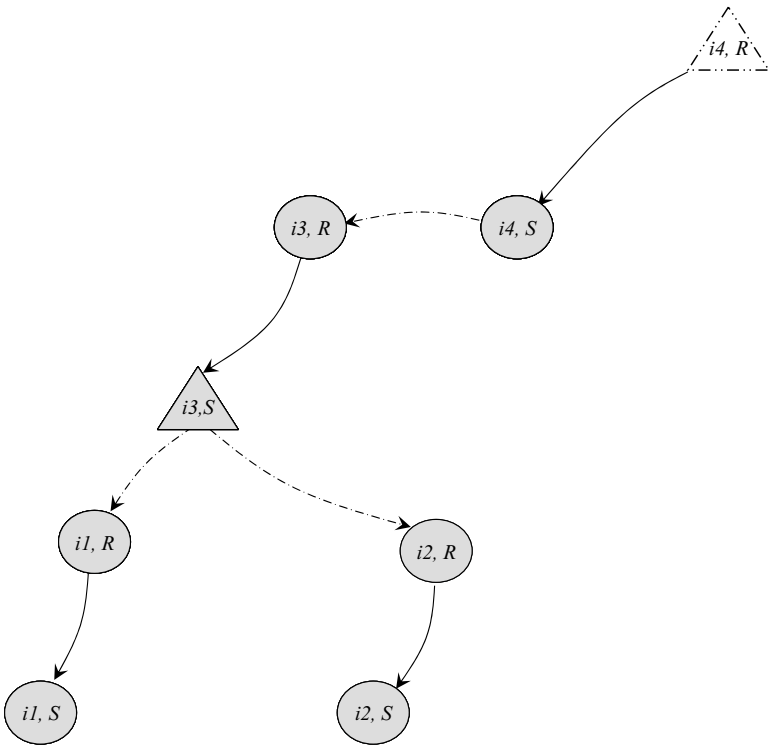


FIGURE 5.25: Retrieving path based on Figure 5.23, using *getProDoc*

data item was exchanged, and a final graph. It then returns the process documentation of the data item. Since an interaction record may be recorded in an alternative store when failures occurred, function *getProDoc_New* searches all the candidate stores to look for the first interaction record of the process documentation and call function *getProDoc*.

In order to call *getProDoc*, a starting node $\langle a_{ps}, j, R\rangle$[9] is required, which must satisfy two conditions. Firstly, there exists another node n' such that $\langle n, n'\rangle \in g_f.ve$. Secondly, $store\_T(n).pas \neq \emptyset$ and $store\_T(n').pas \neq \emptyset$, which implies that n and n' must be FULL or FINAL. By Lemma 36, if $store\_T(n).pas \neq \emptyset$, then $store\_T(n) = \langle a, vl, assertor\_T(a, \kappa, v).pas\rangle$ and hence the corresponding node is FULL or FINAL.

As opposed to *getProDoc*, *getProDoc_New* starts searching from any candidate provenance store and returns entire process documentation. In the case of Figure 5.23, it searches all the alternative stores as well as the default store of assertor a_5. Its traversing path is given in Figure 5.26, where the first node leading to entire process documentation is an intermediary node in context $\langle i4, R\rangle$.

Now, we are ready to establish property DOCUMENTATION RETRIEVABILITY.

[9]Like *getProDoc*, this function is also assumed to retrieve the interaction record about a receiver's interaction context at the beginning of the search. Similar discussion can be easily extended to the case of the sender, which we omit.

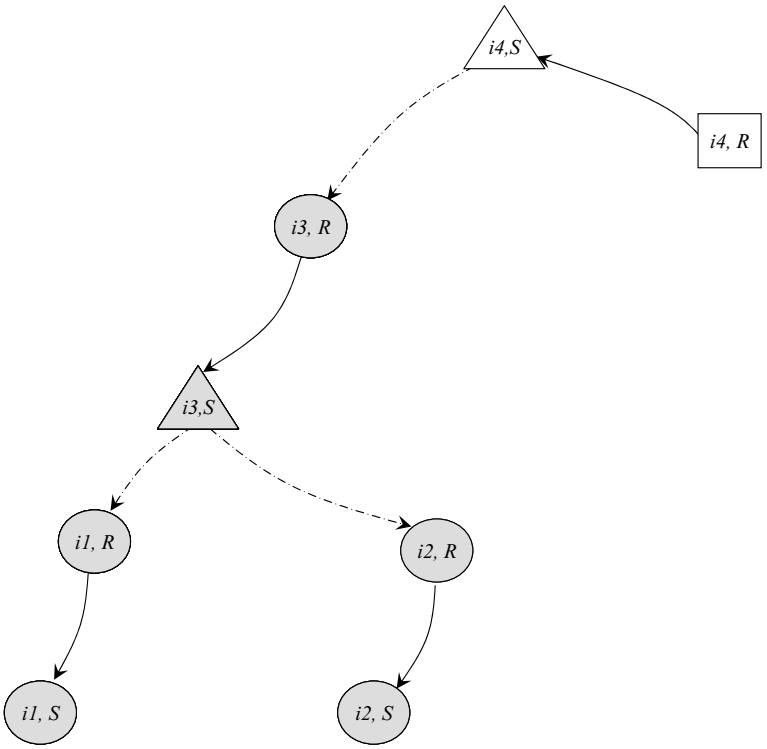


FIGURE 5.26: Retrieving path based on Figure 5.23, using *getProDoc_New*

5.4.3.4 Documentation Retrievability

Property DOCUMENTATION RETRIEVABILITY specifies that process documentation's entire retrievability can be ensured after being recorded in provenance stores.

Lemma 79 (DOCUMENTATION RETRIEVABILITY)**.** *Process documentation recorded by using F-PReP is retrievable in its entirety.* □

PROOF. To facilitate our description, we refer to Figure 5.22 in our proof, where interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$ is represented by interaction key j. By Lemma 78, this lemma is preserved for documentation recorded when no failure occurred. To demonstrate that process documentation is retrievable after being recorded in the presence of failures, we discuss two cases. In Case (1), we assume that the FINAL node n_j of interaction context $\langle j, R\rangle$ can be determined in advance. Since the knowledge about a FINAL node is global and may not always be available[10], we relax this assumption in Case (2), where we use the new query function *getProDoc_New* to facilitate the proof. Function *getProDoc_New* does not assume the global knowledge in the sense that it only checks the content of a provenance store ($store\_T(n).pas \neq \emptyset \wedge store\_T(n').pas \neq \emptyset$).

Case (1):
If a starting FINAL node n_j can be determined, this case is the same as retrieving

[10]Identifying a FINAL node requires assertor's state information as well as provenance store's state information, according to the definition of FINAL node.

process documentation recorded in a failure-free environment. The proof for Lemma 78 has shown that function $getProDoc(n_j, g_f)$ returns the entire process documentation. Therefore, the statement is preserved in this case.

Case (2):
This case uses $getProDoc\_New$ to obtain process documentation recorded in the presence of failures. We study $getProDoc\_New(pslist, j, g_f)$, where $pslist$ contains an asserter's default store and all alternative stores used for recording interaction record about context $\langle j, R\rangle$. Since each store is reflected by a node on the graph, function $getProDoc\_New(pslist, j, g_f)$ checks an assertor's default node, intermediary nodes and alternative final node, and finds the first node that satisfies the above two conditions (Section 5.4.3.3) in order to call $getProDoc$.

The rest of the proof is based on a case analysis of all possible graphs (A to G′) in Figures 5.12 and 5.13. We show that the starting node satisfying the two conditions leads to the retrieval of entire process documentation. Since any alternative final node in context $\langle j, R\rangle$ is always FINAL, as shown in Figures 5.12 and 5.13, it satisfies the two conditions and leads to entire process documentation, demonstrated in Case (1). Therefore, we only discuss the default node and intermediary nodes on each graph.

Graph (A), Graph (B) and Graph (C):
The only node in context $\langle j, R\rangle$ is the default node, which is FINAL. It satisfies the two above conditions and is passed to $getProDoc$. With the conclusion from Case (1), searching from this node leads to entire process documentation.

Graph (D):
The default node in context $\langle j, R\rangle$ is LINK, which refers to a provenance store only maintaining a viewlink for context $\langle j, R\rangle$. Hence the two conditions are not met and $getProDoc\_New(pslist, j, g_f)$ will check the next node.

All the intermediary nodes in context $\langle j, R\rangle$ each have a viewlink edge leading to a NULL node, therefore, the two conditions are not met and $getProDoc\_New(pslist, j, g_f)$ will check the next node.

In this case, searching from the alternative final node gives entire process documentation, which is returned by calling $getProDoc\_New(pslist, j, g_f)$.

Graph (E):
The default node in context $\langle j, R\rangle$ is FULL. It has a viewlink edge leading to a FINAL node. When searching from this node, $getProDoc(n, g_f)$ retrieves one interaction record from each interaction context of interaction j in the first iteration. Since the default node indicates a duplicate interaction record maintained in the corresponding default store, $getProDoc(n, g_f)$ retrieves the same interaction record as from the FINAL node in context $\langle j, R\rangle$. By property (3.b) in Lemma 73, each causelink edge starting from the FINAL node in context $\langle j, S\rangle$ ends at another FINAL node. Therefore, from the second

iteration, all the input nodes for $getProDoc$ are FINAL. Then the conclusion from Case (1) applies. Based on the above analysis, searching from the default node, which is not FINAL though, still leads to entire process documentation.

Each intermediary node in context $\langle j, R\rangle$ connects to a NULL node via a viewlink edge, therefore, the two conditions are not met and $getProDoc\_New(pslist, j, g_f)$ will check the next node.

Graph (F):
The default node and intermediary nodes in context $\langle j, R\rangle$ are FULL. Each has a viewlink edge leading to a LINK node. Therefore, the two conditions are not met and $getProDoc\_New(pslist, j, g_f)$ will check the next node.

Graph (G):
Similar to the discussion for Graph (E), searching from the default node in context $\langle j, R\rangle$ gives entire process documentation.

Each intermediary node in context $\langle j, R\rangle$ is FULL and connects to another FULL node via a viewlink edge. When searching from an intermediary node, $getProDoc(n, g_f)$ retrieves one interaction record about each view of interaction j in the first iteration. Since a FULL node indicates duplicate interaction record maintained in the corresponding store, $getProDoc(n, g_f)$ retrieves the same interaction records as from the FINAL nodes regarding context $\langle j, R\rangle$ and $\langle j, S\rangle$. In addition, the recording of duplicate interaction record also implies that causelink edges from a FULL node are the same as from the FINAL nodes in context $\langle j, S\rangle$. By property (3.b) in Lemma 73, each causelink edge starting from the FULL node in context $\langle j, S\rangle$ ends at another FINAL node. Therefore, all the input nodes for $getProDoc$ are FINAL from the second iteration and the conclusion in Case (1) applies. Based on the above analysis, searching from any intermediary node, which is not FINAL though, still gives us entire process documentation.

Graphs (B′) to (G′):
The analysis is similar to the above, which we omit.

After discussing all the possible graphs regarding interaction $as_i \longrightarrow_{(p\_app;\ c\_app)} as_j$, we have shown that $getProDoc\_New$ returns entire process documentation by searching from any node n which satisfies two conditions: (1) there exists another node n' such that $\langle n, n'\rangle \in g_f.ve$; (2) $store\_T(n).pas \neq \emptyset$ and $store\_T(n').pas \neq \emptyset$, which implies that n and n' must be FULL or FINAL.

With the conclusion of Case (1) and Case (2), the implication is preserved.

□

Lemma 79 has established that the entire retrievability of process documentation recorded by using F-PReP can be ensured.

5.5 Discussion

In this section, we first discuss retrieving process documentation without knowing the termination of the protocol. We then discuss how to remove or reduce garbage information reflected by LINK and FULL nodes.

5.5.1 Early Retrieval

Due to the asynchronous nature of the protocol, process documentation cannot be retrieved in its entirety until the F-PReP terminates for each interaction of the process. For example, if the coordinator still has outstanding update requests, entire process documentation cannot be retrieved. Although this is not a problem specific about failures[11], we provide initial thoughts about it.

As the termination of the protocol may not be known to a querying actor, the actor has to wait a reasonable period of time after the data result is produced. Then it can repeatedly query candidate provenance stores until it obtains the entire process documentation of the result. To validate the entirety of retrieved process documentation, the actor can either use application specific knowledge or rely on mechanisms provided by a provenance store to verify the completeness. These mechanisms are however beyond the scope of this thesis.

5.5.2 Dealing with Garbage Information

The traversal function $getProDoc\_New$ can be used to identify the "primary" records; the ones that are not traversed are regarded as garbage. In Figure 5.23, LINK and FULL nodes reflect garbage information (a single viewlink and a duplicate interaction record, respectively) recorded in a respective provenance store. The presence of a LINK node is due to the coordinator's incomplete knowledge when updating viewlink in a provenance store, whilst a FULL node appears because of the use of alternative stores.

Garbage information reflected by LINK nodes can be safely removed after a querying actor obtains entire process documentation. In practice, a user can remove all the single viewlinks recorded in provenance stores. Formally, for any a_{ps}, κ and v and when the ASM terminates, if $store\_T(a_{ps}, \kappa, v) = \langle \bot, vl, \emptyset \rangle$, then the viewlink vl is ready to be removed.

In terms of FULL nodes, the protocol can be configured to resend messages to a same provenance store for a certain number of times before using an alternative store. Therefore, the probability of recording redundant information can be reduced. After a query-

[11] We also cannot obtain entire process documentation in the absence of failures until the application (i.e., process) terminates and process documentation ends up in provenance stores.

ing actor obtains entire process documentation by following nodes on the retrieving path, it can remove redundant information recorded in provenance stores indicated by nodes not on the retrieving path. However, this requires a robust garbage collection protocol to ensure the safe removal of redundant information, which is out of the scope of this thesis.

5.6 Conclusion

This chapter investigated the properties of process documentation recorded using F-PReP in the presence of failures. We showed that the documentation of a whole process is guaranteed to be recorded in provenance stores and any links are accurate. More importantly, process documentation recorded in multiple interlinked provenance stores can be retrieved in its entirety.

At the beginning of this chapter, we introduced a graph-based formalism (Section 5.1.1) and then investigated the topological properties. Since a process consists of a set of interactions, we exhaustively identified several graph properties regarding one interaction (Sections 5.2 and 5.3). When analysing the properties of process documentation, we performed induction on the interactions of a process to show the documentation of a whole process is guaranteed to be recorded in provenance stores and any links are accurate. In the inductive step, i.e., for each interaction of a process, we reused the properties established in Sections 5.2 and 5.3. In addition, we developed a new query function for retrieving process documentation recorded in the presence of failures. We established that process documentation can be represented as a directed acyclic graph and the query function eventually terminates for retrieving process documentation. Finally, we showed that the new query function ensures the retrieved process documentation is in its entirety.

The appendix in Section 5.7 provides proofs for several properties regarding the topological relationship between an assertor's default store and alternative stores. These properties have been used in Section 5.3.

Next chapter will evaluate F-PReP's performance and illustrate how the implementation of the protocol addresses non-functional requirement Efficient Recording identified in Chapter 3.

5.7 Appendix: Proof of Topology Properties

Provenance stores used by an assertor have been classified as two types: the default store (i.e., the one that an assertor initially used when recording an interaction record) and alternative stores. When the protocol terminates, there is a final store that an assertor

knows to have successfully recorded its interaction records. The final store can be the default store or an alternative store.

Chapter 4 has established properties to demonstrate that F-PReP meets requirements CAUSELINK ACCURACY and VIEWLINK ACCURACY. These properties are only concerned with the final store.

Recall that an assertor sets a timeout when submitting an interaction record to a store. Since it is impossible to distinguish store crash from message loss in the event of a timeout, it may be the case that a provenance store has recorded an interaction record while the assertor has a timeout event and has to choose an alternative store. Therefore, the default store and any of the attempted alternative stores may possess duplicate interaction records, which means there would be viewlinks or causelinks recorded in stores.

This appendix provides proofs for several properties regarding the topological relationship between an assertor's default store and alternative stores. These properties have been used in Section 5.3.

We divide our discussion into two parts. We firstly discuss provenance stores interlinked via viewlinks (from Lemma 80 to Lemma 91) and then discuss those linked via causelinks (Lemma 92). Given notations a, vl and pas, we assume that $a \neq \perp$, $vl \neq \perp$ and $pas \neq \emptyset$ for simplification.

Lemma 80 establishes that if each assertor's default store records the assertor's interaction record, then the assertor's viewlink recorded in the default store points to the same store as referred by the other assertor's ownlink given that the other assertor did not use an alternative store.

Lemma 80. *For any a, κ and v, and for some a' and a_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$, then the following implication holds when the ASM terminates at final configuration:*

If

$$\bigwedge \begin{cases} store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ log\_T(a', \kappa, \overline{v}) = \mathit{FALSE} \end{cases}$$

then

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

□

PROOF. We proceed by the following reasoning.

Given that $log\_T(a', \kappa, \overline{v})$ is FALSE, by Lemma 32,

$$assertor\_T(a, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol. \tag{5.1}$$

Also by Lemma 33, $coord\_T(a_c, \kappa, \overline{v}) = \langle \bot, \bot \rangle$. Then by Lemma 38, $vlstate\_T(a_{ps}, \kappa, v)$ is not UPDATED.

Given that $store\_T(a_{ps}, \kappa, v)$ records an entry, $vlstate\_T(a_{ps}, \kappa, v)$ is either DEFAULT or UPDATED (Lemma 40). Therefore, $vlstate\_T(a_{ps}, \kappa, v)$ can only be DEFAULT. Then with Lemma 39,

$$store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a, \kappa, v).vl. \tag{5.2}$$

Therefore, with (5.1), (5.2), the statement is preserved.

□

Lemma 81 links the state of tables $store\_T(a_{ps}, \kappa, v)$ and $vlstate\_T(a_{ps}, \kappa, v)$.

Lemma 81. *For any reachable configuration, for any a_{ps}, κ, v and vl, and for some a, the following implication holds:*

If $store\_T(a_{ps}, \kappa, v) \neq \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$, then

$$vlstate\_T(a_{ps}, \kappa, v) \neq \mathit{DEFAULT}.$$

□

Proof. We proceed with the following reasoning. The statement holds in the initial configuration since tables are all empty. We now consider only those transitions that may have an effect on terms in the equality.

receive_record:
After this transition, $store\_T(a_{ps}, \kappa, v)$ becomes $\langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$. Therefore, the antecedent is false, which preserves the implication.

receive_update:
By Lemma 36, $store\_T(a_{ps}, \kappa, v)$ has three states before the transition: $\langle \bot, \bot, \emptyset \rangle$ and $\langle \bot, vl, \emptyset \rangle$ and $\langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$.

In the case of $\langle \bot, \bot, \emptyset \rangle$:
After this transition, $store\_T(a_{ps}, \kappa, v)$ becomes $\langle \bot, vl, \emptyset \rangle$ and $vlstate\_T(a_{ps}, \kappa, v)$ is changed to UPDATED. Therefore, the statement is preserved.

In the case of $\langle \bot, vl, \emptyset \rangle$:
After this transition, $store\_T(a_{ps}, \kappa, v).vl$ is updated and $store\_T(a_{ps}, \kappa, v)$ remains as

$\langle \perp, vl, \emptyset \rangle$. In addition, $vlstate\_T(a_{ps}, \kappa, v)$ becomes UPDATED. Therefore, the statement is preserved.

□

Lemma 82 states a possible topology of interlinked provenance stores in the case where one assertor did not use any alternative store while the other assertor did and the other assertor's default store did not record any duplicate interaction record. The property is used by Case (b), Section 5.3.1.1.

Lemma 82. *For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, the following implication holds when the ASM terminates at final configuration:*

If

$$\bigwedge \begin{cases} log\_T(a, \kappa, v) = FALSE \\ log\_T(a', \kappa, \overline{v}) = TRUE \\ store\_T(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

then

$$\bigwedge \begin{cases} store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ store\_T(a'_{ps}, \kappa, \overline{v}).vl = \perp \end{cases}$$

□

Proof. We proceed by the following reasoning. Given $log\_T(a, \kappa, v) =$ FALSE and $a_{ps} = assertor\_T(a, \kappa, v).dl$, by Lemma 30, $a_{ps} = assertor\_T(a, \kappa, v).ol$. Therefore, by Theorem 22, $store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$ and by Theorem 49, $store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol$.

