FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Agile Provenance

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

E937 Software Engineering/Internet Computing

eingereicht von

Andreas Happe
Matrikelnummer 0226373

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Univ. Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Univ. Ass. Dipl.-Ing. Lukasz Juszczyk und Dr.techn. Hong-Linh Truong

Wien, 19.4.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Andreas Happe, Schäffergasse 20/15, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19.4.2010 _____

(Unterschrift Verfasser/in)

Abstract

Provenance describes how objects came into their current state, e.g. it describes different contributions to a document. This information provides detailed audit trails, verification of existing objects as well as reenactment of recorded activities. Provenance solutions consist of data gathering, storage and analysis, its implementation is delegated to domain application developers. This leads to custom provenance solutions that do not adhere to standards and are expensive to maintain. The domain developers' productivity also suffers.

We propose a generic provenance system that can be adapted for different domain applications. It employs advanced unobtrusive data capturing techniques to minimize overhead imposed upon domain developers. Provenance data is refined and results are provided to developers through an easily usable interface. This allows domain developers to focus upon their domain tasks.

To prove the feasibility of our approach a prototype system has been implemented within Ruby on Rails. As domain application an existing experiment management solution was chosen and made provenance-aware.

Zusammenfassung

Provenienz (engl. provenance) beschreibt die Entstehungsgeschichte von Objekten. Die Provenienz eines Dokumentes beinhaltet beispielsweise jede zwischenzeitliche Dokumentenversion, deren Autoren als auch die jeweiligen Dokumentäenderungen. Diese Information erlaubt das Erstellen von detailierten Zugriffsberichten, die Validierung bestehender Objekte als auch das Wiederherstellen alter Objektversionen. Provenienz-Architekturen beinhalten Komponenten zur Sammlung, Speicherung und Analyse von Provenienz, deren Implementierung ist zumeist Aufgabe der Anwendungsprogrammierer. Diese Situation führt zu Insellösungen die nicht standard-konform sind und deren Wartung langfristig hohe Kosten verursacht.

Im Zuge dieser Diplomarbeit wird ein generisches Provenienzsystem vorgestellt dessen Fokus auf der Integration mit bestehenden Systemen liegt. Durch Techniken wie "unobtrusive data capturing" wird der verursachte Overhead und Produktivitätsverlust minimiert. Die genereriten Daten werden überarbeitet, persistiert und an Analyseapplikationen angeboten.

Die Durchführbarkeit unserer Architektur wurde anhand eines Prototypsystems getestet. Als Softwareumgebung fuer den Prototypen wurde ein dynamisches web-basiertes Applikations-Framework, Ruby on Rails, gewählt. Als "use-case" wurde ein bestehendes Experiment-Management-System (Genesis) adaptiert und um Provenienz-Funktionalität erweitert.

Agile Provenance*

Andreas Happe

19.4.2010

*dedicated to Rudolf and Maria Happe

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Provenance

*p*rovenance
Pronunciation: \präv-nn(t)s, prä-v-nän(t)s\
Function: noun
Etymology: French, from provenir to come forth, originate, from Latin provenire, from pro- forth + venire to come
Date: 1785

1. origin , source
2. the history of ownership of a valued object or work of art or literature

Figure 1.1: Merriam-Webster's definition of provenance

Provenance describes how artefacts came into their current state. An artefact can be any object managed by computer software: a language-dependent object, documents or source code.

Provenance is a well-known problem in the physical world: archaelogist and scientists in general must confirm the originiality of their published works. In courts evidence must be presented with an unbroken chain of custody to be valid. Provenance has been addressed within different fields of computer science, from being used for capturing intellectual property rights [15] to being used within the bio-informatics field [24]. Generally provenance information is used to provide audit trails of performed operations, allows reasoning about artefact's linage [54] or recreation of older artefact states [12;54].

The following use cases highlight provenance's importance: In the financial world account changes must be recorded. The data origin of reconciled reports is of importance for later reevaluations. A provenance system automatically gathers the needed data and allows the developer to focus on the business domain.