Given $log\_T(a, \kappa, v) =$ FALSE, by Lemma 33, $coord\_T(a_c, \kappa, v) = \langle \perp, \perp \rangle$. Hence by Lemma 38, $vlstate\_T(a'_{ps}, \kappa, \overline{v}) \neq$ UPDATED.

Given $store\_T(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle$, by Lemma 81, $vlstate\_T(a'_{ps}, \kappa, \overline{v}) \neq$ DEFAULT.

Therefore, by Lemma 40, $store\_T(a'_{ps}, \kappa, \overline{v}).vl = \perp$.

□

Lemma 83 says that an assertor's viewlink points to the store referred by the other assertor's default link, which means each assertor knows the other's default store.

Lemma 83. *For any reachable configuration, for any a, κ and v, and for some a', the following implication holds:*

If $assertor\_T(a, \kappa, v).str \neq (\bot \vee \mathit{INIT})$*, then*

$$assertor\_T(a, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).dl.$$

□

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since table is empty. We now consider only those transitions that may have an effect on terms in the equality.

send_app:
After this transition, $assertor\_T(a, \kappa, v).str$ is set to INIT. Therefore, the precondition is false, which preserves the implication.

prepare_record:
After this transition, $assertor\_T(a, \kappa, v).str$ becomes READY. In addition, $assertor\_T(a, \kappa, v).vl$ equal to the other assertor's default link in the guard to fire this transition. Therefore, the statement is preserved after the transition.

receive_app:
After this transition, $assertor\_T(a, \kappa, v).str$ becomes READY and $assertor\_T(a, \kappa, v).vl$ is set to the link provided by $\mathsf{app}(d, \kappa, vl)$, i.e., the sender's default link (by transition *send_app*). Therefore, the statement is preserved.

timeout_ack:
By Lemma 18, $assertor\_T(a, \kappa, v).str$ was SENT before this transition and is set to ACKED after it. Therefore, the statement remains valid after this transition.

receive_ack:
By Lemma 17, $assertor\_T(a, \kappa, v).str$ was SENT before this transition and may be set to ACKED after it. Therefore, the statement remains valid after this transition.

All the other transitions:
After the other transitions, the precondition remains true and $assertor\_T(a, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).dl$ keeps unchanged. Therefore, the statement holds.

□

Similar to Lemma 38, Lemma 84 connects the state of table *update_T* in the coordinator and the state of a provenance store. It shows how the ASM evolves after the coordinator starts updating a destination provenance store.

Lemma 84. *For any reachable configuration and for any* κ*,* v *and* a_{ps}*, the following equality holds:*

$$update\_T(a_c, \kappa, v, a_{ps}) \neq \bot$$

$$= \bigvee \begin{cases} \mathsf{update}(\kappa, v, ol) \in k(a_c, a_{ps}) \\ \mathsf{update}(\kappa, v, ol) \in lost(a_c, a_{ps}) \\ vlstate\_T(a_{ps}, \kappa, \overline{v}) = \mathit{UPDATED} \\ store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol \\ \mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c) \\ \mathsf{uack}(\kappa, v) \in lost(a_c, a_{ps}) \\ update\_T(a_c, \kappa, v, a_{ps}) = \mathit{UPDATE} \\ update\_T(a_c, \kappa, v, a_{ps}) = \mathit{UPDATED} \end{cases}$$

where $ol = coord\_T(a_c, \kappa, v).ol$.

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty and no message is in transit. We now consider only those transitions that may have an effect on terms in the equality.

receive_repair:
After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE. Therefore, the statement is preserved.

send_update:
After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to WAIT and $\mathsf{update}(\kappa, v, ol)$ is inserted into $k(a_c, a_{ps})$, where $ol = coord\_T(a_c, \kappa, v).ol$. Therefore, the statement is preserved.

timeout_uack:
By Lemma 37, the right part of the statement was true before this transition. After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE. Therefore, the statement is still preserved.

receive_uack:
The statement is preserved by this transition. After this transition, $\mathsf{uack}(\kappa, v)$ is removed from channel. By Lemma 35, $store\_T(a_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol$. Therefore, the right part of the statement was true before the transition and remains true after it, which preserves the statement.

receive_update:
The statement is preserved by this transition. After this transition, $\mathsf{update}(\kappa, v, ol, a_{ps})$ is removed from $k(a_c, a_{ps})$ and an $\mathsf{uack}(\kappa, v)$ is inserted in $k(a_{ps}, a_c)$. In addition, $vlstate\_T(a_{ps}, \kappa, \overline{v})$ is set to UPDATED and $store\_T(a_{ps}, \kappa, \overline{v}).vl$ is set to ol, which is provided by $\mathsf{update}(\kappa, v, ol)$. Since ol is equal to $coord\_T(a_c, \kappa, v).ol$ (Lemma 34), $store\_T(a_{ps}, \kappa, \overline{v}).vl$ is equal to $coord\_T(a_c, \kappa, v).ol$.

msg_loss_pstore:

msg_loss_channel:
The two transitions remove $\mathsf{update}(\kappa, v, ol)$ or $\mathsf{uack}(\kappa, v)$ from channels and then place them in $lost(a_c, a_{ps})$. Therefore, the the statement still holds.

□

Lemma 85 demonstrate the state of the coordinator's timer and update table $update\_T$.

Lemma 85. *For any reachable configuration and for any κ, v and a_{ps}, the following implication holds:*

$$\bigwedge \begin{cases} timer\_T(a_c, \kappa, v, a_{ps}).status = \mathit{ENABLED} \\ timer\_T(a_c, \kappa, v, a_{ps}).to \geq 0 \end{cases}$$

$$\mathit{iff}$$

$$update\_T(a_c, \kappa, v, a_{ps}) = \mathit{WAIT}.$$

□

Proof.

We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
The statement is preserved by these transitions. After this transition, $timer\_T(a_c, \kappa, v, a_{ps})$ remains in the initial state and $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE. Therefore, the statement is preserved.

send_update:
The statement is preserved by this transition. After this transition, $timer\_T(a_c, \kappa, v, a_{ps}).status$ is set to ENABLED and $timer\_T(a_c, \kappa, v, a_{ps}).to$ is initialised, and $update\_T(a_c, \kappa, v, a_{ps})$ is set to WAIT. Therefore, the statement is preserved.

$timer\_click(a_c, \kappa, v, a_{ps})$:
The statement is preserved by this transition. After this transition, $timer\_T(a_c, \kappa, v, a_{ps}).to$ remains greater or equal to 0 and $update\_T(a_c, \kappa, v, a_{ps})$ is not affected.

timeout_uack:
receive_uack:
The statement is preserved by these transitions. After these transitions, $timer\_T(a_c, \kappa, v, a_{ps}).status$ is set to DISABLED and $timer\_T(a_c, \kappa, v, a_{ps}).to$ becomes 0. Meanwhile, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE or UPDATED.

□

Lemma 86 states that if both assertors use alternative stores and request the coordinator to update the other's viewlink, the coordinator updates only one's default store (e.g., receiver b's default store $PS2$ in Figure 3.10). Then the coordinator keeps sending update messages to the destination store $PS2$ until it is updated. However, this update is only successful if the assertor b is still using its default store. If b also uses alternative stores, i.e., the coordinator receives a second repair request in the same interaction (Step 10 in Figure 3.10), the protocol takes actions (Step 11 in Figure 3.10) to ensure the destination store to be updated is correct for both views. Therefore, only one assertor's default store is updated (i.e., $PS2$ in Figure 3.10) due to the coordinator's incomplete knowledge when receiving the first of the two repair requests.

Lemma 86. *For any reachable configuration, for any κ and v, and for some a and a', the following implication holds:*

If $coord\_T(a_c, \kappa, v) \neq \langle \perp, \perp \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \perp, \perp \rangle$, *then*

$$\bigwedge \begin{cases} update\_T(a_c, \kappa, v, a_{ps}) \neq \perp \\ update\_T(a_c, \kappa, \overline{v}, a'_{ps}) = \perp \end{cases}$$

where $a_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$ *and* $a'_{ps} = assertor\_T(a, \kappa, v).dl$. □

Proof. We proceed by the following reasoning. The statement holds in the initial configuration since tables are all empty. We now consider only those transitions that may have an effect on terms in the equality.

receive_repair:
To make the precondition become true, the coordinator receives two repair requests. Due to the symmetric nature of the implication, we assume that the coordinator firstly receives $\mathsf{repair}(\kappa, v, a_{dps}, ol)$ and then $\mathsf{repair}(\kappa, \overline{v}, a'_{dps}, ol')$.

After processing $\mathsf{repair}(\kappa, v, a_{dps}, ol)$, $coord\_T(a_c, \kappa, v)$ becomes not empty while $coord\_T(a_c, \kappa, \overline{v})$ remains $\langle \perp, \perp \rangle$. In addition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE with a_{ps} equal to $coord\_T(a_c, \kappa, v).a_{dps}$. We note that $coord\_T(a_c, \kappa, v).a_{dps}$ is provided by $\mathsf{repair}(\kappa, v, a_{dps}, ol)$. By Lemma 33, a_{dps} in $\mathsf{repair}(\kappa, v, a_{dps}, ol)$ equals to $assertor\_T(a, \kappa, v).vl$. Therefore, $a_{ps} = assertor\_T(a, \kappa, v).vl$. Then with Lemma 83, $a_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$.

After processing another request $\mathsf{repair}(\kappa, \overline{v}, a'_{dps}, ol')$, $coord\_T(a_c, \kappa, \overline{v})$ becomes not empty. According to transition *receive_repair*, $coord\_T(a_c, \kappa, \overline{v}).a_{dps}$ is changed to $coord\_T(a_c, \kappa, v).ol$, different from the one, a'_{dps}, as provided by $\mathsf{repair}(\kappa, \overline{v}, a'_{dps}, ol')$. By Lemma 33, a'_{dps} in $\mathsf{repair}(\kappa, \overline{v}, a'_{dps}, ol)$ equals to $assertor\_T(a', \kappa, \overline{v}).vl$. Hence with Lemma 83, a'_{dps} equals to $assertor\_T(a, \kappa, v).dl$. Therefore, transition *receive_repair*

does not affect $update\_T(a_c, \kappa, \overline{v}, a'_{dps})$, which remains in the initial state $\perp$, such that $a'_{dps} = assertor\_T(a, \kappa, v).dl$.

Thus, after two transitions *receive_repair*, we have $update\_T(a_c, \kappa, v, a_{ps}) \neq \perp$ and $update\_T(a_c, \kappa, \overline{v}, a'_{ps}) = \perp$, such that $a_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$ and $a'_{ps} = assertor\_T(a, \kappa, v).dl$.

send_update:
The statement is preserved by this transition. After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to WAIT from UPDATE. Therefore, the statement was preserved before the transition and remains valid after it.

timeout_uack:
The statement is preserved by this transition. After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE from WAIT (Lemma 85). Therefore, the statement was preserved before the transition and remains valid after it.

receive_uack:
The statement is preserved by this transition. In order to fire this transition, uack is in transit. Therefore, $update\_T(a_c, \kappa, v, a_{ps})$ was not $\perp$ before this transition (Lemma 84). After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is either not changed or set to UPDATED. Hence, the statement was preserved before the transition and remains valid after it.

□

Lemma 87 states a possible topology of interlinked provenance stores in the case where both assertors in an interaction used alternative stores and their respective default store did not record any duplicate interaction record. The property is used by Case (d), Section 5.3.1.1.

Lemma 87. *For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, the following implication holds when the ASM terminates at final configuration:*

If

$$\bigwedge \begin{cases} log\_T(a, \kappa, v) = TRUE \\ store\_T(a_{ps}, \kappa, v) \neq \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ log\_T(a', \kappa, \overline{v}) = TRUE \\ store\_T(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

then

$$\bigwedge \begin{cases} store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ store\_T(a'_{ps}, \kappa, \overline{v}).vl = \perp \end{cases}$$

□

PROOF. We proceed by the following reasoning. Since $assertor\_T(a, \kappa, v).str$ and $assertor\_T(a', \kappa, \overline{v}).str$ are both OK when the ASM terminates (Lemma 20), by Lemma 33, $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle$ and $coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle$. Therefore, by Lemma 86, we have

$$\bigwedge \begin{cases} \text{update\_T}(a_c, \kappa, \overline{v}, a_{ps}) \neq \bot \\ \text{update\_T}(a_c, \kappa, v, a'_{ps}) = \bot \end{cases}$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$.

Given $update\_T(a_c, \kappa, \overline{v}, a_{ps}) \neq \bot$, by Lemma 84, when the ASM terminates,

$$store\_T(a_{ps}, \kappa, v).vl = coord\_T(a_c, \kappa, \overline{v}).ol. \tag{5.3}$$

Since $coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle$, by Lemma 43,

$$coord\_T(a_c, \kappa, \overline{v}).ol = assertor\_T(a', \kappa, \overline{v}).ol. \tag{5.4}$$

With (5.3) and (5.4),

$$store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol. \tag{5.5}$$

Given $update\_T(a_c, \kappa, v, a'_{ps}) = \bot$, by Lemma 84, $vlstate\_T(a'_{ps}, \kappa, \overline{v}) \neq$ UPDATED. Since $store\_T(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle$, by Lemma 81, $vlstate\_T(a'_{ps}, \kappa, \overline{v}) \neq$ DEFAULT. Therefore, $store\_T(a'_{ps}, \kappa, \overline{v}).vl = \bot$ (Lemma 40).

Based on the above proof, the statement is preserved.

□

Lemma 88 states a possible topology of interlinked provenance stores in the case where both assertors in an interaction used alternative stores but one's default store recorded a duplicate interaction record and the other's default store did not. The property is used by Case (e), Section 5.3.1.1.

Lemma 88. *For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, the following implication holds when the ASM terminates at final configuration:*

If

$$\bigwedge \begin{cases} log\_T(a, \kappa, v) = TRUE \\ store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ log\_T(a', \kappa, \overline{v}) = TRUE \\ store\_T(a'_{ps}, \kappa, \overline{v}) \neq \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

then

$$\bigvee \left\{ \begin{array}{l} \bigwedge \left\{ \begin{array}{l} store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ store\_T(a'_{ps}, \kappa, \overline{v}).vl = \bot \end{array} \right. \\ \bigwedge \left\{ \begin{array}{l} store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).dl \\ store\_T(a'_{ps}, \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).ol \end{array} \right. \end{array} \right.$$

□

Proof. We proceed by the following reasoning. Since $assertor\_T(a, \kappa, v).str$ and $assertor\_T(a', \kappa, \overline{v}).str$ are both OK when the ASM terminates (Lemma 20), by Lemma 33, $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle$ and $coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle$. Therefore, by Lemma 86, we discuss two cases:

Case (1):

$$\bigwedge \left\{ \begin{array}{l} \text{update\_T}(\text{a}_c, \kappa, \overline{v}, a_{ps}) \neq \bot \\ \text{update\_T}(\text{a}_c, \kappa, v, a'_{ps}) = \bot \end{array} \right.$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$. In this case, we can use the same proof as in Lemma 87 to demonstrate that:

$$\bigwedge \left\{ \begin{array}{l} \text{store\_T}(\text{a}_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ \text{store\_T}(\text{a}'_{ps}, \kappa, \overline{v}).vl = \bot \end{array} \right.$$

Therefore, the statement is preserved.

Case(2):

$$\bigwedge \left\{ \begin{array}{l} \text{update\_T}(\text{a}_c, \kappa, \overline{v}, a_{ps}) = \bot \\ \text{update\_T}(\text{a}_c, \kappa, v, a'_{ps}) \neq \bot \end{array} \right.$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$.

Given $update\_T(a_c, \kappa, v, a'_{ps}) \neq \bot$, by Lemma 84,

$$store\_T(a'_{ps}, \kappa, \overline{v}).vl = coord\_T(a_c, \kappa, v).ol. \tag{5.6}$$

Since $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle$, by Lemma 43,

$$coord\_T(a_c, \kappa, v).ol = assertor\_T(a, \kappa, v).ol. \tag{5.7}$$

With (5.6) and (5.7),

$$store\_T(a'_{ps}, \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).ol. \tag{5.8}$$

Given $update\_T(a_c, \kappa, \overline{v}, a_{ps}) = \bot$, by Lemma 84, $vlstate\_T(a_{ps}, \kappa, v) \neq \mathsf{UPDATED}$. Since $store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle$, such that $vl \neq \bot$, by Lemma 40, $vlstate\_T(a_{ps}, \kappa, v) = \mathsf{DEFAULT}$. Therefore, by Lemma 39 and Lemma 83, $store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).dl$. Now we have

$$\bigwedge \begin{cases} \text{store\_T}(\text{a}_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).dl \\ \text{store\_T}(\text{a}'_{ps}, \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).ol \end{cases}$$

Therefore, the statement is preserved.

□

Lemma 89 states a possible topology of interlinked provenance stores in the case where both assertors in an interaction used alternative stores and their respective default store recorded a duplicate interaction record. The property is used by Case (f), Section 5.3.1.1.

Lemma 89. *For any a, κ and v, and for some a', a_{ps} and a'_{ps}, such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$, the following implication holds when the ASM terminates at final configuration:*

If

$$\bigwedge \begin{cases} log\_T(a, \kappa, v) = \mathit{TRUE} \\ store\_T(a_{ps}, \kappa, v) = \langle a, vl, assertor\_T(a, \kappa, v).pas \rangle \\ log\_T(a', \kappa, \overline{v}) = \mathit{TRUE} \\ store\_T(a'_{ps}, \kappa, \overline{v}) = \langle a', vl', assertor\_T(a', \kappa, \overline{v}).pas \rangle \end{cases}$$

then

$$\bigwedge \begin{cases} store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ store\_T(a'_{ps}, \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).dl \end{cases}$$

□

Proof. We proceed by the following reasoning. Since $assertor\_T(a, \kappa, v).str$ and $assertor\_T(a', \kappa, \overline{v}).str$ are both OK when the ASM terminates (Lemma 20), by Lemma 33, $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle$ and $coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle$. Therefore, by Lemma 86, we have:

$$\bigwedge \begin{cases} \text{update\_T}(\text{a}_c, \kappa, \overline{v}, a_{ps}) \neq \bot \\ \text{update\_T}(\text{a}_c, \kappa, v, a'_{ps}) = \bot \end{cases}$$

such that $a_{ps} = assertor\_T(a, \kappa, v).dl$ and $a'_{ps} = assertor\_T(a', \kappa, \overline{v}).dl$. Following the similar proof for Case (2) of Lemma 88, we can derive that

$$\bigwedge \begin{cases} \text{store\_T}(\text{a}_{ps}, \kappa, v).vl = assertor\_T(a', \kappa, \overline{v}).ol \\ \text{store\_T}(\text{a}'_{ps}, \kappa, \overline{v}).vl = assertor\_T(a, \kappa, v).dl \end{cases}$$

Therefore, the statement is preserved.

□

The above analysis discusses the topology of two assertors' default stores and final stores in terms of how they are interlinked via viewlinks when the ASM terminates. Since any alternative store used by an assertor may also record duplicate interaction records with viewlinks pointing to another store, we now investigate the topology regarding alternative stores.

Lemma 90 states that only the provenance store (a_{ps}) referred by an asserter's default link or ownlink in the interaction context $\langle \kappa, \overline{v} \rangle$ may lead to $update\_T(a_c, \kappa, v, a_{ps}) \neq \bot$.

Lemma 90. *For any κ and v, and for some a, the following statement holds when the ASM terminates at final configuration:*

If $update\_T(a_c, \kappa, v, a_{ps}) \neq \bot$, then

$$a_{ps} = assertor\_T(a, \kappa, \overline{v}).dl \text{ or } assertor\_T(a, \kappa, \overline{v}).ol.$$

□

Proof. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The statement holds in the initial configuration since tables are empty. We now consider only those transitions that may have an effect on terms in the implication.

receive_repair:
After this transition, $coord\_T(a_c, \kappa, v)$ is no longer $\langle \bot, \bot \rangle$ and $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE. We discuss two cases in order to determine a_{ps}.

Case (1) $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) = \langle \bot, \bot \rangle$:
In this case, a_{ps} is set to $coord\_T(a_c, \kappa, v).a_{dps}$, which is provided by the a_{dps} from message $\mathsf{repair}(\kappa, v, a_{dps}, ol)$. By Lemma 33, $a_{dps} = assertor\_T(a, \kappa, \overline{v}).vl$. Therefore,

$$a_{ps} = assertor\_T(a, \kappa, \overline{v}).vl.$$

Case (2) $coord\_T(a_c, \kappa, v) \neq \langle \bot, \bot \rangle \wedge coord\_T(a_c, \kappa, \overline{v}) \neq \langle \bot, \bot \rangle$:
In this case, a_{ps} is set to $coord\_T(a_c, \kappa, v).a_{dps}$, which is provided by $coord\_T(a_c, \kappa, \overline{v}).ol$ (Lemma 44). By Lemma 43,

$$coord\_T(a_c, \kappa, \overline{v}).ol = assertor\_T(a, \kappa, \overline{v}).ol.$$

Therefore, we have

$$a_{ps} = assertor\_T(a, \kappa, \overline{v}).ol.$$

Based on the above analysis, the statement is preserved.

send_update:
After this transition, the antecedent remains true. Therefore, the statement is still preserved.

timeout_uack:
The statement is preserved by this transition. After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is set to UPDATE from WAIT (Lemma 85). Therefore, the statement was preserved before the transition and remains valid after it.

receive_uack:
The statement is preserved by this transition. Before this transition, $update\_T(a_c, \kappa, v, a_{ps})$ was not $\perp$ (Lemma 84). After this transition, $update\_T(a_c, \kappa, v, a_{ps})$ is either not changed or set to UPDATED. Therefore, the statement was preserved before the transition and remains valid after it.

□

Lemma 91 states that if a provenance store referred by the element in $PSSet(a, \kappa, v)$ (defined in Section 5.1.3) has recorded a viewlink, then the store possesses the whole interaction record with a viewlink pointing to the other assertor's default store. This property is used in Section 5.3.1.2.

Lemma 91. *For any reachable configuration, for any $a_{ps} \in PSSet(a, \kappa, v)$, any κ and v, and for some a, a', the following implication holds:*

If $store\_T(a_{ps}, \kappa, v).vl \neq \perp$, then

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

where $vl = assertor\_T(a', \kappa, \overline{v}).dl$. □

Proof. We proceed with the following reasoning. Given $a_{ps} \in PSSet(a, \kappa, v)$, we have $a_{ps} \neq assertor\_T(a, \kappa, v).dl$ and $a_{ps} \neq assertor\_T(a, \kappa, v).ol$. Then by Lemma 90, $update\_T(a_c, \kappa, \overline{v}, a_{ps}) = \perp$. Therefore, we can infer that $vlstate\_T(a_{ps}, \kappa, v) \neq$ UPDATED (Lemma 84). Given $store\_T(a_{ps}, \kappa, v).vl \neq \perp$ and $vlstate\_T(a_{ps}, \kappa, v) \neq$ UPDATED, by Lemma 40, $vlstate\_T(a_{ps}, \kappa, v) =$ DEFAULT. Therefore, by Lemma 81,

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

By Lemma 39,

$$store\_T(a_{ps}, \kappa, v).vl = assertor\_T(a, \kappa, v).vl \tag{5.9}$$

When the ASM terminates, $assertor\_T(a, \kappa, v).str$ is OK (Lemma 20). Therefore, with

(5.9) and Lemma 83,

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

Based on the above proof, the statement is preserved.

□

We have discussed provenance stores interlinked via viewlinks (from Lemma 80 to Lemma 91). Now we discuss those linked via causelinks.

CAUSELINK ACCURACY property (Property 23) has stated that any causelinks recorded in the provenance store that is referred by an assertor's ownlink are accurate, i.e., pointing to the provenance store referred by the other assertor's ownlink. We now extend this conclusion to a general case.

Lemma 92 shows that an assertor's causelinks duplicately stored in any provenance store are always accurate, pointing to the store referred by the other assertor's ownlink. This implies a topology of provenance stores interlinked by causelinks. This property is used in Section 5.3.2.

Lemma 92. *For any a_{ps}, κ and vl, and for some a, the following implication holds when the ASM terminates at final configuration:*

If $\mathsf{rel\text{-}pa}(rel, \langle \kappa, S \rangle, cids) \in store\_T(a_{ps}, \kappa, S).pas$, *then*

$$\begin{aligned} &for\ any\ c \in cids,\ let\ \langle cl', \kappa', R \rangle = c, \\ &\quad cl' = assertor\_T(a, \kappa', R).ol \wedge \\ &\quad store\_T(cl', \kappa', R) = \langle a, vl, assertor\_T(a, \kappa', R).pas \rangle \end{aligned}$$

□

PROOF. We proceed by the following reasoning. Lemma 20 shows that when the ASM terminates, $assertor\_T(a, \kappa, S).str$ is OK. Therefore, all causelinks in $assertor\_T(a, \kappa, S).pas$ are accurate before being recorded in a provenance store (Lemma 26).

Given $\mathsf{rel\text{-}pa}(rel, \langle \kappa, S \rangle, cids) \in store\_T(a_{ps}, \kappa, S).pas$, i.e., $store\_T(a_{ps}, \kappa, S).pas \neq \emptyset$, by Lemma 36, $store\_T(a_{ps}, \kappa, S).pas = assertor\_T(a, \kappa, S).pas$. This shows that all causelinks recorded in $store\_T(a_{ps}, \kappa, S).pas$ are also accurate. Therefore, this implication is preserved.

□

Chapter 6

Implementation and Evaluation

F-PReP is a generic protocol that does not specify implementation details. In order to put the protocol into practice, this chapter describes a system architecture that implements F-PReP while considering practical issues such as communication, storage and performance impact on a provenance-aware application's execution.

Our performance evaluation is conducted at several levels. Firstly, we measure the throughput of the provenance store and the coordinator. We investigate how the contention for the coordinator affects an actor's recording performance when the number of recording actors increases. We demonstrate that a single coordinator does not result in a performance bottleneck. Secondly, we benchmark the recording performance of F-PReP. The results show that its remedial actions introduce small overhead (below 10%). Thirdly, we investigate the performance impact on the execution time of a scientific application. We find that PReP and F-PReP have similar impact on application execution when there is no failure. In tests with failures, the recording overhead of F-PReP varies depending on configurations.

The contributions of this chapter are twofold:

We describe F-PReServ, an implementation of F-PReP with architectural support for practical issues such as communication, storage and performance. The implementation of F-PReServ supports requirement Efficient Recording identified in Chapter 1. Its features include a novel way of creating process documentation, basic flow control management for recording documentation, and a local store for temporarily maintaining documentation to avoid severe performance degradation in the presence of failures. We also discuss various approaches that are complementary to our implementation.

Another contribution is the extensive evaluation of F-PReServ's performance. The experimental results show that F-PReServ introduces reasonable overhead when compared to PReServ and has some performance impact on an application's execution. We believe these results are still acceptable given that the process documentation is guaranteed to

be recorded in the presence of failures and still retrievable in its entirety from multiple provenance stores. Lessons are learned and recommendations are given on achieving good performance in the case of failures.

The rest of this chapter is organised as follows. Section 6.1 presents the design and implementation of F-PReServ in terms of its three components: Client Side Library, Provenance Store Service and Coordinator Service. Section 6.2 describes our evaluation environment and methodology, followed by a series of performance experiments in controlled environments and in a real scientific application. Based on lessons learned from the experimental results, Section 6.3 provides several recommendations on the development of next version F-PReServ and discusses relevant technologies that can be integrated with F-PReServ. Section 6.4 briefly reviews related work. Finally, Section 6.5 summarises and concludes this chapter.

6.1 Implementation

In this section, we first outline the technologies used by F-PReServ and then introduce F-PReServ's components: an assertor side library (F-PSL), a Provenance Store Service and a Coordinator Service (Figure 6.1).

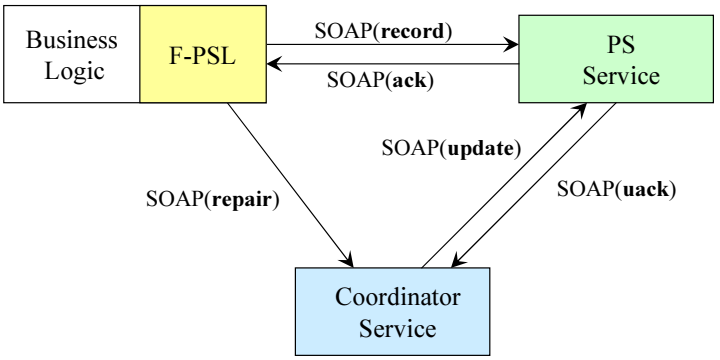


FIGURE 6.1: F-PReServ components

One of the main ways to enable SOAs is to build interoperable applications out of Web Services, where Simple Object Access Protocol (SOAP) specifies a platform-independent standard of exchanging messages between heterogeneous services in SOAs. SOAP messages are typically transmitted over Hypertext Transport Protocol (HTTP) to facilitate easy communication through network proxies and firewalls. PReServ [76, 79], an implementation of PReP, adopts these technologies. In PReServ, provenance stores are described as Web Services and implemented as a Java Servlet deployed in the Apache Tomcat web container [1] with protocol messages exchanged in the form of SOAP over HTTP. To be compatible with PReServ, F-PReServ also adopts these technologies. In addition to extending PReServ's provenance store service, F-PReServ introduces a coordinator service, implemented as a Java Servlet. The extension of the provenance store

service and implementation of coordinator service also feature *Ease of Installation* [76], so that developers can easily deploy multiple provenance stores and coordinators to test their applications and to enable scalable recording of process documentation in SOAs.

F-PSL extends a Provenance Support Library (PSL), which is developed using Java by the University of Southampton. PSL provides users with a set of Application Programming Interfaces (API) for creating and recording interaction records as well as for querying a Provenance Store. The selection of Java allows for the software to run on any platform that has a Java Virtual Machine without recompilation. The reason to extend PSL rather than the counterpart client side library in PReServ (termed *PReServ-Client*) is that PSL encodes SOAP messages using Apache Axis, which is a popular Web service engine used by many applications such as Globus Toolkit 4 (GT4)[1]. Therefore, F-PSL can be integrated in a wide range of Grid applications to reliably record process documentation.

We now detail F-PReServ's components in Sections 6.1.1, 6.1.2 and 6.1.3.

6.1.1 F-PSL

F-PSL encompasses five modules (Figure 6.2): Client API, Thread and Queue Management (TQM), Recording Management (RM), Local Persistence Management (LPM) and a Configuration file.

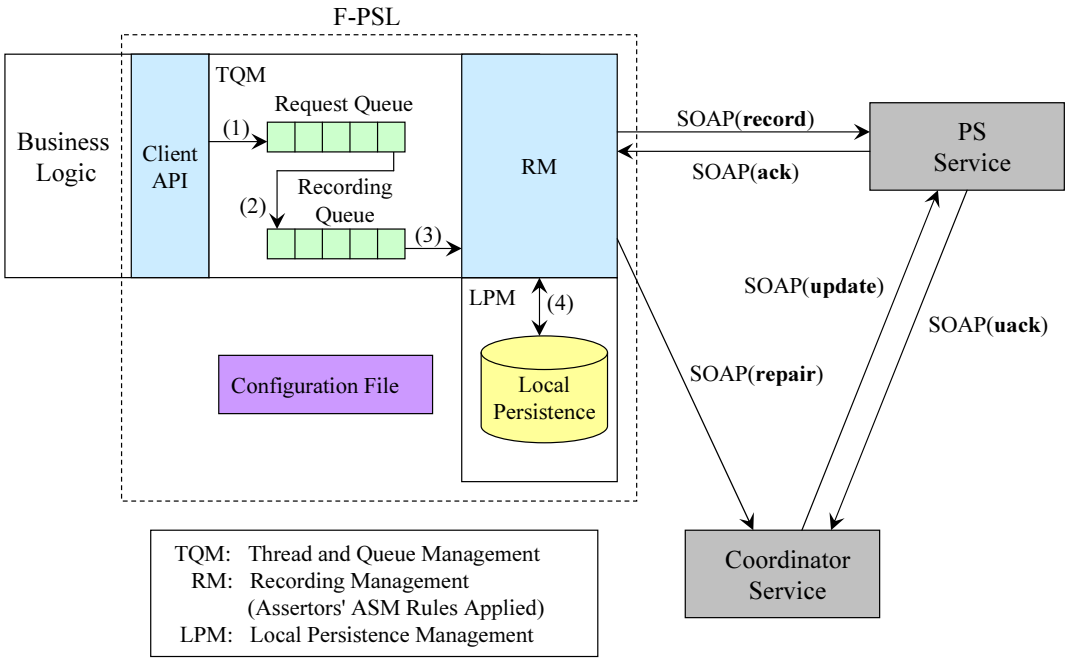


FIGURE 6.2: F-PSL overview

Client API

Making an application provenance-aware inevitably affects the application's execution

[1] GT4, an open source software toolkit employing Web Services to construct Grid systems.

as it takes time to create and record interaction records (IRs). One novelty of the Client API in F-PSL is the change in the way that IRs are generated, so that the impact on application's performance can be minimised. This improvement supports requirement Efficient Recording.

We illustrate this change in Figure 6.3, where we compare three approaches: PSL, PReServ-Client and F-PSL. The difference between these approaches is when to create and record IRs. In PSL[2], the creation and recording are all synchronous to an application's execution, resulting in performance penalty. In PReServ-Client, although application execution and IR creation are still synchronous, the recording of IR is made asynchronous, leading to performance enhancement. In the case of F-PSL, the Client API enables developers to only generate requests containing minimum information in order to create an IR. IRs are then asynchronously created and recorded by separate threads managed by TQM, further increasing recording performance.

We note that Figure 6.3 does not intend to give specific timing measurements regarding each approach. But we did observe a significant increase in performance (20-30%) when comparing F-PSL to PSL in our tests[3].

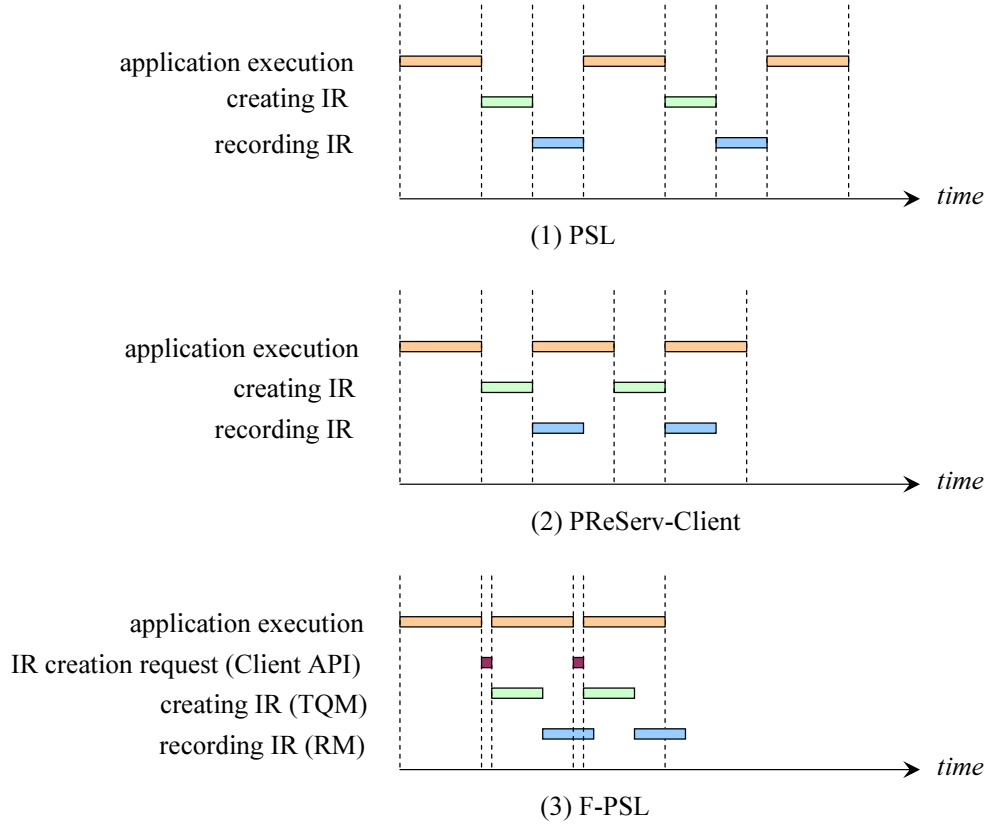


Figure 6.3: Comparing approaches to the creating and recording of IR

We now outline in Figure 6.4 how an application's source code is modified in order

[2]PSL v1.3 was used in the comparison.
[3]In the tests, we modified PSL so it can also record IRs asynchronously, as PReServ-Client does.

to use the Client API. In the figure, we assume the example has a request-response communication model, which means that there are two interactions and an assertor is the sender or receiver in a respective interaction. Each asserter uses a same provenance store to record two IRs about the two interactions and the store's address is sent to the other assertor in the request or response message along with the interaction key created by the sender of the message (Steps 1, 4). After knowing the viewlink, an assertor generates requests to create IRs, which are placed in the Request Queue (Figure 6.2). We note that the assertor's ASM rules in Figures 3.12 and 3.13 are high-level specification, which do not model the approach adopted by F-PSL in terms of generating requests to create IRs.

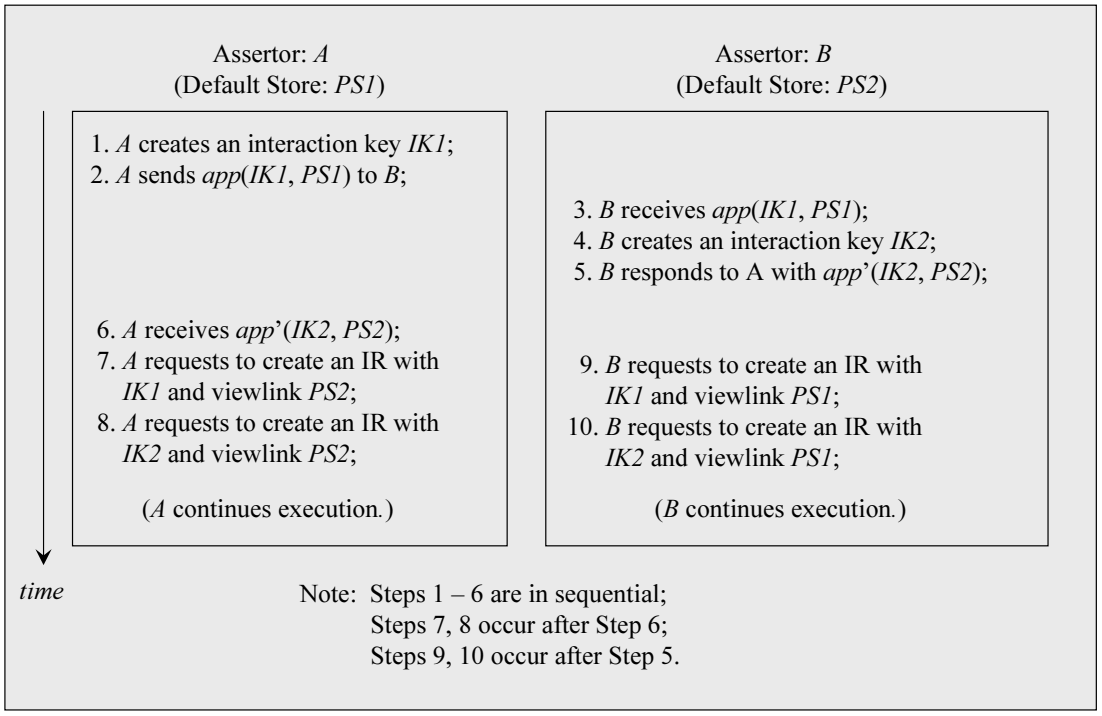


FIGURE 6.4: A Request-response example

In addition to the creation of IRs, Client API also implements the extended retrieval function introduced in Section 5.4.3.4.