In the intelligence world provenance of dossiers is important. While authenticity of provenance must be provided the dossier's sources must still be hidden from readers. One example that shows the importance of a dossier's source can be found in "On Homeland Security and the Semantic Web" [19]

which describes a system that utilizes the semantic net to infer intelligence information. The same requirement can also be found in business use cases, for example reviewers of job reviews should be valid but stay anonymous.

Access to medical files must be recorded and evaluated[32 1]. The origin of medical results must be traceable and last but not least when a decision was made the medical base for this decision should be recorded. Recently the multi-agent paradigm has been applied to health care systems[33], in such a decentralized system provenance of data changes is of even higher importance.

1.2 Benefactors of Provenance

Provenance is seldom a means in its own end but rather an aspect of a larger domain application. Generally speaking activities that transform input data into new data gain from provenance. This includes business applications that alter bank accounts as well as most analytical processes that reduce original data into a concise report (maybe find references). This section highlights various areas in which provenance is utilized.

Security and Safety

Legal documents need provenance. In the physical world we are convinced that a once submitted offer cannot be easily altered and provenance is verified by solicitors. In the virtual world a written document can easily be changed without anyone noticing as long as there are no provenance and security checks in place.

Example: a cost calculation for a project was established between different parties. Months later the project is overdue and one party believes that the calculation was altered to cover the increased spending. A provenance solution is able to guarantee that the calculation was not altered.

Provenance data gathering is related to monitoring. The former can utilize the later for capturing base provenance data upon which analysis can performed. Gained data shares common characteristics with regard to safety, security and access control.

Example: A performance review for an employee was written and published. While it must be verify-able that its authors are allowed to write the report the direct identity of the authors should not be revealed.

Each process that compacts data into a concise end result must reduce incoming data. For example an scientific experiment performs statistical methods upon multiple input data sets to gain one single computed output datum. Through time the performed steps might change and the scientist is responsible for recording each experiment with all corresponding parameters. This task can be transfered to a provenance system and saves precious time.

Example: a medical researcher performs various experiments upon collected data. After some time period an erogenous sensor is detected, some captured data was corrupted. Provenance system now show which subsequent experiments were affected and have to be redone.

Development Process

Provenance systems impose initial overhead upon software development. In turn provenance also helps with software development. A time consuming task during application development is debugging which often boils down to watch a system react to input parameters. This can take place through augmenting the code with output statements or utilizing sophisticated tools as tracing frameworks. A comprehensive provenance framework especially provides this tracing framework.

Questions asked during the first provenance challenge mirror high-level debug problems. How an object came into being is a typical provenance question, it can also be applied upon a software object which is examined by a software developer as it has invalid content. Dynamic languages increase provenance usefulness as less validations can be done statically. The preferred approach using those languages, test- or behavior driven development, even aggravate this.

Example: A developer wants to change the structure of an software object but is not sure which results this will impose on the overall workflow. Provenance solutions can tell him which actions and artifacts are derived from the to be altered object.

Optimization

The inherent monitoring and tracing capabilities help with optimization problems. Questions that commonly arise during distributed system deployment focus on data distribution: which systems access which artifacts? Provenance shows how to distribute data artifacts more efficiently and how to minimize access times. On a smaller scale access-patterns are important. The question "which read or write patterns lead to data changes?" often shows redundant data access which impacts performance.

Example: provenance can show that support documents and functionality are frequently used by a special help desk section. The overall performance can be optimized by locating those documents and tools nearer to their most frequent callers

Regulations

Regulations mandate compulsory data archival. Legal documents must be stored in a "safe" manner that prevents tampering with them for a well-defined period of time to prevent fraud. Companies comply through complicated storage procedures – which cost precious resources. Cryptographically secured provenance storage provides the needed level of safety and security.

1.3 Open Problems

The prior chapter has shown the need for provenance but provenance solutions are not in wide-spread use. The following problems explain this discrepancy:

Integration with existing domain systems is mandatory. Provenance systems must be integrated with domain-specific systems. The domain system must be augmented with functionality that provides the means of provenance gathering. Various means of achieving this differ in quality and quantity of gathered provenance data as well as in their imposed overhead upon performance and the application developer's agility.

The problem is increased through the multi-system nature of provenance systems. One provenance system captures data from various sources which provide distinct provenance data. Domain objects are distributed over multiple systems, e.g. the same invoice can be represented in two systems by two different objects but the provenance system must be able to integrate both identities.