Thread and Queue Management (TQM)

F-PSL enables the concurrent creation and recording of IRs through multithreading, as illustrated in Figure 6.3. An application's requests for creating IRs (Step (1) in Figure 6.2) are queued to be processed by a thread, which creates IRs later. To save on the cost of network connection, multiple IRs can be wrapped in a single batch, which is also queued (Step (2) in Figure 6.2) before being handled to RM by a recording thread (Step (3) in Figure 6.2). The use of multiple-threading and batches of messages supports requirement EFFICIENT RECORDING.

Another enhancement over PSL and PReServ-Client is that F-PSL provides basic flow control mechanism to prevent exhausting a client's memory due to large amount of

messages enqueued. This may however affect an application's performance, since the application is postponed occasionally to reduce the speed of producing requests to create IRs when queues become full frequently. We will demonstrate in Section 6.2.5 the impact of flow control mechanism on an application's execution.

Recording Management (RM)
RM records IRs into a provenance store and takes remedial actions in the presence of failures. Its behaviour is specified by the ASM rules in Figure 3.14. We now briefly introduce the implementation of these rules.

pre_check:
This rule specifies how to check and update causelinks before recording an IR. The rule involves three tables. Table $queue\_T(a)$ models the Recording Queue in TQM. Table $assertor\_T(a, \kappa, v)$ reflects an assertor's knowledge during the recording of an IR, which is solely for the purpose of proof and hence does not need to be implemented. Table $log\_T(a, \kappa, v)$ maintains history information regarding the use of alternative stores, which is crucial to fixing inaccurate causelinks. Therefore, RM needs to maintain the log information when an alternative store is used and check the information before recording any relationship p-assertion.

send_record:
timer_click:
timeout_ack:
These rules specify the communication with a provenance store. Should an acknowledgement fail to be received before a timeout, the same IR is resent to an alternative store. Although an alternative store is always used in the formalisation, the retransmission policy can be configured in the configuration file provided by F-PSL. For example, an IR can be resent to the same provenance store for certain times before using an alternative store.

We note that the formalisation of a timeout is designed to be an abstract way of detecting a potential failure in order to trigger the remedial actions. In practice, the triggering condition is not necessarily limited to the expiry of a timeout. For example, it can be a failure to connect to a provenance store and can also be the receipt of a response message with a fault code indicating any exception thrown in the provenance store service, e.g., storage exception. The occurrence of these events does not guarantee successful recording of IRs in a provenance store, therefore remedial actions need to be taken. In the current implementation of F-PSL, remedial actions are taken in response to an exception thrown on the client side regarding the submission of a batch of IRs to a provenance store, which may be due to a connection failure or the expiry of a timeout. In future work, we will revise the definition of an acknowledgement message to include fault information reflecting the exceptions that occurred on the provenance store's side.

receive_ack:

Once an acknowledgement is received from a provenance store, RM removes the acknowledged IR from the recording queue.

post_check:
RM communicates with the coordinator service if an alternative store was used in an interaction context. To reduce network connection overhead, multiple repair requests are accumulated before being sent to the coordinator in a single batch.

Local Persistence Management (LPM)
Another novelty of F-PSL over PSL and ReServ-Client is that it introduces a local store for temporarily maintaining IRs. Recall that F-PSL provides flow control mechanism, setting a limit on the capacity of the request and recording queues to prevent running out of a client's memory. If the threshold is reached, then the application's execution has to be suspended until there is space in queues to accommodate additional requests or IR messages. In order not to degrade the application's performance significantly, outstanding IR messages in the recording queue can be maintained in the local store and resubmitted later (Step (4) in Figure 6.2).

We employ Oracles Berkeley DB Java Edition database (BDB) as the local file store for the following reasons. Firstly, BDB is an embedded database without the complexity of installation as a separate service or application. The only requirement is that BDB must be provided with a directory where it can write its files to. Secondly, BDB is an append-only database and thus is optimised for write performance. When employing the local file store, another thread is provided to resubmit IRs from BDB to a provenance store.

Configuration File
The configuration file allows developers to customise F-PSL's behaviour. The following properties are specific to the new functionalities introduced by F-PSL.

ALT_STORE_LIST	A list of alternative provenance store addresses
TIMEOUT	The deadline for receiving an acknowledgement from a store
RETRY_COUNTS	The max number of message retransmissions to stores
USE_ALT_STORE	Boolean value indicating redelivering a message to an alt. store or a same store
COORD_URL	The address of the coordinator service
REQ_QUEUE_SIZE	The capacity of the request queue
REC_QUEUE_SIZE	The capacity of the recording queue
REC_BATCH_SIZE	The number of interaction records that are batched together for delivery to a provenance store
REP_BATCH_SIZE	The number of repair requests that are batched together for delivery to the coordinator
LOCAL_STORE_DIR	The path to the directory of the local store

TABLE 6.1: Configuration properties

6.1.2 Provenance Store Service

Figure 6.5 gives the architecture of provenance store service (PS). F-PReServ extends PReServ's implementation of PS (termed *PReServ-PS*) in terms of the following aspects:

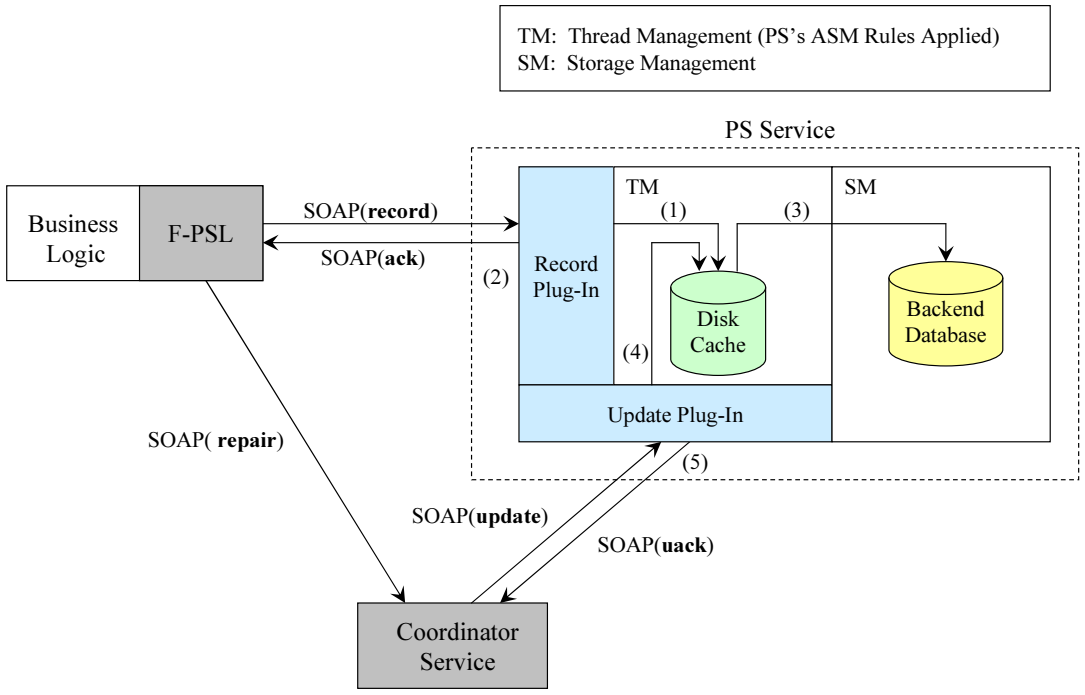


FIGURE 6.5: Overview of provenance store service

(1) Disk Cache. PS *persistently* writes received IRs to BDB (Step (1) in Figure 6.5) before providing an acknowledgement (Step (2) in Figure 6.5). Therefore, the data is guaranteed to be available when the PS comes back up after a crash. By contrast, PReServ-PS, though caching IR messages into BDB before replying an acknowledgement, does not write through the messages to disk on flush, thus having a risk of losing IRs in the event of system crash. We will demonstrate this in Section 6.2.4.

In order to reduce the cost of writing through data to disk, F-PReServ does not record IRs directly in its backend storage but processes the cached items at a later stage after providing an acknowledgement (Step (3) in Figure 6.5). This asynchronous strategy supports requirement EFFICIENT RECORDING because it delays actual message processing and hence reduces response time to the corresponding assertor.

(2) Update Plug-In. PS has been designed to facilitate convenient integration of new features through the use of plug-ins [76]. A new plug-in, Update Plug-In, is implemented as a Java Servlet to receive `update` requests from the coordinator and update the requested view links. In order to balance the tradeoff between reliability and performance, PS also caches incoming `update` messages into disk (Step (4) in Figure 6.5) before returning an acknowledgement to the coordinator (Step (5) in Figure 6.5) and at a later stage processes those messages.

Regarding the implementation of the two ASM rules in Figure 3.15, we note that table $vlstate\_T(a_{ps}, \kappa, v)$ reflects the store's state and is solely for the proof in Chapter 4, hence it does not need to be implemented. The backend database was designed and implemented in PReServ-PS. It uses an interaction context (an interaction key and a viewkind) to index an interaction record, containing the assertor's identity, a viewlink and a set of p-assertions documenting the interaction which are created by the assertor.

6.1.3 Coordinator Service

Figure 6.6 shows the architecture of the Coordinator Service.

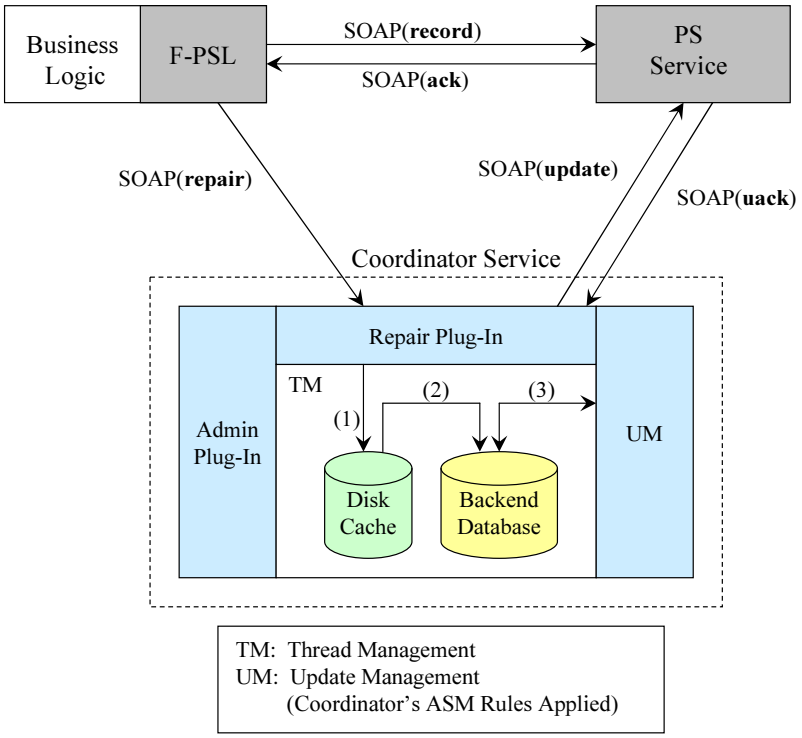


FIGURE 6.6: Overview of coordinator service

The coordinator persistently stores received requests in a local file store (Step (1) in Figure 6.6). At a later stage, it then processes the cached requests by storing them in a backend store (Step (2) in Figure 6.6) and performing update operations (Step (3) in Figure 6.6). An assertor continues its execution after receiving a response indicating its repair request has been cached in the coordinator. BDB is also employed as the file store.

Since the coordinator service is required to be highly available, multiple coordinator services can be clustered in a business critical environment, which we will discuss in Section 6.3.3.

We also provide basic administrative functionality to the coordinator service. An admin plug-in has been implemented to monitor the update status of the coordinator by

querying the backend data store.

In SOAs, the process of executing a large-scale application may span different organisations. Multiple coordinators can be utilised in the whole process. When using more than one, any two assertors exchanging an application message must share the same one in order to ensure requirement VIEWLINK ACCURACY. The identifier of a coordinator can be built in assertors or exchanged to other assertors in the application message.

The behaviour of UM has been detailed in Figure 3.6.5 and we outline its implementation.

receive_repair:
Contents in tables *coord_T* and *updateT* are maintained in the backend database. Only minimum information is maintained in the backend database for each repair request: the identity of the destination store that needs to be updated, the identity of the store that successfully recorded the requesting assertor's interaction record for a given interaction and the update state (UPDATE and UPDATED). We use the associated interaction context (an interaction key and a viewkind) to index the above information in database. The Update Management (UM) component then performs update operations based on the information in the backend database.

send_update:
timer_click:
timeout_uack:
These rules specify the sending of an update message to a provenance store. In the current implementation, a message is resent to the same provenance store upon a communication exception thrown on the coordinator's side, which may be due to a connection failure to a provenance store or the expiry of a timeout for receiving an uack message.

receive_uack:
Once an acknowledgement is received, UM changes the updating status of the corresponding interaction context to UPDATED in the backend database (Step (3) in Figure 6.6). However, it may not be able to delete the information regarding the interaction context from the database for the following reason.

The request information maintained in the database is essential to ensuring successful update of viewlinks given the case where two assertors may each issue a repair request about the same interaction. If the coordinator does not record the first assertor's request or it deletes the first assertor's request information before receiving the other assertor's request, then it would not send update messages to the correct destination stores. Therefore, UM does not remove request information from backend database until receiving two requests each from an assertor of an interaction. If only one assertor of an interaction sends a repair message, this information is kept in the database for a long period of

time[4] before it can be removed.

6.2 Performance Evaluation

After presenting the design and implementation of F-PReServ, this section evaluates the performance of F-PReServ and its impact on an application's execution time.

Our experiments were run on the Iridis Computing Cluster at the University of Southampton. Iridis contains several sets of nodes (i.e. computers). Nodes used in the experiments each have two Single Core AMD Opteron processors running at 2.2 GHz and 2 GB of RAM. The Provenance Store Service and Coordinator Service were run on nodes each with 4 Dual Core AMD Opteron processors running at 2.4 Ghz and 2 GB of RAM. In the experiments with failures, one coordinator service was employed. All nodes are connected by Gigabit Ethernet. All applications used in the evaluation were written in Java and were run using the Java 1.5.0 05 64-bit Server Virtual Machine.

Our evaluation was conducted at three levels. First of all, we measured the throughput of the provenance store service and coordinator service. We also investigated how the contention for the coordinator service affects an actor's recording performance when the number of recording actors increases. Then, we benchmarked the recording performance of F-PSL without considering contention. Finally, we investigated F-PReServ's impact on the execution time of a scientific application. In each level, we performed two experiments: failure-free experiment and experiment with failures. A comparison with PReServ was made in all the failure-free experiments.

There are numerous factors that can affect the system performance [144, 83], such as disk speed, processor speed, system memory, networks, Java Virtual Machine's heap size, Tomcat web container's thread pool size as well as the capacity of queues in F-PSL. Given the available hardware and software resources for the experiments, we modified those configurations that may potentially affect our results, tuned the system until we achieved the near-best results, which are presented here.

6.2.1 Injecting Failures

Before advancing to the experiments, this section introduces the methodology we adopted to inject failures.

Given the failure assumptions in Section 3.2, provenance store crashes and communication channel omission failures need to be considered in order to measure a provenance-aware application's performance in the presence of failures. However, these failures are

[4] longer than an application's lifetime

non-deterministic in nature and typically very hard to predict, therefore it is infeasible to perform experiments with real failures. Hardware and software based fault-injection tools for Web-Service applications have been developed [86, 108, 109]. In addition to the administrative complexity, these tools inevitably introduce operating overhead such as decoding SOAP messages in order to inject faults, which may not be negligible in a high performance cluster environment and hence may affect our results.

Instead, we decided to develop a generator on a client side to inject random failure events when submitting or resubmitting a message (i.e., a batch of interaction records) to a provenance store. A failure event results in an exception caught by F-PSL and consequently the remedial actions taken by F-PSL. The generator generates a failure event based on a failure rate, i.e., the number of failure events that occur out of a total number of messages sent to a provenance store. The generator can emit a failure event immediately to simulate an instant connection failure or can postpone generating a failure event for a pre-specified timing interval to mimics the latency of detecting a potential failure. For example, the failure to receive an acknowledgement is not known until a timeout is expired.

The advantages of adopting such a generator are as follows:

Firstly, the client-side generator is easy to implement. Provenance store crashes and omission failures only matter when an assertor is sending messages to a store. These failures, if not masked by HTTP transporting SOAP messages, lead to exceptions thrown on the client side, and to the remedial actions taken by F-PSL, which may further affect an application's execution. Our aim is to measure the impact of F-PReServ on the application's performance, therefore it is reasonable to generate exceptions on the client side without concerning real failures or using fault-injection tools to simulate service crashes or message omissions.

Secondly, the generator enables us to fully control the number of failure events that occur in the system. This helps us understand the correlation between system performance and failure rate.

Thirdly, since the length of a delay to produce a failure event is configurable, we can investigate different cases by increasing the delay interval's value from zero. The experiment in the best case, i.e., failure events are produced immediately without latency, implies the most messages sent to a provenance store and the coordinator in a given time frame. This leads to the heaviest load on the coordinator service and provenance store service for a given failure rate, which is useful in the throughput experiments in Section 6.2.3. In addition, by increasing the interval, we can also observe a general pattern in terms of how the latency of detecting a failure event may affect application performance and accordingly, we can provide recommendations to developers regarding how to improve the performance of their provenance-aware application, as illustrated in Section 6.2.5.

We note that there is no generator on the coordinator's side in our experiments. According to Figure 6.6, the application continues execution after the coordinator caches repair requests. At a later stage, the coordinator processes the requests to update provenance stores. Hence, the failures that may occur during the communication between the coordinator and provenance stores do not affect the application's performance, which is the reason why we did not generate failure events on the coordinator's side.

6.2.2 Throughput Experiments

In the throughput experiments, we simulate a large number of concurrent clients communicating with the provenance store service and coordinator service to determine the saturation point of the service, i.e., at what point the service capabilities are fully stressed.

Our approach to simulating concurrent clients is as follows. On each node of the cluster, we created up to 16 threads (i.e., clients) sending messages to a service at the same time. An MPI based test harness was used in the experiments to guarantee that all clients were run in parallel. Given that an experiment is allowed to use up to 32 nodes in the Iridis environment, we can have 512 active clients sending messages to a provenance store service or coordinator service. More specifically, active clients send a request, wait for the response, and immediately create a new request upon response reception and resend the new request.

Throughput of Provenance Store Service In Section 6.1.2, a *disk cache* mechanism was introduced as the default setup of a provenance store in F-PReServ. This means the store forces every received interaction record (record) message into disk before providing an acknowledgement in order to maintain the durability of these messages. However, this mechanism may sacrifice a provenance store's throughput (i.e. the number of record messages accepted in a period of time).

We performed two failure-free tests with and without *disk cache* enabled, respectively. All record messages, each in a single batch, were directly created and submitted to a provenance store without using threading.

Figure 6.7 shows the results. In both setups, the provenance store's throughput levels off, where about 212,200 and 176,000 10k record messages are accepted in a 10 minute period in the setup *without disk cache* and *with disk cache*, respectively. This means a store's throughput decreases by 20% due to enabling disk cache.

Throughput of Coordinator Service We also measured the coordinator's throughput (i.e. the number of repair requests accepted in a period of time) with up to 512 clients simultaneously sending repair messages to an update coordinator. To save on the cost of network connection, 100 repair requests were sent to a coordinator in a single batch.

Figure 6.8 shows that a coordinator can accept up to around 30,000*100 repair requests in a 10 minute period. This means there were at least 30,000*100 failure events in 10 minutes, which is unlikely to appear in applications.