There is no generic provenance framework which can be used as building block for new systems. There exists multiple domain-specific provenance systems that provide similar functionality but do this through redundant implementations. Different application domains need different analysis techniques and tools.

There must be no lock-in. Provenance solutions are coupled with their monitored domain-specific applications. As provenance must be kept a long time an open accessible interface format is mandatory. The provenance itself should be stored in a format that can easily be accessed and utilized. The interface must also be open for changes within the sensor implementation. New

technologies and techniques lead to new sensors that must be able to interface the storage system utilizing the same legacy sensor interface.

Data Integrity must be preserved. Provenance data is sensible data. It must be protected from malicious users as the original data. This is amplified through the multiple system approach: especially when provenance is distributed throughout different companies there must be means of policing which data is exchanged.

Performance and Storage overhead must not impact the domain system. Provenance data is larger than the original document's data. This problematique is amplified as provenance data needs to be stored for a long time period. To solve this various techniques for data deduplication and compaction can be employed.

While this eases the storage problem it adds computational complexity. The provenance system must not impact the normal domain application execution through its performance needs or associated latencies. A new provenance system must find the balance those two problems.

Additional problems arise if the provenance solution is used in conjunction with an application developed in an agile manner:

Unobtrusive Provenance Gathering is mandatory. Agile developed software is open to rapid and frequent change and any provenance solution must not harm the domain application's code ability for rapid re-factoring. This implies that provenance should be gathered in an automatic and minimal intrusive manner so that application developers can focus on the application domain's problems.

The provenance system should be simple and easy to adapt. Agile systems focus on simplicity and the provenance system should mirror this behaviour. This allows software developers to easily extend or augment the provenance solution to fulfill their needs.

The analysis interface will also be utilized by agile analysis applications. Its interface must be easy to integrate and utilize. If it does not achieve this domain-specific extensions might be directly integrated within the core and taint the provenance system's generic principle.

1.4 Motivation

We believe that there is a need for an integrated provenance solution that is not currently fulfilled by existing frameworks or applications. While the amount and quality of gathered provenance is sufficient existing solutions do not provide the ease-of-use and adaptibility needed for an agile software project.

A generic common provenance base system will reduce development efforts and costs while providing the base for reusing existing provenance knowledge. In addition a clean, simple and agile base system can scope with changing trends within provenance thus increasing the life expectation of developed provenance systems.

1.5 Structure of this Thesis

Chapter 2 shows existing techniques and solutions that implement provenance gathering and analysis.

Chapter 3 explores recent related work that could be utilized when designing a new provenance system. The scope of the analysis includes provenance gathering, storage and analysis. The describes techniques will be base for the design of our provenance solution.

Chapter 4 introduces the generic architecture our provenance solution.

Chapter 5 explains the reference implementation of our generic provenance architecture. It starts by describing underlying technologies upon which our solution was built and then shows architectural details.

Chapter 6 provides some evaluation of our prior claims. We verify if our reference implementation is able to gather the needed provenance automatically and external analysis applications are provided with easy to use access to the refined provenance data. We are baseing our evaluation upon a usecase within the experiment management domain.

Chapter 7 concludes the thesis and gives an outlook upon possible future work.

CHAPTER 2

State of the Art

This chapter will show various techniques currently employed for provenance gathering, storage and analysis. We start with completely homegrown self-written solutions, then explore provenance added through plugins to existing system and finally show some complete provenance solutions.

2.1 Homegrown Provenance Systems

Provenance capabilities are needed by most applications. As there is no integrated read-for-use framework available, most of this functionality is implemented by developers themsevles on a per-application base.

This approach yields various drawbacks:

- inefficient use of developer's time

  Each application gets a specific provenance system grafted upon which needs to be designed and developed by a software developer. This leads to duplicate and redundantly implemented capabilities.

- non-standardized solutions

  As each provenance system is developed for one application, developers cannot share their experience between different provenance solutions. This can be a problem if developers change between projects.

- provenance applications not separated from domain applications

  Another problem lies in the missing separation between domain and provenance applications. Their data is mixed in an commonly shared database. This prevents separate release circles of the provenance and domain application, e.g. prevents developers to focus their work on one of both of them without taking the impact upon the other into account.