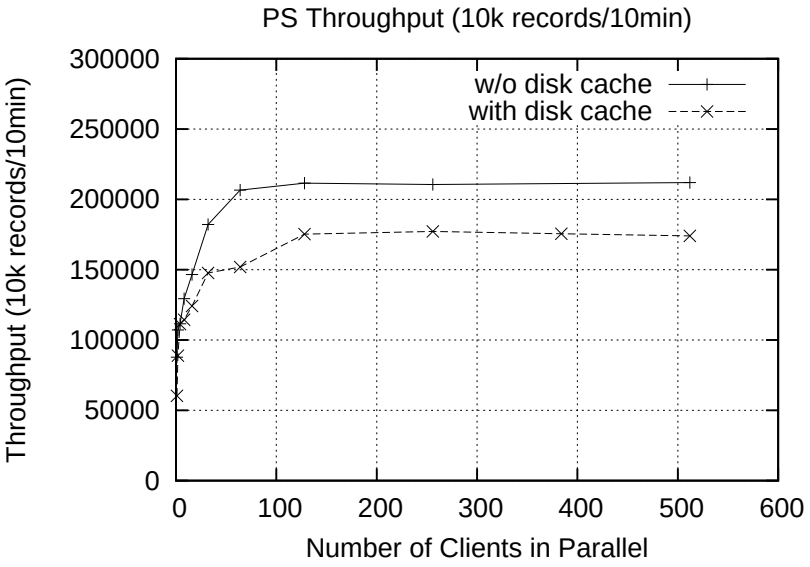


FIGURE 6.7: Provenance store throughput

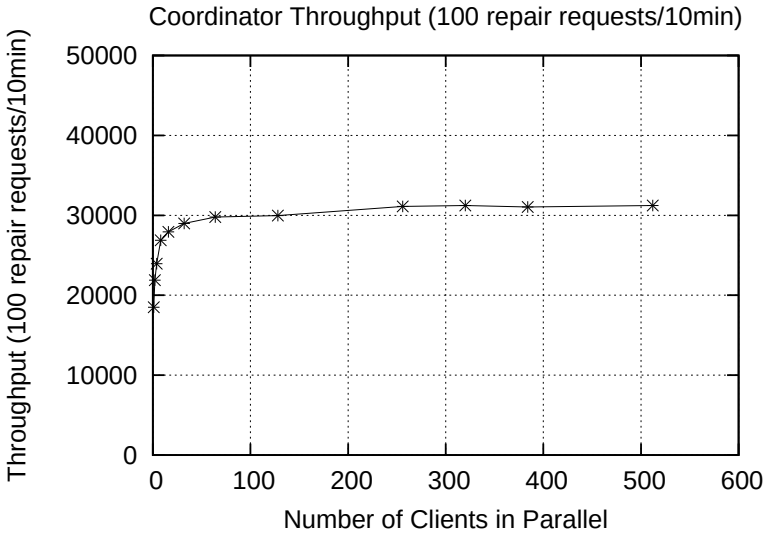


FIGURE 6.8: Coordinator throughput

6.2.3 Throughput Experiments with Failures

The experiments in this section consider the following two issues.

Firstly, given that one coordinator may become a performance bottleneck, we need to investigate the impact of contention for a coordinator on a client's recording performance[5].

[5]The impact of contention for a provenance store has been studied in [76].

In addition, as introduced in Section 6.1.1, F-PSL provides a configuration file enabling users to specify how to retransmit record messages, i.e., resending to a same store or an alternative store, where there exists a tradeoff. Retransmitting messages to the same provenance store can tolerate transient failures, such as message losses. However, if a provenance store has crashed and is to be recovered after a long period of time, resending messages to the same store is not a good solution. On the other hand, the use of an alternative store ends up with an assertor's causelinks or another assertor's viewlink incorrect. This introduces additional cost for updating links. Therefore, it is worth comparing the tradeoff between using the same and alternative store for redelivery.

We conducted two experiments where a single client and 128 clients kept recording 10k record messages into one provenance store in a 10 minute period. Various failure rates (5%, 10%, 16%, 20%, 25%, 33% and 50%) were considered[6]. A second provenance store was employed as the alternative store[7]. One coordinator service was deployed in the experiments and 100 repair requests were sent in a single batch. Since the more failure events the more repair requests, failure events were immediately generated without considering latency to maximise the number of repair requests that could be sent to the coordinator within 10 minutes.

Figure 6.9 shows the result in the experiment with a single client. The result was averaged from five runs of the experiment. We have two observations.

(1) When using the alternative store in each retransmission, up to around 20,000 repair requests are produced (because around 40,000 record messages are recorded when failure rate is 50%). This means the coordinator, in the worst case, receives 200 batches, each containing 100 repair requests, from a single client within 10 minutes. According to coordinator's throughput experiment in Section 6.2.2 and the fact that the 200 repair batches are received by the coordinator from a single client all across 10 minutes, we imply that with about 100 clients, each having its own provenance store and alternative stores, the impact of contention for a coordinator on a client's recording performance would be very small.

(2) Resending messages to the same provenance store can record more record messages than to an alternative store, assuming that only transient failures are present. This implies a bottleneck on the client side since the use of an alternative store requires extra actions on the client side to update links, limiting the number of record messages that can be sent within 10 minutes.

Figure 6.10 shows the result when 128 clients record record messages into one provenance store in the presence of failures. This experiment considers the contention for

[6]We did not consider failure rates beyond 50% because that implies severe communication problems and an application's performance would be significantly degraded, to be demonstrated in Section 6.2.5.3. In that case, the local file store can be used to temporarily maintain interaction records.

[7]When two provenance stores are used, the measured throughput in this section is actually the total number of messages sent from all clients to all provenance stores within 10 minutes.

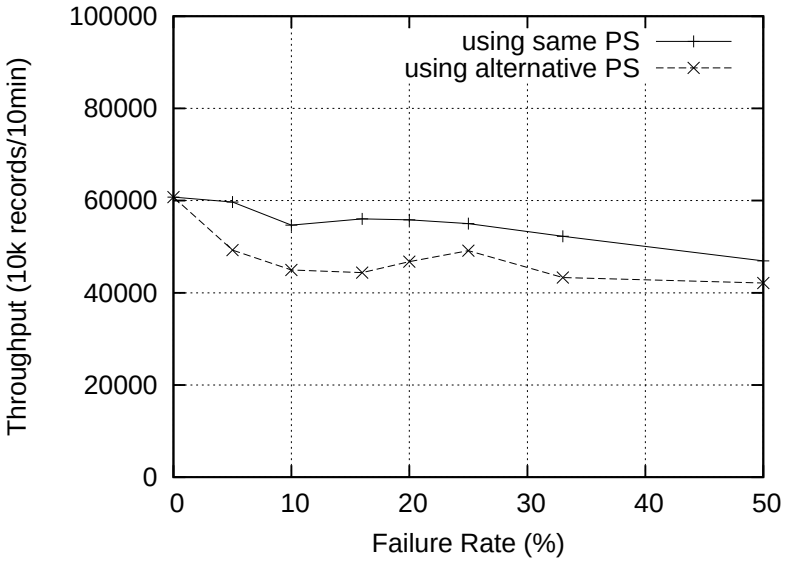


FIGURE 6.9: Throughput experiment (single client)

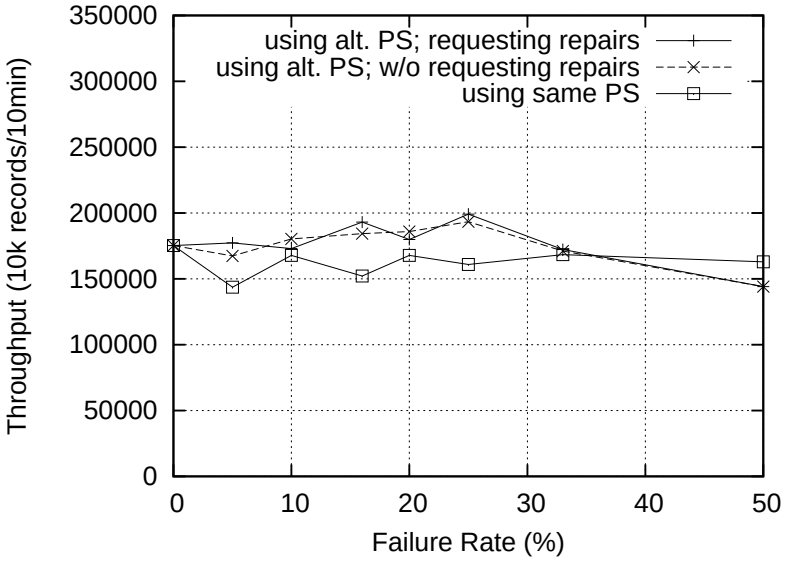


FIGURE 6.10: Throughput experiment (128 clients)

a provenance store as well as potential contention for a coordinator. We also have two observations. Firstly, communicating with the coordinator does not affect total throughput. This implies that the contention for a coordinator is negligible (It can be calculated that up to about 750 repair batches are sent to the coordinator from 128 clients in 10 minutes.). Secondly, using an alternative store, in general, results in more record messages recorded than using a same store to do so. This implies a bottleneck on the provenance store since a heavily loaded provenance store affects client's performance while the use of an alternative store helps to balance the load (especially when failure rate is 25%), though introducing additional cost of updating links.

From these experiments, we have two conclusions. Firstly, the coordinator is scalable and the impact of its contention on a client's recording performance is very small or

negligible. Since our implementation supports the use of multiple coordinators, we believe the coordinator service does not affect an application's recording performance. Secondly, to achieve a better recording performance, an alternative store can be employed after a message fails to be resent to a same store for certain times.

6.2.4 Benchmark Experiments

We now investigate the recording performance of a single client without considering contentions. All the benchmark experiments were run with one client recording record messages into one provenance store. All record messages were directly created and submitted to a provenance store without using multithreading.

Failure-free Experiment The experiment compares F-PReServ to PReServ in a failure-free environment. We measured the time to record 10,000 10k record messages. To minimise the impact of network connection overhead, 100 record messages were shipped in a single batch. Measurements were taken after recording each batch. Figure 6.11 summarises the record time. The graph displays an average from ten trials. From the figure, we have two observations:

(1) The provenance store without using disk cache, i.e., in the setup *using PReServ*, periodically flushes 900 record messages into disk. This means if the provenance store crashes, up to 900 10k record messages may be lost.

(2) The average time to record 100 10k record messages is 198.8ms and 174.4ms using F-PReServ and PReServ, respectively. Therefore, F-PReServ has an overhead of 13.8% compared to PReServ due to the use of disk cache mechanism. We note that in an application, the impact of F-PReServ on the application's performance is similar to that of PReServ in a failure-free environment, as illustrated later in the application experiment (Section 6.2.5.1). This similarity benefits from the use of multithreading to asynchronously create and record messages.

Experiment with Failures In Section 6.2.3, we measured a client's recording performance in the presence of failures in terms of throughput. However, we did not consider the overhead of updating causelinks. Updating causelinks matters only when a relationship p-assertion is to be recorded. In this experiment, we approximated the maximum overhead of taking remedial actions by measuring the record time of relationship p-assertions.

In F-PSL, the more causes a relationship p-assertion has, the longer it takes to check and update causelinks. Therefore, we increased the number of causes from 10 to 100. Given a number of causes, several tests were conducted with various failure rates (5%, 25% and 50%). For each failure rate, the p-assertions about cause interactions of a relationship

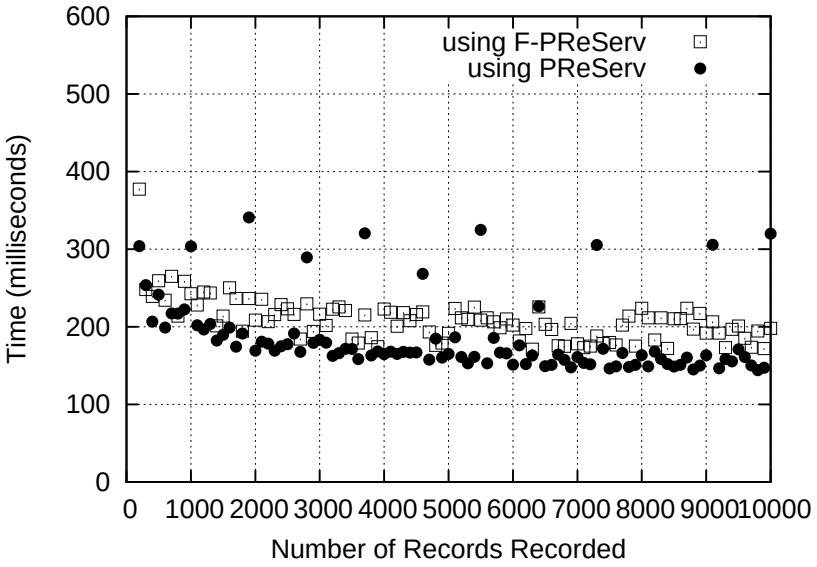


FIGURE 6.11: Time to record 100 10k interaction records

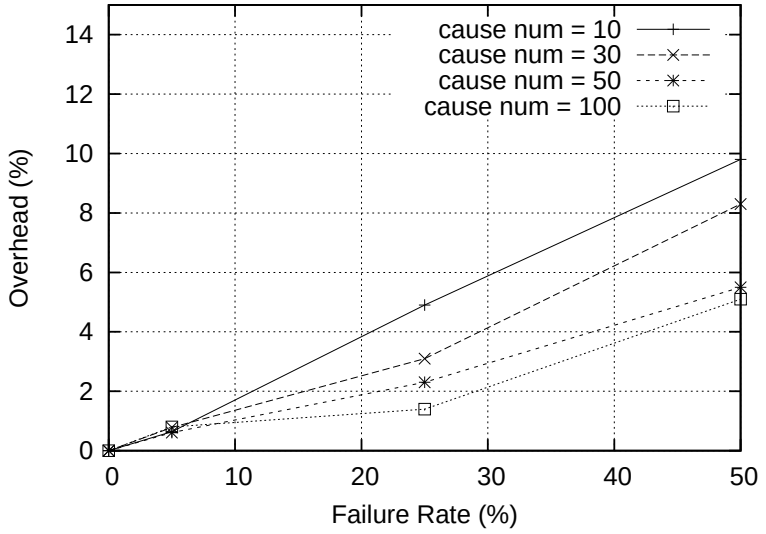


FIGURE 6.12: Overhead of taking remedial actions

p-assertion were recorded prior to measuring the recording time for the relationship p-assertion itself. In order to measure the actual cost of remedial actions by means of record time, failure events were immediately generated without considering latency. We deployed another store as an alternative store, which was used in the retransmission of a relationship p-assertion. Repair requests were sent to a coordinator in batch sizes of 100.

Figure 6.12 summarises the results in terms of overhead. The measurements were taken after recording 100 relationship p-assertions. We can observe a maximum overhead of 10% for taking remedial actions, when compared to the record time when no failure occurred. Broadly speaking, the overhead increases linearly with the increase in failure rate. We note that since it takes much longer time to record a relationship p-assertion

with larger number of causes, the overhead of taking remedial actions becomes relatively small in the settings with more causes. Therefore, we observe the smallest overhead in the setting with 100 causes. The result also shows a bounded overhead of taking remedial actions. This means that given a failure rate, the overhead does not increase as the number of causes of a relationship p-assertion increases.

6.2.5 Application Experiment

This experiment aims to investigate F-PReServ's recording performance in a scientific application, the Amino Acid Compressibility Experiment (ACE) [81]. ACE attempts to find possible new relationships between amino acids by investigating the information theoretic properties (e.g., information efficiency) of their computational representations.

ACE is chosen because of its general properties representing a range of provenance-aware applications in SOAs. First, it can be used to answer a range of provenance queries, as summarised in [81]. Second, it is high performance and fine-grained. There is no network connection in the original application before being made provenance-aware; a large volume of interaction records needs to be recorded within a short time (process documentation of 20 Gigabytes recorded within 30 minutes). These all imply that recording process documentation may be difficult. Therefore, the evaluation results obtained from this difficult application are representative to a large set of applications with less demanding requirements.

6.2.5.1 The ACE Application

The Amino Acid Compressibility Experiment (ACE), designed by Dr. Klaus-Peter Zauner and Dr.Stefan Artmann, attempts to find possible new relationships between amino acids, the basic building blocks of life, by investigating the information theoretic properties of their computational representations. ACE starts from a basic assumption that proteins are information efficient, i.e. they use the least number of amino acids possible to obtain their function. Hence, evolution results in the best and most efficient use of information. Based on this assumption, ACE tests whether particular substitutions of one amino acid for another would result in higher information efficiency. The intuition is that high information efficiency values is key to creating functioning proteins.

ACE was implemented in Tool Command Language (TCL) by Dr. Klaus-Peter Zauner and Dr.Stefan Artmann, and has been rewritten in Java [76]. In our experiment, we modified the Jave edition of the application by generating Cluster-compatible job descriptions to execute ACE on Iridis, and using F-PSL to produce and record process documentation, as illustrated in Section 6.1.1.

Figure 6.13 shows the deployment workflow of the ACE application. Selected sequences are collated locally (i.e. on the bioinformaticians computer) into several sample sequences. The samples must be of sufficient size so that the statistical methods used by the compression algorithms in the later portion of the workflow can work appropriately. The Jobs Creator then generates a series of jobs to be submitted to the Iridis cluster and executed. The executables used by the jobs are pre-staged on Iridis (i.e. they are already available on Iridis). Each job analyses several collated samples according to a set of group codings. In order to produce information efficiency values, each sample is firstly encoded with a given group coding (Encode by Groups). The recoded sequence is then compressed with compression algorithms, e.g., gzip, bzip2 or ppmz, to obtain the length of the compressed sequence (Compress). Meanwhile, the Shannon Entropy of the encoded sample is computed (Compute Entropy), which provides a standard of comparison for the encoded sample. The Shannon Entropy removes the influence of two factors from the compression: the particular data encoding used to represent the sample, and the non-uniform frequency of groups. With the compressed sample size and Shannon Entropy, the information efficiency is able to be calculated (Calculate Efficiency).

Information efficiency values can be compared since their calculation takes into account the size of the sequence used, and the compression method and group coding employed. Once the information efficiency values for different groups are calculated, they can be plotted to find those codings that maximise efficiency and thus are good candidates for further substitution investigation.

6.2.5.2 Experimental Setup

The collate sample portion of the ACE workflow is typically run once and a number of jobs are generated to process these samples with different groups. In our experimental setup, one run of ACE consisted of 20 jobs, which are run on the cluster at the same time. Each job analysed 900 unique groups on 5 different 100K collated samples, thus, a job generates 4500 information efficiency values, involving 54,000 interactions in total between seven assertors[8]. Given that one interaction is documented by two interaction records (record), one job hence produces 108,000 record messages, each containing 10Kb p-assertions on average.

To minimise network connection overhead, record and repair messages were sent in batches of 100. Multi-threading for creating and recording record messages was used in all tests and all the assertors share one request queue and one recording queue. Various failure rates (5%, 10%, 16%, 20%, 25%, 33% and 50%) were considered. The impact

[8]Local methods are instrumented using F-PSL as recording assertors. Assertors exchange application messages by means of method calls without network connections. They record interaction records documenting the messages they receive and send to contribute to the process documentation of an information efficiency value.