2.2 Provenance as Addition to existing Systems/Island Provenance

Software developers do not like spending time upon reimplementing the same things. As home-grown provenance solutions bear a high resemblance common characteristics have been moved into the underlying frameworks or within plugins.

An example of the former would be Ruby on Rails' handling of special "magic" object fields. If an object contains a date field named "created_at" or "updated_at" its fields are filled automatically with this information by the framework. This is a very small subset of information provided by a provenance system.

Plugins can capture provenance information for specific purposes. For example under Ruby on Rails the "acts_as_paranoid"[46] plugin record the destruction time of objects which allows later reasoning about why the object was removed. The "acts_as_versioned"[31] plugin stores revision history for specific objects. This allows data-centric provenance capture, i.e. what has happended when to an object, but not how, who or why did this change happen.

Those plug-in based provenance approaches share common problems:

- they still mix provenance and domain data. This is rooted in the shared database which is used by both of them. In contrast to home-grown solutions they allow decoupled updates of the plugin.
- they are "island" solutions. While "acts_as_paranoid" allows provenance gathering for one object type changes between different objects can only be related through their temporal dependencies and not because of their direct causality.

  Also provenance is not shared between different applications. Service-oriented architectures are often composed of different services located on different systems. With plugin-based provenance systems correlating provenance events across different systems can only be achieved manually by comparing the distinct databases.

2.2.1 Comparison of homegrown and framework-provided Provenance

To show the benefits of our system we contrast it against a system with customarily written provenance.

A software developer wants to create a provenance-enabled application for experiment execution and management. The application's development process passes through the following phases:

1. Design
2. Implementation
3. Execution/Usage

We will differate a custom-build solution with a system utilizing our provenance framework for each phase.

Design

With a custom-build application the software developer has to design the data model with provenance in mind. This includes relationship tables for associating data with the executed experiments, capturing runtime information and tracing data operations. This implies that domain-specific experiment management and provenance aspects get intermingled: the software developer cannot treat provenance as an aspect. This slows software evolution: domain-specific improvements must always reflect upon their impact upon provenance; provenance-related improvements cannot be achieved without touching domain-specific code.

The provenance system's reusability is also reduced through the tight coupling with the domain system: changes to one provenance system need to be ported to all other domain systems with embedded provenance components. In contrast a dedicated provenance framework allows the application developer to focus upon the domain-specific design issues. Provenance is handled as an aspect: it is gathered and stored automatically, refined information is provided for later analysis.

The usage of provenance framework also decouples provenance gathering and storage from the domain-specific design. Future improvements within the provenance system can be introduced by

updating just its components without altering the domain application. This allows parallel development of domain and provenance aspects.

Implementation

With the custom system all instrumentation must be written by hand: all data manipulation operations need to be monitored. This is mostly done through embedding monitoring code within the domain code, thus cluttering up the domain-specific logic. The detected changes are then persisted for later usage. The advantage is that provenance can be captured at the desired granularity and can easily be customized for specific needs, but there are also various drawbacks:

- the instrumentation has to be redone for each domain application
- the captured data's quality and granularity depends upon the instrumentation's quality and is likely to fluctuate
- software designers need to integrate provenance gathering into all domain operations. This should not be their concern.

Agile systems also focus upon software verification through software testing. As the provenance gathering is implemented per domain application each implementation needs to be verified on its own thus introducing additional overhead.

In contrast when utilizing our system the application developer just integrates the provenance gathering sensor. Its scope and granularity can then be configured by the developer. It provides provenance with constant quality and can be reused through various domain applications. As analysis engineers can assume provenance data of a basic quality in a known structure they can design their analysis applications to fit, thus allowing reuse of those with alternate domain systems.

Execution

After design and implementation is finished the domain application is executed. As long as the custom provenance gathering was implemented carefully the quality of its gathered provenance is on the same level as with our system.

Maintenance costs for the custom solution are higher. As the provenance gathering has been tailored for one domain application advances and bug fixes in one provenance system need to be adapted for all other provenance systems. The information engineer utilizing provenance has no guarantee that all custom provenance solutions provide data with the same quality nor granularity.

In contrast our system encapsulates generic provenance gathering into an exchangeable plugin. After plugin's improvement all plugin installations can be updated without any alterations. This is achieved through the separation of domain-specific from generic functionality within our sensors. Provenance engineers are provided with one common interface to the provenance pool. Through this they are able to reuse analysis applications with multiple domains. Assumptions about quality and granularity of provenance can be made as it is collected through the same sensors throughout the system.

2.3 Full Provenance Frameworks

There is strong ongoing research on integrating scientific workflow systems with provenance information gathering[52] and analysis[7]. Scientific work always included provenance-gathering: the traditional laboratory notepad is not able to cope with the increased information amount of today's computerized science systems.

2.3.1 Document Management Systems

The goals of provenance gathering systems overlap with those of traditional document management systems: preservation of one document's state and history. One example for Document Management Systems (DMS) would be Microsoft Sharepoint.

It is not possible to substitute a provenance system with a Document Management System as there are various differences in their scope:

- provenance systems focus on objects not documents. This allows for finer audit trail granularity.
- document management systems focus, as the name implies, upon documents. The data and its changes is the first-level citizen, important data as user or process information is neglected. Especially the process information is of high importance for further analysis.
- DMS are written for end-users while provenance systems are rather utilized in the backend area. While this is not a fundamental difference it leads to different goals for designing user interface experiences.
- Provenance solutions are integration-centric. Traditional document management systems presume that all document access and changes happen within their systems.

Provenance systems also allow decoupling of document from provenance storage. This allows choosing or adapting the best storage solution while preserving an existing provenance solution. As provenance exists orthogonal to storage provenance can be gathered across various document management systems.

2.3.2 Chimera

Chimera[52] is used for analysis of data objects and their derivations in collaborative workflows. Its scope is generation of derived data, called virtual data, auditing and data comparison.

It is process-oriented and depends upon manually entered user's descriptions of workflows. Through this meta-language a graph which describes data changes, the so called derivation graph, is generated. Those graphs are used as model for data changes during execution. One execution (which can be parametrized) generates new data which is called a derivation. Provenance is stored through derivation invocations: a derivation graph with added runtime information. The graph can also be enriched by annotations but those do not directly influence provenance decisions.

The provenance information is also used to reduce stored data: data is divided into original and provenance data, the latter can be regenerated through execution of the derivation graph and need not be stored. Also the system depends upon the integrity of users: they must not provide malicious provenance data that could corrupt the whole provenance pool. Overall the user describes his computational work in a meta-language from which in turn the derivation graphs are built. This is suitable for specialized work loads but fails to provide a generic solution for process provenance gathering and management.

The drawback of this solution lies within the dependence upon user input. A malicious user can seriously impact the provenance sphere. The custom meta-language also imposes an learning overhead upon the software developer.

2.3.3 myGrid

myGrid[41] is a middleware solution for usage with bio-informatics experiments. In addition to service discovery and workflow enactment it provides meta-data and provenance management.

It is service-oriented, workflows are implemented in the XScrufl language and executed through an custom execution engine. The engine records various provenance related information during workflow execution. In addition, the end user is required to annotate workflows and services with semantic provenance information, but it should be noted that not all of those information is used automatically by the provenance system, some are only required to describe the experiment for other humans.

In comparison to Chimera this system is more dynamic as generated provenance information can trigger other workflow runs. The provenance store is implemented using a relational database, but all data access is done through RDF. This adds overhead to the processing as the provenance system must convert queries between the relational and graph-based systems.

myGrid depends upon manual user interaction for gathering provenance information as did Chimera. A generic provenance system should be able to perform at least basic provenance operations without this user interactions. This requirement is even stronger if malicious users are to be expected. It also utilizes a meta-language for workflow composition. In contrast to Chimera data gathering is performed upon workflow service level: this yields less but more useful provenance data.

2.3.4 Trio

Trio[57] is not a comprehensive provenance system but focuses on tracking view data in data warehouses. Its approach is adaptable for various other data centric uses.

Database views are transformed into query trees. In those trees the leaf nodes are database tables, parent nodes represent the data manipulations that are done throughout the query. When such a query tree is executed the evaluation starts at the leafs and propagates up to the single top of the tree where the end result is produced. Trio uses the inverted tree to provide provenance information of query results.