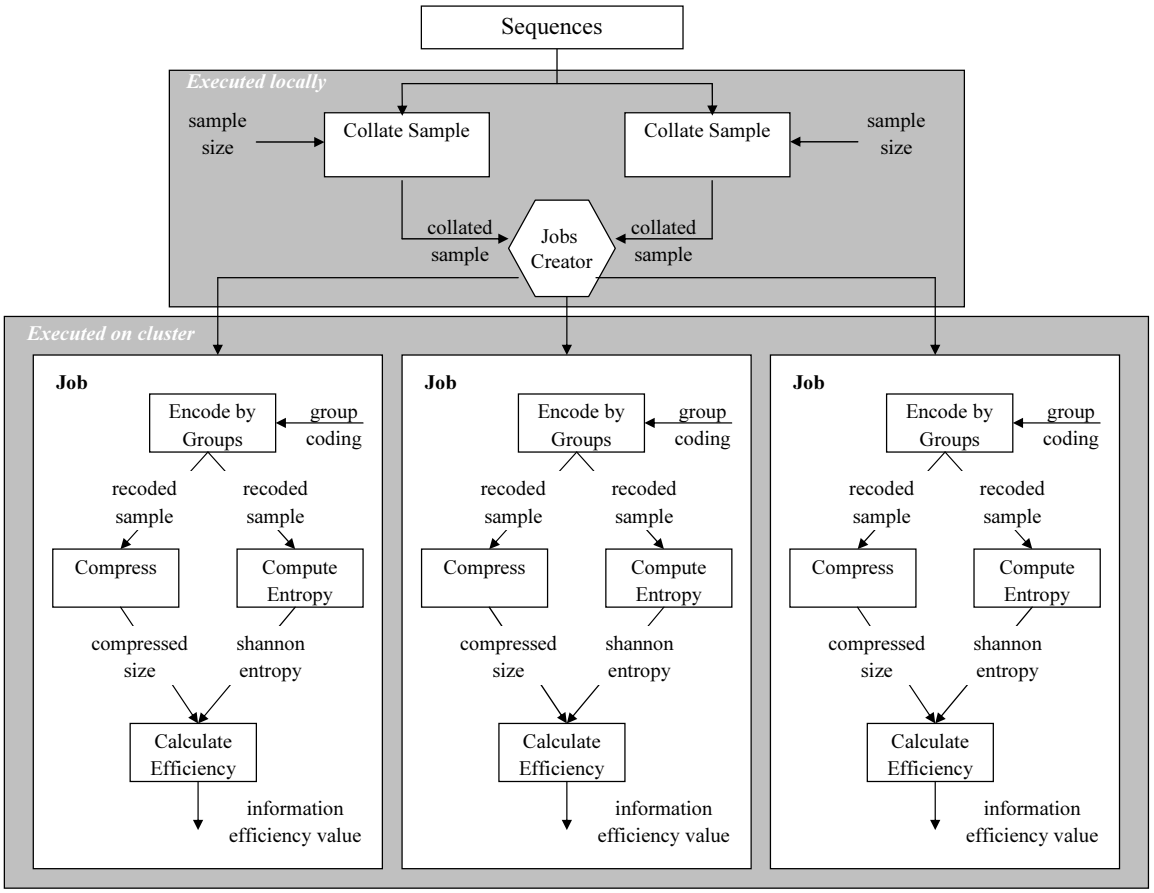


FIGURE 6.13: ACE deployment workflow [76]

of the latency to detect a failure event on an application's performance was also investigated. We studied three latency intervals, 0s, 1s and 2s to obtain a general pattern of the impact. The interval 0s provides an extreme case, where a failure event occurs immediately. Reason for not choosing higher values is that if we can conclude the pattern with these values, then we do not need to discuss higher ones.

Due to the limited resources a user is eligible to use on Iridis, we employed five provenance stores to record process documentation. These provenance stores were the default and alternative stores known by each assertor. When taking remedial actions, an alternative store was randomly selected.

6.2.5.3 Results

Since the time to collate samples is constant and small as opposed to the job runtime, the application runtime is approximated as the average runtime of all jobs [76]. We average application runtime from three runs of ACE. The runtime of an application without recording process documentation is 22:24 (in the format *mm*:*ss*). When no failure occurs, the application runtime using PReServ and F-PReServ are 24:58 and 25:07, respectively. Therefore, the recording overheads of PReServ and F-PReServ are similar

(about 12%). This similarity benefits from the use of multithreading to asynchronously record documentation.

The asynchronous approach allows an application's record messages to be queued before being shipped to a provenance store. F-PReServ has provided a flow control mechanism in the request and recording queues to avoid exhausting memory. For example, record messages cannot be queued until there is space in the recording queue. This may however affect the application's performance, since the application is postponed occasionally in order to reduce the speed of issuing requests to create record messages when queues become full frequently.

Our results in Figures 6.14 and 6.15 demonstrate the correlations among application performance, failure rate, latency of producing failure events and queue utilisation. In Figure 6.14, the recording overhead slightly increases as the failure rate is below 25% in all latency setups. However, it is sharply increased when failure rate is beyond certain points, such as 25% and 33%. Figure 6.15 shows how often a recording queue is in a full capacity when a new batch of record messages is to be enqueued. It clearly reveals that the sharp increase in the recording overhead in Figure 6.14 results from the flow control mechanism.

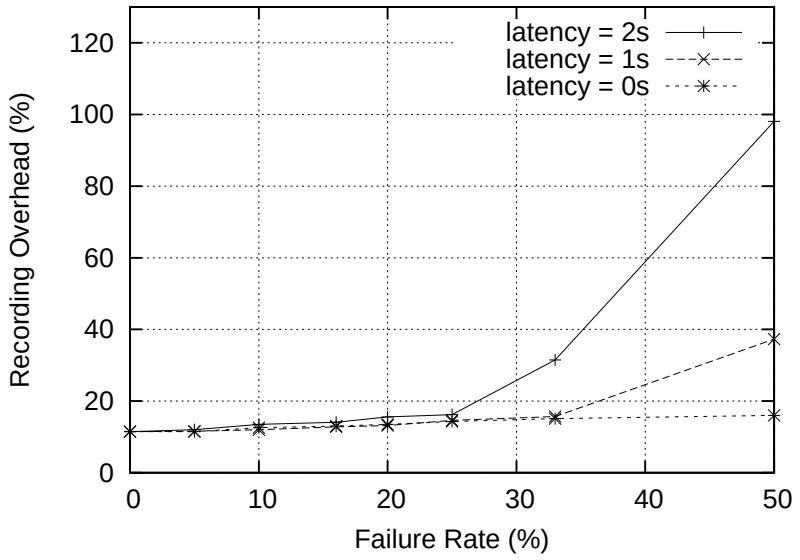


FIGURE 6.14: Recording overhead of F-PReServ

From this application experiment, we can draw several general conclusions:
(1) Both PReServ and F-PReServ have similar recording overhead when there is no failure (around 12% in ACE);
(2) If the application is not slowed down due to limit on queue capacity, F-PReServ introduces acceptable recording overhead in the presence of failures (around 18% in ACE);
(3) The latency of generating a failure event (i.e., the delay in detecting a failure) can affect an application's performance. If the latency is caused by the timeout for waiting for an acknowledgement from a provenance store, then there is a trade-off: a smaller

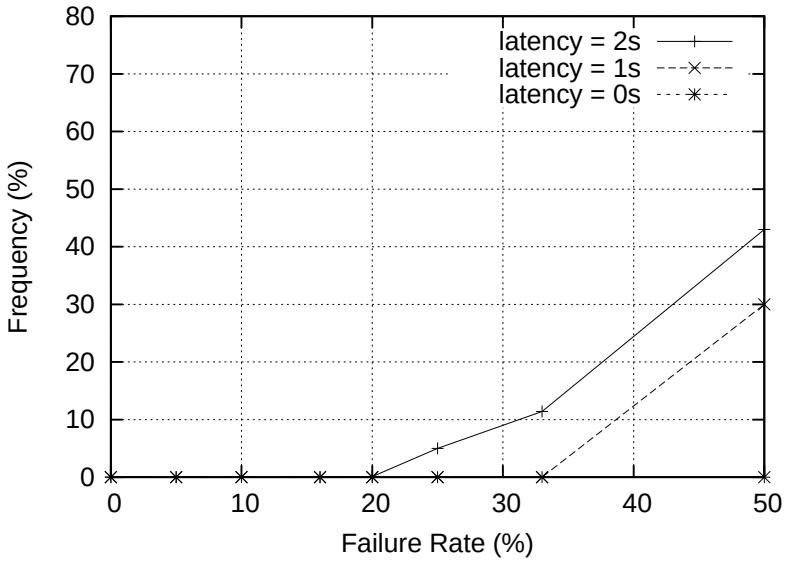


FIGURE 6.15: The frequency of a recording queue in full capacity

timeout would enable F-PReServ to take remedial actions more quickly in the presence of failures, which can avoid slowing down the application's execution but may result in redundant information recorded in the original store which erroneously was considered to have failed;
(4) By monitoring the utilisation of queues, we can detect if an application's performance has been severely degraded and then take actions to improve the performance. For example, the local file store introduced in F-PSL can be automatically employed for temporarily maintaining interaction records[9], when the frequency of the queue in maximum capacity reaches a certain threshold, say, 40%.

An ACE job does not involve network connection apart from communicating with provenance stores, the coordinator. This implies that recording process documentation is difficult and hence the above results are representative to a wide range of applications in SOAs which are normally Internet-based with less performance requirement.

Query After each run of ACE, we queried the provenance stores to further verify the quality of documentation recorded by F-PReServ. In order to compare results, we also reran ACE using PReServ to record process documentation in the presence of failures. The query results show that:
(1) F-PReServ records an equal number of interaction records in provenance stores and produced in ACE, whilst PReServ fails to record complete process documentation (Figure 6.16);
(2) F-PReServ does not produce isolated documentation islands and dangling links in the documentation retrieval path, whilst PReServ produces disconnected process documentation and dangling links (Figure 6.17);

[9]We note that the use of local file store also has performance penalty as it involves disk I/O and thread management, which causes 42% overhead in ACE experiment.

(3) Distributed documentation of the process that led to a data product (i.e., an information efficiency value) can always be retrieved in its entirety after being recorded using F-PReServ, whilst PReServ cannot as process documentation is disconnected;
(4) The retrieved process documentation can answer all the use case questions summarised in [76];

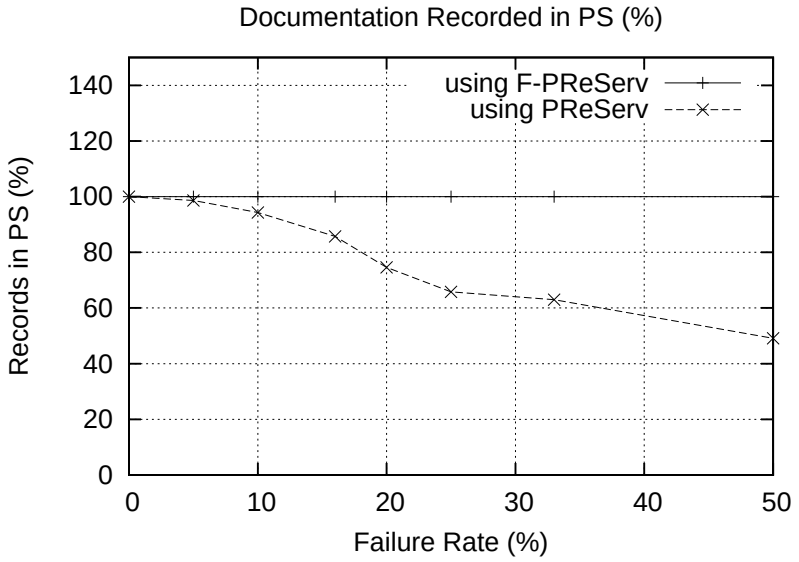


FIGURE 6.16: Number of interaction records in PS

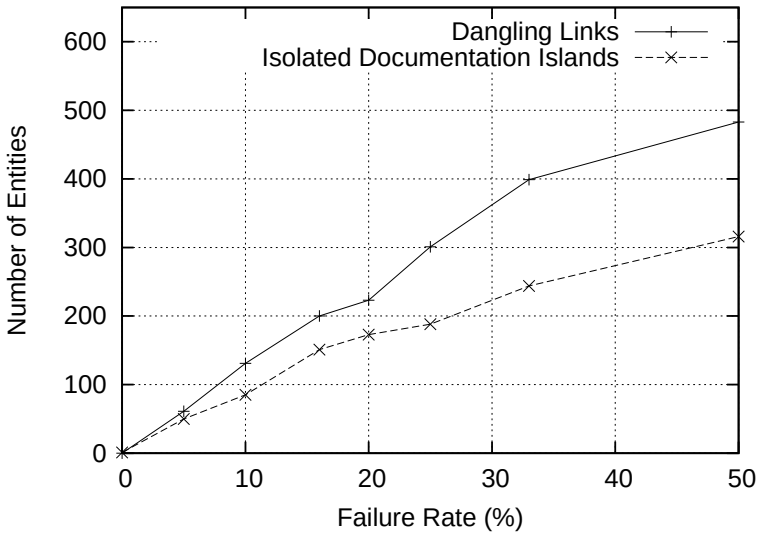


FIGURE 6.17: Isolated islands and dangling links resulted from using PReServ

6.3 Discussion

Having learned lessons from the experimental results, this section provides several recommendations on the next version of F-PReServ on achieving good performance in the case of failures. In addition, we discuss relevant technologies that can be integrated with

F-PReServ.

6.3.1 F-PSL

6.3.1.1 Including Fault Codes in Acknowledgement

The experimental results in Section 6.2.5.3 show that the latency of detecting a potential failure may affect an application's performance. In the current implementation, the receipt of an acknowledgement marks the successful recording of interaction records (IRs) in a provenance store. If exceptions are thrown in the provenance stores during the processing of received IRs, then no acknowledgement is provided. The occurrence of some exceptions on provenance stores does not guarantee the successful recording of IRs, therefore the related assertor still needs to take remedial actions. Due to lack of acknowledgement, the assertor has to wait until a timeout is expired, which implies a latency before remedial actions are taken. Consequently, the application's execution may be affected when queues reach full capacity, as illustrated in Figure 6.14.

In next version F-PReServ, we will modify the acknowledgement message by defining fault codes. Therefore, the provenance store can inform the assertor with a response message in the event of exceptions, and the corresponding assertor can interpret the fault codes in the response message and takes remedial actions as soon as possible.

6.3.1.2 Developing Intelligent Policies

In addition to F-PSL's configuration file, intelligent policies can be developed to help maintain good recording performance in future work.

(1) A timeout should be appropriately set in order to balance the tradeoff between the speed and accuracy of detecting failures to receive an acknowledgement from a provenance store. An adaptive algorithm can be developed to predict a timeout based on the size of messages to be delivered and previous failure events.

(2) As shown in the experiments, using the original and alternative store when resending messages can achieve a better performance by balancing the tradeoff between the cost of taking remedial actions, the workload of provenance stores, and the impact on application's execution. A policy can be introduced to select an alternative store after messages fail to be resent to a same provenance store for a number of times and to adjust the appropriate interval between retransmissions.

(3) In the current implementation, an alternative store is randomly chosen. However, a heartbeat service can be introduced to periodically test the candidate alternative stores during the execution of an application and choose the most suitable one in terms of service availability and speed of response.

(4) If an application's performance has been severely degraded due to recording process documentation in the presence of failures, process documentation can be temporarily maintained in the local file store. Hence, another policy can be considered to automatically employ the local store at appropriate time.

6.3.1.3 Integrating Enterprise Messaging Services

In enterprise applications, messaging services such as IBM WebSphere MQ, Tibco Enterprise JMS, or SonicMQ are often used to guarantee message delivery. When integrated with F-PReServ, enterprise messaging services can reliably transport SOAP messages provided by F-PSL.

As introduced in Section 2.6.3, a messaging service typically requires the configuration of a broker [4, 42], residing in the middle of a client application and a provenance store service. In the event of provenance store crashes, process documentation is temporarily maintained in the broker until the store comes back online. Policies can be set up for the broker to choose alternative stores. Since F-PReP provides a general and flexible approach to recording process documentation in the presence of failures, we can customise a broker's behaviour by providing actions such as sending repair requests to the coordinator service based on rules in Figure 3.14.

6.3.2 Provenance Store Service

6.3.2.1 Clustering Provenance Store Services

Provenance store services can be organised as a cluster sharing a same provenance store URL and one persistent data store. If one provenance store fails, another one takes over the pending work of the failed store so as to provide uninterrupted service to the failed store's clients as if no failure occurs.

Since our provenance store service has been implemented as a web service deployed in a web container, it is natural to adopt this approach as many application servers[10] provide high-available services through clustered web servers and databases.

Though can be complementary to our approach, clustering provenance services has the following problems. Firstly, the potentially significant cost of replicating information could limit its application, given the documentation produced in a process can be in large quantity, e.g., on the order of terabytes. Secondly, it does not deal with disconnected process documentation that could happen in the presence of failures unless each

[10]For example, Apache Tomcat, IBM WebSphere Application Server Network Deployment v6.1, WebSphere MQ Cluster, DB2 Enterprise Server Edition and Sun Cluster on Solaris.

provenance store in the system has its own clustered services. However, in open distributed environments like SOAs, where a large number of provenance stores could be present, it is unrealistic to assume each is facilitated with replicated backups.

6.3.2.2 Availability of Process Documentation

According to Figure 6.5, the provenance store caches received messages and at a later stage stores them in a backend database. This means that it may take a while before process documentation is ready to be retrieved. To accelerate this process, we can utilise multithreading in Step (3), Figure 6.5 to increase the processing rate. Therefore, process documentation will be available for retrieval more quickly.

6.3.3 Coordinator Service

6.3.3.1 Clustering Coordinator Services

In Section 3.2, we assume *the Update Coordinator does not fail*. To implement this assumption, the coordinator service can be clustered for the following reasons. Firstly, failures, though occasionally occur during recording process documentation, are still not common. Messages can be retransmitted to the same provenance store for several times before using an alternative one. Therefore, the coordinator is not involved in every interaction and the number of connections to a coordinator can be small. Secondly, only a minimum amount of information maintained by the coordinator, thus reducing the overhead for replication.

In practice, a few public coordinator services can be deployed in application servers with clustering functionality and maintained by several organisations to support all provenance-aware applications in the world.

6.3.3.2 Security Issues

Security issues pertaining to PReServ have been investigated in [77, 156]. In this section, we briefly discuss those related to the Update Coordinator. Authentication, Authorisation and Encryption are three fundamental aspects that need to considered.

- Authentication and Authorisation ensure that only verified users can communicate with a service to perform allowed operations. Firstly, only verified assertors or administrators can establish a connection with a coordinator service to send repair requests or to monitor the coordinator's status. Secondly, only verified coordinator services can update links in a provenance store. These notions are conceptually similar to the general case of accessing a database with multiple users.

- Encryption protects repair, update and uack messages from being tampered with during delivery over a connection. Transmission of messages can be secured using the Transport Layer Security (TLS) or its predecessor, Secure Sockets Layer (SSL) with HTTP.

6.4 Related Work

There is not much work on performance study related to provenance. Performance evaluations of PReServ are presented in [78, 76]. A detailed comparison on recording performance between Karma and PReServ is seen in [151], which shows that the performance of PReServ was similar to, or outperformed, Karma. Extensive performance evaluations have been made on techniques to reduce the amount of storage required for process documentation [32]. There has been a performance study on PASS [129], an automatic provenance collection and maintenance storage system at the operating system level. None of these evaluations considers failures.

6.5 Conclusion

In this chapter, we have introduced F-PReServ, an implementation of F-PReP with architectural support for practical issues such as communication, storage and performance. The implementation of F-PReServ supports requirement Efficient Recording identified in Chapter 1. We detailed the design philosophy of F-PReServ's three components: F-PSL, Provenance Store Service and Coordinator Service. Its features include a novel way of creating process documentation, basic flow control management for recording documentation, and a local store for temporarily maintaining documentation to avoid severe performance degradation in the presence of failures. We also discuss various approaches that are complementary to our implementation.

Another contribution of this chapter is the extensive evaluation of F-PReServ's performance. The experimental results showed that F-PReServ introduces reasonable overhead when compared to PReServ and has some performance impact on an application's execution. We believe these results are still acceptable given that the process documentation is guaranteed to be recorded in the presence of failures (Figure 6.16) and still retrievable in its entirety from multiple provenance stores.