Trio is a purely non-annotation based provenance scheme. This makes it very robust against maliciously entered data: it is not possible to corrupt the provenance data as the gathering is integrated within the storage engine itself. Linage information is stored in a special database table and can be queried by SQL statements. The developers feel this to be inadequate and are working on TriSQL which is a specialized language for linage queries.

Trio does not depend upon user generated provenance data but through its focus on only the data level it looses valuable information. In an ideal system Trio would be used to gather the low-level provenance baseline which then would be enriched by additional sensor data. In our proposed system Trio would be perfectly suited as a data-centric capturing sensor.

CHAPTER 3

Related Work

This chapter describes contemporary work that has influence upon our system's design. To create a common starting ground it introduces a conceptional provenance model which terminology is used further on.

3.1 Conceptional Provenance System

A domain system can either be based upon a provenance framework or augment existing systems with provenance-enabled components. The integrated sensors capture provenance data and forward it to the storage and analysis system. The persisted information in turn can be accessed through an analysis interface. While the graphs show discrete analysis applications they can also be part of existing components and use the gathered provenance information for self-tuning.

Domain applications can utilize provenance-enhanced components (Figure 3.1) for gathering provenance data. This can either happen directly within the corresponding component or achieved by wrapping existing components into a provenance gathering proxy. All communication to and fro the wrapped component can be analyzed for provenance. The inner workings of the component still stay hidden from capturing, component-internal data alterations cannot be detected by the provenance wrapper.

Examples for this technique would be Provenance Aware Condor[50] that wraps workflow's jobs into a provenance wrapper or PrIMe[44] that wraps the actor to gain data about its interaction with a system.

Domain applications can also be built directly within provenance-aware systems (Figure 3.2). Such systems can either provide a dedicated interface for provenance related tasks or capture provenance through augmenting already existing interfaces.

A dedicated provenance framework allows custom directives for provenance data gathering. This improves the captured data but imposes overhead upon software developers. An example of this approach would be Chimera[52] where data transformations are recorded in a custom language and then translated into the data manipulation statements but are also used to collect provenance information.

Transparently augmented existing interfaces do not enforce any application changes for provenance gathering. All sensor integration is done within the execution engine itself. This has been done for commercial workflow engines[8].

Application sensors can be supplied by users. Automatic capturing cannot gather all semantic provenance information. Additional Application sensors enrich the already gathered data by user-provided data. Their information must be subject to security and policy checking.

The Storage System stores provenance fragments gathered from different sources. Different sensors can capture different aspects of the same workflow, the storage system needs to integrate them. As the audit trails are very space intensive deduplication and compression might need to be implemented. In distributed systems the storage system also imposes security policies upon collected data.

It provides low and high level interfaces to analysis applications. Through the unified storage it decouples data gathering from data analysis.

Analysis Applications utilize refined provenance data. Domain-specific analysis methods should be implementable through external applications as this keeps the provenance system core generic.

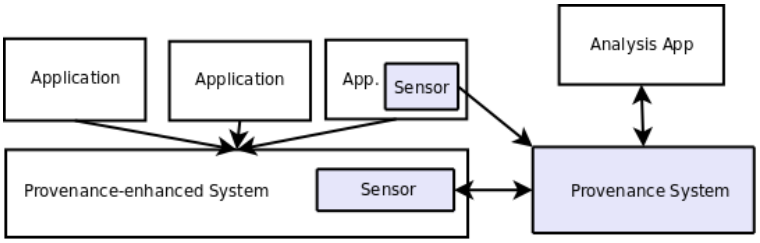

Figure 3.1: Domain Applications utilizing provenance-aware components

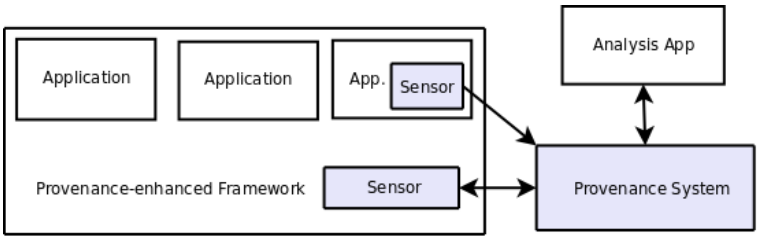

Figure 3.2: Domain Applications build upon a provenance-enabled Framework

3.2 Provenance Gathering

This section will investigate solutions and technologies relevant to provenance data gathering. A sensor is a piece of software that provides provenance data. Different techniques for achieving this are discussed in this chapter.