Here we summarise our evaluation results:
(1) The use of *disk cache* in F-PReServ limits a provenance store's throughput by 20% compared with PReServ (Figure 6.7). However, PReServ may lose interaction records in the event of store crashes (Figure 6.11).
(2) The coordinator is scalable (Figure 6.8) and the impact of its contention on a client's

recording performance is very small or negligible (Figures 6.9 and 6.10).
(3) Remedial actions taken by F-PSL introduce small overhead (below 10%, Figure 6.12). In addition, given a failure rate, the overhead does not increase as the number of causes of a relationship p-assertion increases (Figure 6.12).
(4) F-PReServ has an overhead of 13.8% compared to PReServ when recording messages directly to a provenance store (Figure 6.11). However, the impact of F-PReServ on an application's performance is similar to that of PReServ in a failure-free environment, benefiting from the use of multithreading to asynchronously create and record messages (both having around 12% overhead compared to ACE's performance without recording process documentation, Figure 6.14).
(5) If the application is not slowed down due to the limit on a queue's capacity, F-PReServ introduces acceptable recording overhead in the presence of failures (around 18% in ACE, Figure 6.14).
(6) The latency of generating a failure event (i.e., the delay in detecting a failure) can affect an application's performance (Figure 6.14).
(7) By monitoring the utilisation of queues (Figure 6.15), we can detect if an application's performance has been severely degraded and then take actions to improve the performance. For example, the local file store introduced in F-PSL can be automatically employed for temporarily maintaining interaction records.

The scientific application ACE used in our evaluation is high performance and fine-grained, which implies that recording process documentation is difficult. Hence the above results (4), (5), (6) and (7) are representative to a wide range of applications in SOAs which are normally Internet-based with less performance requirement.

Finally, based on the above experimental results, we projected the next version of F-PReServ by recommending the improvements on its implementation to achieve better performance (Section 6.3).

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Scientific and engineering communities have presented unprecedented requirements for knowing the provenance of their data products, i.e., where they originated from, how they were produced and what has happened to them since creation. Without such important knowledge, scientists and engineers cannot reproduce, analyse or validate experiments and processes.

A number of provenance systems have been developed to record provenance information (i.e., process documentation) for SOA-based scientific and engineering applications. In order to support scalable recording, multiple provenance stores are employed to maintain distributed process documentation.

According to our survey, PASOA has a number of advantages over the other provenance systems: first, it models process documentation in a domain and technology independent approach; second, it specifies a generic recording protocol PReP, which can be implemented in different languages; third, it supports multi-site recording to obtain sufficient provenance data in highly decentralised applications; fourth, it introduces a linking mechanism to form a chain of pointers, connecting any number of provenance stores to record process documentation. These advantages have been demonstrated in a variety of applications. Therefore, this dissertation has adopted PASOA's approach to modelling process documentation and extended PReP to inherit these advantages.

None of the current provenance systems, however, adequately addresses the problem of reliably recording process documentation in SOA-like large scale environments such as the Grid, where failures (specifically, communication failures and the crash of provenance stores) often occur. The presence of these failures leads to poor quality (incomplete and disconnected) process documentation recorded in provenance stores. Consequently, it

is not acceptable in the domains that rely on process documentation to determine the provenance of their data products.

This dissertation has addressed the following thesis statement:

In SOA-based applications, the problem of recording process documentation in the presence of failures (provenance store crashes and communication failures) while still ensuring its entire retrievability is solved via a generic and efficient coordinator-based protocol to guarantee successful recording of complete documentation and to preserve accurate links that connect multiple provenance stores.

To establish this thesis, we firstly presented F-PReP (Chapter 3), a coordinator-based protocol to record interaction records (which are the elements of process documentation) in the presence of failures. Then we formalised the protocol and proved that it has properties Guaranteed Recording, Causelink Accuracy, Viewlink Accuracy (Chapter 4). After that, we investigated the properties of process documentation (Chapter 5) recorded in provenance stores by using F-PReP. We defined graph notations to intuitively demonstrate that the documentation of a whole process is guaranteed to be recorded and all viewlinks and causelinks are accurate, and more importantly, process documentation recorded in multiple interlinked provenance stores is still retrievable in its entirety. Finally, we introduced the implementation of F-PReP and conducted evaluations to show that F-PReP is efficient and has acceptable recording overhead on an application's execution (Chapter 6).

We now revisit the core contributions of this dissertation.

- We presented and formalised a generic coordinator-based recording protocol F-PReP that provides basic fault-tolerant mechanisms such as timeouts, retransmission of messages and alternative provenance stores to cope with failures. The coordinator plays a crucial role in updating viewlinks to enable documentation retrievability. Three functional requirements were identified for the protocol to record complete and retrievable distributed documentation in the presence of failures. The protocol's correctness was formally proved against each requirement using mathematical inductions.
- We graphically represented the topology of distributed process documentation spanning across interlinked provenance stores. We performed an exhaustive analysis on the forms of graphs, considering all possible topologies after documentation was recorded in the presence of failures. This exhaustive analysis helped us to demonstrate the entire retrievability of distributed process documentation.
- A system architecture F-PReServ was described, which employs F-PReP and supports practical issues such as communication, storage and performance. Its features include a novel way of creating process documentation, a new retrieval function, and implementation strategies (e.g., basic flow control management and a

local store for temporarily maintaining documentation) for achieving good performance while ensuring the reliability of recording process documentation in the presence of failures.

- An extensive evaluation of F-PReServ was performed, which reveals that it introduces acceptable recording overhead to a provenance-aware application's execution. The evaluation was conducted at several levels. First of all, we measured the throughput of the provenance store and coordinator. We demonstrated that a single coordinator does not result in a performance bottleneck. Then, we benchmarked the recording performance of F-PReServ and showed that remedial actions introduce small overhead. In addition, we investigated the performance impact on the execution time of a scientific application.

7.2 Future Work

Sections 5.5 and 6.3 have identified a number of issues for future developments and extensions. In addition to them, this section discusses several issues on future work.

7.2.1 Garbage Collection of Redundant Process Documentation

Chapter 5 has defined the notion of reachable process documentation, by following references to primary nodes. Any process documentation that is not reachable is regarded as garbage. Reclaiming garbage could be beneficial in terms of storage efficiency. Chapter 5 definition could be the basis for an algorithm for automatically reclaiming garbage.

While chapter 5 defines reachability, the automatic garbage collector needs to identify all possible roots. The investigation will have to identify these, and provide tractable means to determine them at runtime.

A simple approach is a stop-and-copy algorithm [94], but it typically requires partitioning disk space in two (from-space/to-space) which may not always be realistic. A stopping collector may not be realistic for applications that run 24/7, and an incremental approach may be desirable. A stopping algorithm however may be practical in systems, where operations terminate at the end of the day. A consolidation/compacting phase that copies provenance in a long-term archiving repository could rely on our reachability analysis.

7.2.2 Application Failures

An important research direction is to investigate the impact of application failures on the recorded process documentation. When designing F-PReP in Chapter 3, we assumed

failure-free application executions. However, assertors may crash before submitting all of their interaction records to provenance stores, thus losing part of them.

When considering application failures, the application should rely on its own fault-tolerant mechanisms[1]. For example, a process can be simply replayed in the case of application failures and our protocol ensures that process documentation is eventually recorded following the successful completion of the process. A user is more interested in the provenance information documenting the successfully completed process irrespective of how many times the process is replayed. However, there may be redundant documentation recorded in a provenance store due to repeating partial or whole process. Such redundant information can be garbage collected similarly to our discussion in Section 6.3.

7.2.3 Recording Failure Information in Process Documentation

Process documentation can capture faults or failures that occurred during the execution of a process. Such information can be used for failure diagnosis and analysis.

A few provenance systems have considered faults or failure information. For example, PASOA was used in a workflow application [167] to record instances of service invocations at run time including information such as start time/end time of invocations, memory usage and events indicating invocation failure/success etc. Recorded information is then used by scientists to evaluate experimental results. Kepler workflow system records process provenance, i.e., data related to the execution of the workflow, or intermediate data products that were processed when an error occurred [38]. By mining and analysing process provenance, users may figure out exactly what was happening at the time of an error.

Further investigation can be conducted to decide what failure information should be included as part of process documentation. Do we need to document execution exceptions? Do we need to record retried service invocations? Do we need to treat retried invocations as separate interactions? Do we need to capture the relationship between retried invocations? How do we make use of recorded fault information?

7.2.4 A Generic Link Update Mechanism

As mentioned in Section 2.2.1, three provenance challenges[2], a community effort to understand and compare systems addressing provenance, have been organised in 2006, 2007 and 2009, attracting more than 20 institutions to participate. These challenges have identified interoperability between different provenance systems as a key issue. Efforts

[1] A number of fault-tolerant mechanisms have been surveyed in Section 2.5.
[2] http://twiki.pasoa.ecs.soton.ac.uk/bin/view/Challenge/WebHome

have been made to integrate provenance information derived from different provenance systems and represented using different models. A common data model, Open Provenance Model (OPM) [125], has been proposed and revised to promote interoperability among the existing provenance systems.

These research activities clearly signal that process documentation will be generated from different systems and integrated together under a uniform model. In this sense, the ability to retrieve distributed process documentation from provenance repositories owned by different provenance systems will become essential too. We expect the linking mechanism in PASOA to be extended as a general approach to connecting provenance repositories of different systems. In addition, a coordinator-based link update algorithm can be separated from F-PReP to ensure a correct pointer chain connecting provenance repositories when a different repository is used by a participating provenance system in the presence of failures.

7.2.5 Cloud Computing

Cloud Computing is an emerging technology that attracts more and more attention nowadays. Cloud computing implies a service-oriented architecture, reduced information technology overhead for the end-user, great flexibility, reduced total cost of ownership, on-demand services and many other advantages [161]. It has been supported by a number of industry leaders such as Amazon, Google, Yahoo, Microsoft and IBM.

Cloud Computing evolves from Grid Computing with on-demand resource provisioning [59]. It relies on the Grids as its backbone and infrastructure support whilst delivering services (infrastructure, platform and software) on demand. With Cloud Computing, services are sold to users, which only pay the services they use like they pay for a public utility (e.g., electricity and gas). Therefore, users can scale up their applications to massive capacities in an instant without having to invest in new infrastructure, train new personnel, or license new software.

Clouds are providing new challenges for provenance [59]. As surveyed in Chapter 2, a number of provenance systems have been developed for Grid applications. However, provenance is still an unexplored area in Cloud environments, where we need to deal with even more challenging issues such as tracking data production across different clouds (with different platform visibility and access policies) and secure access of process documentation when provenance services (software or storage support) are sold to users which may not have control on their provenance information.

In terms of reliably recording process documentation, Cloud users would appreciate our solution. Firstly, a growing frequency of faults would be seen as Cloud computing delivers heterogenous Internet-based services. Consequently, Cloud services must be designed under assumption that they will experience frequent and often unpredictable

failures [17]. On the other hand, the Cloud is designed to be highly available as resources are created or allocated on demand. This means a large number of alternative provenance store services could be present, which supports one of our assumptions for F-PReP. Therefore, our solution will still apply.

The issues discussed in Sections 7.2.2, 7.2.3 and 7.2.4 are also interesting topics for future research in a Cloud Context.

7.2.6 Concluding Remarks

This dissertation has shown that the problem of recording retrievable process documentation in the presence of failures can be solved via a generic and efficient coordinator-based protocol to guarantee successful recording of complete process documentation and to preserve accurate links that connect multiple provenance stores for SOA-based applications.

Although we have emphasised SOA-based applications, we believe that same approach can be applied to any system whether business, engineering or scientific. By providing the reliability and retrievability of process documentation, the provenance of all the things that we deal with in our daily lives can be made available. Therefore, we will have greater confidence and knowledge about the world.

Bibliography

[1] Apache tomcat. User guide, http://tomcat.apache.org/tomcat-5.5-doc/index.html.

[2] *13th International Symposium on High-Performance Distributed Computing (HPDC-13 2004), 4-6 June 2004, Honolulu, Hawaii, USA*. IEEE Computer Society, 2004.

[3] *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), 16-19 May 2006, Singapore*. IEEE Computer Society, 2006.

[4] Sun java system message queue 4.3 technical overview. Technical report, Sun Microsystems, Inc, 2008.

[5] J R Abrial. Extending b without changing it. In *Proceeding of 1st Conference on B Method*, pages 169–191, 1996.

[6] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for event-b. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[7] Dionisis X. Adamopoulos and Constantine A. Papandreou. Enabling web services: towards service grids. In *ISCC*, pages 38–43. IEEE Computer Society, 2004.

[8] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[9] Rocio Aldeco-Perez and Luc Moreau. Provenance-based Auditing of Private Data Use. In *BCS International Academic Research Conference, Visions of Computer Science*, pages 141–152, September 2008.

[10] Ilkay Altintas and et.al. Provenance collection support in the kepler scientific workflow system. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 118–132, Chicago, Illinois, 2006.

[11] Sergio Álvarez-Napagao, Javier Vázquez-Salceda, Tamás Kifor, László Zsolt Varga, and Steven Willmott. Applying provenance in distributed organ transplant management. In Moreau and Foster [124], pages 28–36.

[12] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[13] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerance. *IEEE Transactions on Software Engineering*, 24:63–78, 1998.

[14] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[15] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[16] Narjess Ayari, Denis Barbaron, Laurent Lefèvre, and Pascale Vicat-Blanc Primet. Fault tolerance for highly available internet services: Concepts, approaches, and issues. *IEEE Communications Surveys and Tutorials*, 10(1-4):34–46, 2008.

[17] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[18] Carsten Bochner, Roland Gude, and Andreas Schreiber. A python library for provenance recording and querying. In Freire et al. [61], pages 229–240.

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

[20] Rajendra Bose, Ian T. Foster, and Luc Moreau. Report on the international provenance and annotation workshop: (ipaw'06) 3-5 may 2006, chicago. *SIGMOD Record*, 35(3):51–53, 2006.

[21] Dimitri Bourilkov, Vaibhav Khandelwal, Archis Kulkarni, and Sanket Totala. Virtual logbooks and collaboration in science and software development. In Moreau and Foster [124], pages 19–27.

[22] Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[23] Keith A. Brewster1, Daniel B. Weber2, Suresh Marru3, Kevin W. Thomas1, Dennis Gannon3, Kelvin Droegemeier1, Jay Alameda4, and Steven J. Weiss. On-demand severe weather forecasts using teragrid via the lead portal. In *TeraGrid'08*, 2008.

[24] Sharon Brunett, Karl Czajkowski, Steven Fitzgerald, Ian T. Foster, Andrew E. Johnson, Carl Kesselman, Jason Leigh, and Steven Tuecke. Application experiences with the globus toolkit. In *Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, pages 81–88, 1998.

[25] Roberto Bruni, Michael Butler, Carla Ferreira, Tony Hoare, Hernn Melgratti, and Ugo Montanari. Comparing two approaches to compensable flow composition. In *In CONCUR, LNCS 3653*, pages 383–397. Springer, 2005.

[26] Peter Buneman, Adriane Chapman, James Cheney, and Stijn Vansummeren. A provenance model for manually curated data. In Moreau and Foster [124], pages 162–170.

[27] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[28] Steve Burbeck. The tao of e-business services. Technical report, Emerging Technologies, IBM Software Group, 2000.

[29] Michael Butler, Carla Ferreira, and Muan Yong Ng. Precise modelling of compensating business transactions and its application to bpel. *Journal of Universal Computer Science*, 11(5):712–743, 2005.

[30] Michael J. Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors. *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*. Springer, 2006.

[31] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[32] A. Chapman and H.V. Jagadish. Efficient provenance storage. In *SIGMOD Conference*, pages 993–1006, June 2008.

[33] James Cheney, editor. *First Workshop on the Theory and Practice of Provenance, February 23, 2009, San Francisco, CA, USA, Proceedings*. USENIX, 2009.

[34] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In Fatma Özcan, editor, *SIGMOD Conference*, pages 942–944. ACM, 2005.

[35] Ben Clifford, Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice and Experience*, 20(5):565–575, 2008.

[36] Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Gerard Huet, Chetan Murthy, Cesar Munoz, Catherine Parent, Amokrane Saibi Benjamin Werner, Calcul Symbolique, Pascal Manoury, Pascal Manoury, Amokrane Saibi, Benjamin Werner, and Projet Coq. The coq proof assistant - reference manual v 5.10, 1995.

[37] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design, Fourth Edition*. Addison-Wesley, 2005.

[38] Daniel Crawl and Ilkay Altintas. A provenance-based fault tolerance mechanism for scientific workflows. In Freire et al. [61], pages 152–159.

[39] Flaviu Cristian. A rigorous approach to fault-tolerant programming. *IEEE Trans. Software Eng.*, 11(1):23–31, 1985.

[40] J. Crowcroft, T. Moreton, I. Pratt, and A. Twigg. *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter 29. Peer-to-Peer Technologies, page 593C622. Morgan Kaufmann, 2004.

[41] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In Jason Tsong-Li Wang, editor, *SIGMOD Conference*, pages 1345–1350. ACM, 2008.

[42] Saida Davies and Peter Broadhurst. Websphere mq v6 fundamentals. Redbook, IBM, 2005.

[43] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In Marios D. Dikaiakos, editor, *European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2004.

[44] Vikas Deora, Arnaud Contes, Omer F. Rana, Shrija Rajbhandari, Ian Wootten, Tamás Kifor, and László Zsolt Varga. Navigating provenance information for distributed healthcare management. In *Web Intelligence*, pages 859–865. IEEE Computer Society, 2006.

[45] Vijay Dialani, Simon Miles, Luc Moreau, David De Roure, and Michael Luck. Transparent fault tolerance for web services based architectures. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par*, volume 2400 of *Lecture Notes in Computer Science*, pages 889–898. Springer, 2002.

[46] Paul F. Dubois. Ten good practices in scientific programming. *Computing in Science Engineering*, 1:7–11, Jan/Feb 1999.

[47] Frantisek Dvorák, Daniel Kouril, Ales Krenek, Ludek Matyska, Milos Mulac, Jan Pospísil, Miroslav Ruda, Zdenek Salvet, Jirí Sitera, and Michal Vocu. glite job provenance. In Moreau and Foster [124], pages 246–253.

[48] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[49] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[50] Jing Fang, Wenqing Liu, and Haibo Tan. A web-based spectral database for environmental application. *Data Science Journal*, 6:898–903, 2007.

[51] Pascal Felber, Xavier Défago, Rachid Guerraoui, and Philipp Oser. Failure detectors as first class objects. In *DOA*, pages 132–141, 1999.

[52] Layna Fischer. *Workflow Handbook 2005*. Future Strategies Inc.,Lighthouse Point, FL, USA, 2005.

[53] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. Supercomputer Applications*, pages 15–18, 2001.

[54] I. T. Foster, J.-S. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *SSDBM*, pages 37–46, 2002.

[55] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[56] Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. *J. Comput. Sci. Technol.*, 21(4):513–520, 2006.

[57] Ian T. Foster, Jerry Gieraltowski, Scott Gose, Natalia Maltsev, Edward N. May, Alex Rodriguez, Dinanath Sulakhe, A. Vaniachine, Jim Shank, Saul Youssef, David Adams, Richard Baker, Wensheng Deng, Jason Smith, Dantong Yu, Iosif Legrand, Suresh Singh, Conrad Steenberg, Yang Xia, M. Anzar Afaq, Eileen Berman, James Annis, L. A. T. Bauerdick, Michael Ernst, Ian Fisk, Lisa Giacchetti, Gregory E. Graham, Anne Heavey, Joseph Kaiser, Nickolai Kuropatkin, Ruth Pordes, Vijay Sekhri, John Weigand, Yujun Wu, Keith Baker, Lawrence Sorrillo, John Huth, Matthew Allen, Leigh Grundhoefer, John Hicks, Fred Luehring, Steve Peck, Rob Quick, Stephen Simms, George Fekete, Jan vandenBerg, Kihyeon Cho, Kihwan Kwon, Dongchul Son, Hyoungwoo Park, Shane Canon, Keith R. Jackson, David E. Konerding, Jason Lee, Doug Olson, Iowa Sakrejda, Brian Tierney, Mark Green, Russ Miller, James Letts, Terrence Martin, David Bury, Catalin Dumitrescu, Daniel Engh, Robert Gardner, Marco Mambelli, Yuri Smirnov, Jens-S. Vöckler, Michael Wilde, Yong Zhao, Xin Zhao, Paul Avery, Richard Cavanaugh, Bockjoo Kim, Craig Prescott, Jorge Luis Rodriguez, Andrew Zahn, Shawn McKee, Christopher T. Jordan, James E. Prewett, Timothy L. Thomas, Horst Severini, Ben Clifford, Ewa Deelman, Larry Flon, Carl Kesselman, Gaurang Mehta, Nosa

Olomu, Karan Vahi, Kaushik De, Patrick McGuigan, Mark Sosebee, Dan Bradley, Peter Couvares, Alan DeSmet, Carey Kireyev, Erik Paulson, Alain Roy, Scott Koranda, Brian Moe, Bobby Brown, and Paul Sheldon. The grid2003 production grid: Principles and practice. In *HPDC* [2], pages 236–245.

[58] 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.

[59] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. *Computing Research Repository*, abs/0901.0131, 2009.

[60] T.S. Franks. The age old problem: Provenance. *Educational: Importance of Provenance - Fine Art Registry Exclusive Articles*, Accessed Online, 2007.

[61] Juliana Freire, David Koop, and Luc Moreau, editors. *Provenance and Annotation of Data and Processes, Second International Provenance and Annotation Workshop, IPAW 2008, Salt Lake City, UT, USA, June 17-18, 2008. Revised Selected Papers*, volume 5272 of *Lecture Notes in Computer Science*. Springer, 2008.

[62] Juliana Freire, Cláudio T. Silva, Steven P. Callahan, Emanuele Santos, Carlos Eduardo Scheidegger, and Huy T. Vo. Managing rapidly-evolving scientific workflows. In Moreau and Foster [124], pages 10–18.

[63] James Frew and Rajendra Bose. Earth system science workbench: A data management infrastructure for earth science products. In *SSDBM*, pages 180–189. IEEE Computer Society, 2001.

[64] James Frew, Dominic Metzger, and Peter Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience*, 20(5):485–496, 2008.

[65] James Frey, Todd Tannenbaum, Miron Livny, Ian T. Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[66] Jeremy G. Frey, Mark Bradley, Jonathan W. Essex, Michael B. Hursthouse, Susan M. Lewis, Michael M. Luck, Luc Moreau, Dave C. De Roure, Mike Surridge, and Alan Welsh. *Grid Computing — Making the Global Infrastructure a Reality*, chapter Combinatorial chemistry and the Grid, pages 945–962. Wiley Series in Communications Networking and Distributed Systems. John Wiley and Sons, Chichester, England, 2003.

[67] F. Gagliardi, B. Jones, F. Grey, M. E. Bgin, and M. Heikkurinen. 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):1729C1742, August 2005.

[68] Stephen J. Garland and Nancy Lynch. Using i/o automata for developing distributed systems. In *In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[69] Felix C. Gartner. Specifications for fault tolerance: A comedy of failures. Technical report, Darmstadt University of Technology, 1998.

[70] Felix C. Gartner. An exercise in systematically deriving fault-tolerance specifications. In *In Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS)*, 1999.

[71] Felix C. Gartner. A survey of transformational approaches to the specification and verification of fault-tolerant systems. *Journal of Universal Computer Science (J.UCS)*, 5:94–103, 1999.

[72] Yolanda Gil, Ewa Deelman, Mark H. Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole A. Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):24–32, 2007.

[73] Carole Goble, Chris Greenhalgh, Steve Pettifer, and Robert Stevens. *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter 9. Knowledge Integration: In Silico Experiments in Bioinformatics, page 121. Morgan Kaufmann, 2004.

[74] G. Graham, R. Cavanaugh, P. Couvares, A. D. Smet, and M. Livny. *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter 10. Distributed Data Analysis: Federated Computing for High-Energy Physics, pages 95–108. Morgan Kaufmann, 2004.

[75] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[76] Paul Groth. 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.

[77] 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.

[78] Paul Groth, Simon Miles, Weijian Fang, Sylvia C. Wong, Klaus-Peter Zauner, and Luc Moreau. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the 14th IEEE International Symposium on High*

Performance Distributed Computing (HPDC'05), pages 201–208, Research Triangle Park, North Carolina, July 2005.

[79] Paul Groth, Simon Miles, and Luc Moreau. Preserv: Provenance recording for services. In *Proceedings of the UK OST e-Science Fourth All Hands Meeting (AHM05)*, 2005.

[80] Paul Groth, Simon Miles, and Luc Moreau. A Model of Process Documentation to Determine Provenance in Mash-ups. *Transactions on Internet Technology (TOIT)*, 9(1):1–31, 2009.

[81] Paul Groth, Simon Miles, and Luc Moreau. A Model of Process Documentation to Determine Provenance in Mash-ups. *Transactions on Internet Technology (TOIT)*, 9(1):1–31, 2009.

[82] Paul Groth and Luc Moreau. Recording process documentation for provenance. *IEEE Transactions on Parallel and Distributed Systems*, In publication, September 2009.

[83] Harvey W. Gunther. Websphere application server development best practices for performance and scalability. White paper, IBM, 2000.

[84] Theo Harerder and Andreas Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15:287–317, 1983.

[85] Naohiro Hayashibara, Adel Cherif, and Takuya Katayama. Failure detectors for large-scale distributed systems. In *SRDS*, pages 404–409. IEEE Computer Society, 2002.

[86] William Hoarau and Sbastien Tixeuil. A language-driven tool for fault injection in distributed applications. In *Proceedings of the IEEE/ACMWorkshop GRID*, November 2005.

[87] Victor P. Holmes, Wilbur R. Johnson, and David J. Miller. Integrating metadata tools with the data services archive to provide web-based management of large-scale scientific simulation data. In *Annual Simulation Symposium*, pages 72–79. IEEE Computer Society, 2004.

[88] Yi Huang, Aleksander Slominski, Chathura Herath, and Dennis Gannon. Ws-messenger: A web services-based messaging system for service-oriented grid computing. In *CCGRID* [3], pages 166–173.

[89] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole A. Goble, Matthew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006.

[90] Soonwook Hwang and Carl Kesselman. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing*, 1(3):251–272, 2003.

[91] John Ibbotson. Provenance and compliance. Technical report, IBM UK, 2006.

[92] Keith R. Jackson. pyglobus: a python interface to the globus toolkit. *Concurrency and Computation: Practice and Experience*, 14(13-15):1075–1083, 2002.

[93] N. Jacq, J. Salzemann, Y. Legre, M. Reichstadt, F. Jacq, M. Zimmermann, A. Maass, M. Sridhar, K. Vinod-Kusam, H. Schwichtenberg, M. Hofmann, and V. Breton. Demonstration of in silico docking at a large scale on grid infrastructure. *Studies in health technology and informatics*, pages 155–157, 2006.

[94] Richard E. Jones and Rafael Lins, editors. Wiley, 1996.

[95] Arnaud Kerhornou and Roderic Guigó. Biomoby web services to support clustering of co-regulated genes based on similarity of promoter configurations. *Bioinformatics*, 23(14):1831–1833, 2007.

[96] Guy K. Kloss and Andreas Schreiber. Provenance implementation in a scientific simulation environment. In Moreau and Foster [124], pages 37–45.

[97] Ales Krenek, Jirí Sitera, Ludek Matyska, Frantisek Dvorák, Milos Mulac, Miroslav Ruda, and Zdenek Salvet. glite job provenance - a job-centric view. *Concurrency and Computation: Practice and Experience*, 20(5):453–462, 2008.

[98] Zoé Lacroix and Hervé Ménager. Semanticbio: Building conceptual scientific workflows over web services. In Bertram Ludäscher and Louiqa Raschid, editors, *DILS*, volume 3615 of *Lecture Notes in Computer Science*, pages 296–299. Springer, 2005.

[99] L. Laibinis and E. Troubitsyna. Refinement of fault tolerant control systems in b. In Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, editors, *SAFECOMP*, volume 3219 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2004.

[100] Linas Laibinis, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky. Rigorous development of fault-tolerant agent systems. In Butler et al. [30], pages 241–260.

[101] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[102] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[103] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6:254–280, 1984.

[104] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[105] D.P. Lanter. Lineage in gis: The problem and a solution. Technical Report 90-6, National Center for Geographic Information and Analysis (NCGIA),UCSB, Santa Barbara, CA, 1991.

[106] G. Latif-Shabgahi, J.M. Bass, and S. Bennett. History-based weighted average voter: a novel software voting algorithm for fault-tolerant computer systems. In *Proceedings of 9th IEEE Euromicro Workshop on Parallel and Distributed Processing*, pages 402–409, Mantova,Italy, 2001.

[107] Zhangxi Lin, Sathya Ramanathan, and Huimin Zhao. Usage-based dynamic pricing of web services for optimizing resource allocation. *Inf. Syst. E-Business Management*, 3(3):221–242, 2005.

[108] Nik Looker, Malcolm Munro, and Jie Xu. Ws-fit: A tool for dependability analysis of web services. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, 2004.

[109] Nik Looker and Jie Xu. Dependability assessment of grid middleware. In *DSN*, pages 125–130. IEEE Computer Society, 2007.

[110] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[111] D. Luna. Creation of a provenance-aware rss system. Technical report, University of Southampton, May 2007.

[112] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 137–151, 1987.

[113] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[114] Nayden Markatchev, Cameron Kiddle, and Rob Simmonds. A framework for executing long running jobs in grid environments. In *High Performance Computing Symposium*, pages 69–75. IEEE Computer Society, 2008.

[115] Brian McBride. The resource description framework (rdf) and its vocabulary description language rdfs. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 51–66. Springer, 2004.

[116] Roger Menday. The web services architecture and the unicore gateway. In *AICT/ICIW*, page 134. IEEE Computer Society, 2006.

[117] Simon Miles, Paul Groth, Miguel Branco, and Luc Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 5(1):1–25, 2007.

[118] Simon Miles, Paul Groth, Ewa Deelman, Karan Vahi, Gaurang Mehta, and Luc Moreau. Provenance: The bridge between experiments and data. *Computing in Science and Engineering*, 10(3):38–46, May/June 2008.

[119] Luc Moreau. Distributed Directory Service and Message Router for Mobile Agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[120] Luc Moreau. 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*, pages 93–100, Madrid, Spain, March 2002.

[121] Luc Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. *Scalable Computing: Practice and Experience*, 7(4):53–87, December 2006.

[122] Luc Moreau, Peter Dickman, and Richard Jones. Birrell's Distributed Reference Listing Revisited. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):52, 2005.

[123] Luc Moreau and Jean Duprat. A Construction of Distributed Reference Counting. *Acta Informatica*, 37:563–595, 2001.

[124] Luc Moreau and Ian T. Foster, editors. *Provenance and Annotation of Data, International Provenance and Annotation Workshop, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers*, volume 4145 of *Lecture Notes in Computer Science*. Springer, 2006.

[125] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In Freire et al. [61], pages 323–326.

[126] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. The Provenance of Electronic Data. *Communications of the ACM*, 51(4):52–58, April 2008.

[127] Luc Moreau, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven P. Callahan, George Chin Jr., Ben Clifford, Shirley Cohen, Sarah Cohen Boulakia, Susan B. Davidson, Ewa Deelman, Luciano A. Digiampietri, Ian T. Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole A. Goble, Jennifer Golbeck, Paul T. Groth, David A. Holland, Sheng Jiang, Jihie

Kim, David Koop, Ales Krenek, Timothy M. McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale, Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Eduardo Scheidegger, Karen Schuchardt, Margo I. Seltzer, Yogesh L. Simmhan, Cláudio T. Silva, Peter Slaughter, Eric G. Stephan, Robert Stevens, Daniele Turi, Huy T. Vo, Michael Wilde, Jun Zhao, and Yong Zhao. Special issue: The first provenance challenge. *Concurrency and Computation: Practice and Experience*, 20(5):409–418, 2008.

[128] Sape Mullender. *Distributed Systems*. Addison-Wesley, 1993.

[129] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56. USENIX, 2006.

[130] J. Myers, C. Pancerella, C. Lansing, K. Schuchardt, and B. Didier. Multi-scale science, supporting emerging practice with semantically derived provenance. *Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, 2003.

[131] J. D. Myers, T. C. Allison, S. Bittner, B. T. Didier, M. Frenklach, W. S. Koegler, C. Lansing, D. Leahy, M. Lee, R. McCoy, M. Minkoff, S. Nijsure, G. v. Laszewski, D. Montoya, C. M. Pancerella, R. Pinzon, W. Pitz, L. A. Rahn, B. Ruscic, K. Schuchardt, E. Stephan, T. L. Windus, and C. L. Yang. A collaborative informatics infrastructure for multiscale science. *CLADE*, page 24, 2004.

[132] James D. Myers, Carmen Pancerella, Carina Lansing, Karen L. Schuchardt, and Brett Didier. Multi-scale science: supporting emerging practice with semantically derived provenance. In *In ISWC 2003 Workshop: Semantic Web Technologies for Searching and Retrieving Scientific Data*, 2003.

[133] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.

[134] Savas Parastatidis, Jim Webber, Paul Watson, and Thomas Rischbeck. Ws-gaf: a framework for building grid applications using web services. *Concurrency - Practice and Experience*, 17(2-4):391–417, 2005.

[135] M. Pasin, S. Fontaine, and S. Bouchenak. Failure detection in large scale systems: a survey. In *Network Operations and Management Symposium Workshops (NOMS Workshops 2008)*, pages 165–168, 2008.

[136] Laura Pearlman, Carl Kesselman, Sridhar Gullapalli, B. F. Spencer Jr., Joe Futrelle, Kathleen Ricker, Ian T. Foster, Paul Hubbard, and Charles Severance. Distributed hybrid earthquake engineering experiments: Experiences with a ground-shaking grid application. In *HPDC* [2], pages 14–23.

[137] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12(3):477–482, 1986.

[138] Beth Plale, Dennis Gannon, Jerry Brotzge, Kelvin Droegemeier, James F. Kurose, David McLaughlin, Robert Wilhelmson, Sara J. Graves, Mohan Ramamurthy, Richard D. Clark, Sepi Yalda, Daniel A. Reed, Everette Joseph, and V. Chandrasekar. Casa and lead: Adaptive cyberinfrastructure for real-time multiscale weather forecasting. *IEEE Computer*, 39(11):56–64, 2006.

[139] Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. Fault-tolerant behavior in state-of-the-art grid workflow management systems. CoreGRID Technical Report TR-0091, October,2007.

[140] S. Rajbhandari, A. Contes, O. F.Rana, V. Deora, and I. Wootten. Establishing workflow trust using provenance information. In *1st IEEE International Workshop on Modelling Autonomic Communications Environments (MACE 2006)*, pages 21–30, October 2006.

[141] Christine F. Reilly and Jeffrey F. Naughton. Exploring provenance in a distributed job execution system. In Moreau and Foster [124], pages 237–245.

[142] Christine F. Reilly and Jeffrey F. Naughton. Transparently gathering provenance with provenance aware condor. In Cheney [33].

[143] Michel Reynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70, 2005.

[144] Birgit Roehm, Gang Chen, Andre de Oliveira Fernandes, Cristiane Ferreira, Rodney Krick, Denis Ley, Robert R. Peterson, Gerhard Poul, Joshua Wong, and Ruibin Zhou. Websphere application server v6 scalability and performance handbook. Redbook, IBM, 2005.

[145] Birgit Roehm, Adeline Chun, William Joly, Tim Klubertanz, Li-Fang Lee, Hong Min, Yoshiki Nakajima, Nagaraj Nunna, Terry OBrien, Kristi Peterson, Jens Rathgeber, and Michael Schmitt. Websphere application server network deployment v6: High availability solutions. Redbook, IBM, 2005.

[146] Jorge Luis Romeu. Data quality and pedigree. *Material Ease*, 1999.

[147] Fred B. Schneider. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.*, 4(2):125–148, 1982.

[148] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[149] Yogesh Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[150] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. Karma2: Provenance management for data-driven workflows. *International Journal of Web Services Research (JWSR)*, 5(2):1–22, 2008.

[151] Yogesh L. Simmhan, Beth Plale, Dennis Gannon, and Suresh Marru. Performance evaluation of the karma provenance framework for scientific workflows. In Moreau and Foster [124], pages 222–236.

[152] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley Sons, Ltd., 2005.

[153] Richard Spillane, Russell Sears, Chaitanya Yalamanchili, Sachin Gaikwad, Manjunath Chinni, and Erez Zadok. Story book: An efficient extensible provenance framework. In Cheney [33].

[154] Alexander S.Szalay and Jim Gray. *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter 7. Scientific Data Federation: The World-Wide Telescope, pages 95–108. Morgan Kaufmann, 2004.

[155] Martin Szomszor and Luc Moreau. Recording and reasoning over data provenance in web and grid services. In *International Conference on Ontologies, Databases and Applications of SEmantics (ODBASE'03)*, volume 2888 of *Lecture Notes in Computer Science*, pages 603–620, Catania, Sicily, Italy, November 2003.

[156] Victor Tan, Paul Groth, Simon Miles, Sheng Jiang, Steve Munroe, Sofia Tsasakou, and Luc Moreau. Security issues in a soa-based provenance system. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 203–211, Chicago, Illinois, 2006. Springer-Verlag.

[157] Paul Townend, Paul T. Groth, and Jie Xu. A provenance-aware weighted fault tolerance scheme for service-based applications. In *ISORC*, pages 258–266. IEEE Computer Society, 2005.

[158] Paul Townend, Nik Looker, Dacheng Zhang, Jie Xu, Jianxin Li, Liang Zhong, and Jinpeng Huai. Crown-c: A high-assurance service-oriented grid middleware system. In *HASE*, pages 35–44. IEEE Computer Society, 2007.

[159] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *CoRR*, abs/cs/0501002, 2005.

[160] Hagen Völzer. Verifying fault tolerance of distributed algorithms formally - an example. In *ACSD*, pages 187–196. IEEE Computer Society, 1998.

[161] Mladen A. Vouk. Cloud computing: Issues, research and implementations. In *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, pages 31–40, August 2008.

[162] Ting Wang, Jochem Vonk, Benedikt Kratz, and Paul W. P. J. Grefen. A survey on the history of transaction management: from flat to grid transactions. *Distributed and Parallel Databases*, 23(3):235–270, 2008.

[163] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[164] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*, pages 262–276, 2005.

[165] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[166] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In W. A. Gray and Per-rAke Larson, editors, *ICDE*, pages 91–102. IEEE Computer Society, 1997.

[167] Ian Wootten, Shrija Rajbhandari, Omer F. Rana, and Jaspreet Singh Pahwa. Actor provenance capture with ganglia. In *CCGRID* [3], pages 99–106.

[168] Jie Xu, Brian Randell, Alexander B. Romanovsky, Robert J. Stroud, Avelino F. Zorzo, Ercument Canver, and Friedrich W. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Trans. Computers*, 51(2):164–179, 2002.

[169] Jie Xu, Paul Townend, Nik Looker, and Paul T. Groth. Ft-grid: a system for achieving fault tolerance in grids. *Concurrency and Computation: Practice and Experience*, 20(3):297–309, 2008.

[170] Divakar Yadav and Michael Butler. Rigorous design of fault-tolerant transactions for replicated database systems using event b. In Butler et al. [30], pages 343–363.

[171] Divakar Yadav and Michael Butler. Formal development of a total order broadcast for distributed transactions using event-b. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 152–176. Springer, 2009.

[172] Jian Yang and Mike P. Papazoglou. Web component: A substrate for web service reuse and composition. In Anne Banks Pidduck, John Mylopoulos, Carson C. Woo, and M. Tamer Özsu, editors, *CAiSE*, volume 2348 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2002.

[173] Demetrios Zeinalipour-Yazti, Kyriakos Neocleous, Chryssis Georgiou, and Marios D. Dikaiakos. Identifying failures in grids through monitoring and ranking. In *NCA*, pages 291–298. IEEE Computer Society, 2008.

[174] Jun Zhao, Carole A. Goble, and Robert Stevens. An identity crisis in the life sciences. In Moreau and Foster [124], pages 254–269.

[175] 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.

[176] Yong Zhao, Michael Wilde, Ian T. Foster, Jens-S. Vöckler, James E. Dobson, Eric Gilbert, Thomas Jordan, and Elizabeth Quigg. Virtual data grid middleware services for data-intensive science. *Concurrency and Computation: Practice and Experience*, 18(6):595–608, 2006.