The basic concept of a sensor as something capturing provenance data is quite simple. Real sensor implementations and their generated provenance data differ in many aspects. The following sections detail those differences and show various characteristics of provenance data.

3.2.1 Scope of Generated Provenance Information

Provenance data differs in their monitoring scope[23]. The two big families are data- and workflow-oriented provenance sensors with user-supplied application-oriented sensors forming a distinct third family.

Data-centric sensors capture provenance by monitoring the domain data pool. They catch the alter operations directly. As they capture the changes on a direct data level they achieve a good coverage as long as all data sources are monitored, but it is captured data is hard to use as its low-level nature does not posses much context information that is needed to detect co-related operations.

Process-centric sensors capture provenance on a workflow level. They mostly augment execution engines[1] to trace domain processes. Through their focus on workflow level they capture higher-level data than data-centric sensors which is also easier to co-relate as the execution engine possesses important context information. They do not experience data changes directly, e.g. they only monitor the data access from the execution engine's point of view. If the underlying technology alters the data this is not recorded, e.g. a database trigger that performs some input data correction or automatic database character set conversions go by undetected.

User-provided application-centric sensors differ from the former mentioned. They introduce provenance data on behalf of a reporting human actor to the provenance data continuum. This is needed for various data that cannot be easily be generated by automatic means.

This sensor family can further be divided through their implementation. Either they work automatically by capturing system states, preprocessing and forwarding them to the provenance system or they can depend fully on direct user input. System that utilize user-provided annotations fall into this category. While they provide semantically rich data the possibility of erogenous or malicious data being inserted by corrupt sensors arises.

3.2.2 Capturing Mechanism

Sensors differ in the ways they are employed for gathering provenance data[23]. The following distinct approaches can be identified:

Automated monitoring sensors trace the execution of domain-specific code to gather provenance data. They do not depend upon user input for data gathering but might be parametrized by the domain application's developer. They do have the advantage of a high resistance against malicious users as those cannot alter the gathered provenance information directly.

They do impose an overhead for the software developer as he has to integrate the automatic sensors with the domain application and also impose an runtime performance overhead through the gathering process but do not impose this upon the final end users. The gathered provenance is called observed provenance.

Manual sensors depend upon user input for provenance gathering. The quality of provenance data depends upon the user-supplied information: if users enter good information the captured provenance data is semantically rich and useful, if not the data can be very lacking.

This mechanism has security problems in face of malicious users as their data is taken to be correct and forwarded to the provenance system. The provenance data is sometimes called disclosed provenance.

Constructing sensors are workflow systems themselves. They accept a workflow definition from users and create a workflow based upon this definition. The new workflow process is augmented with provenance capturing aspects. The captured provenance data is of high-level and differs from the

[1] or are execution engines themselves

other captured provenance data in an very important aspect: it is perspective. It describes how the provenance and data should come look after workflow execution, this information can be utilized to verify captured provenance data against a workflow's expected provenance output.

3.2.3 Granularity of captured Provenance Data

Provenance data is situated between the following two extremes:

fine-grained provenance information is collected on a work-flow level, the sensors are integrated into the workflow execution engine. The gathered information is of a high-level and its amount rather small.

coarse-grained provenance information is collected on a database level. While this cannot answer high-level questions about the workflow that generated data is can give detailed answer about data derivations happening because of database level transformations. The amount of data captured is rather high [29].

The problems of fine-grained provenance can be seen in "Issues in Automatic Provenance Gathering" [10]. The authors propose a very low-level provenance gathering system actually positioned at system or kernel level. This lead to massive amounts of collected data which turned out too fine-grained for useful further analysis. Various techniques are discussed to minimize the needed storage, policed pruning (removal of unneeded provenance information) is deemed to be the best approach.

Workflow provenance focuses on capturing high-level data. Through augmenting workflow systems the captured information is coarse but rich: reasoning about why a data transformation has taken place is easy, but the provenance is not direct experienced (thus could be invalidated by underlying processes).

Users mostly want coarse-grained provenance while automatic systems generate fine-grained provenance information. This mismatch can be overcome by integrating the fine-grained data in coarse-grained provenance information. Versioned objects are a means of achieving this abstraction: multiple data changes are combined into one transaction that generates a new object version.

3.2.4 Temporal differences

Provenance can be differated through its temporal and causal relationship to the executed workflow [23].

Retrospective provenance is gathered during workflow execution and includes runtime information. This information is essential for logging and reasoning about data changes after they happened.

Perspective provenance describes a workflow before it is executed. It contains information about which activities will be executed and how data should be altered. Prospective provenance does not depend upon retrospective provenance while the later should have resemblance with the former. Workflow-based constructing systems are the only ones that are able to capture perspective provenance information.

The following table shows various combinations of the different sensor aspects that yield good candidates for practical provenance sensors:

3.2.5 Integration of Sensors with Existing Systems

A provenance system must always be integrated with a domain system. The domain system must be augmented with functionality that provides the means of provenance gathering. There are various ways of achieving this integration, they differ in the quality and quantity of gathered provenance data as well as in the overhead that they impose upon software development and runtime execution.

There are various approaches to achieve this:

Table 3.1: Sensor aspect combinations

mechanism	scope	granularity	pattern
monitoring	data	fine	observed
monitoring	workflow	coarse	observed
manual	user	any	disclosed
constructing	workflow	medium	disclosed
			constructing

1. using a dedicated provenance framework for writing the domain specific application. This reaps the best quality provenance but imposes a large overhead for the application developer
2. wrapping existing components into a provenance proxy. This allows to capture input/output to and fro a component with small overhead but does not produce comprehensive provenance information. Data manipulation that happens within the component but does not lead to immediate communication with an external instance is not detected.
3. augmenting an existing framework while keeping the same API. This allows finer data granularity than the last approach while keeping the application overhead small. Its overhead and data quality is a compromise of the other two approaches.

All three approaches might harm the developer's productivity and thus prevent easy provenance system adoption. A new system should minimize those costs while gathering provenance information at a reasonable level.

The integration problematic is increased through the multi-system nature of provenance systems. One provenance system captures data from various sources which differ in their provided data quality, scope and security assumptions that can be made. An additional problem are objects that are distributed throughout systems, e.g. the same invoice can be represented in two systems by two different documents but the provenance system should be able to match both.

Of course the usability of an integration technique depends upon the concrete use case. For example, a wrapped data store can be used by an domain application. While we are not able to capture internals the captured data is enough to provide a low-level source of data changes.

3.2.6 Automatic Capturing of Provenance Data

Issues in automatic Provenance Collection[10] states that contemporary provenance gathering systems either depend upon manual or automatic provenance gathering. Manual provenance data entries can also be divided into two families: user-provided data and developer provided data.

The former depends upon direct user input for capturing provenance. Users can be the most powerful source of information: they do know the semantics and reasons behind their workflow. This also implies that malicious users can supply invalid provenance data to destroy the provenance trail. Invalid provenance data can also be introduced through user errors. The quality of the gathered data depends upon the quality of the user's input: a lazy user can impact the whole provenance pool. Automatic verification of provenance is not possible.

Developer-supplied provenance tries to solve some of those problems by removing the direct user interaction. During development time means for provenance gathering are integrated into the domain application by software developers. This cannot provide as good provenance as good direct user input but provides a constant base quality level that cannot be invalidated by erogenous or malicious user input. This approach does impose overhead upon software development and the final provenance data

can still be corrupted by errors during provenance capturing.

Fully automated provenance sensors do neither depend upon user or developer interactions. They achieve this through deep integration with the underlying software framework. The development of the initial provenance system is demanding in terms of complexity but subsequent deployments are easy as the same sensor can be reused.

Automatic gathering through data-centric Sensors

Wang-Chiew Tan focuses [56] on data provenance and gives an overview of current approaches to data-based provenance gathering.

There seem to be two major approaches: annotation based and non-annotation based. Non-annotation based solutions capture the database's state before and after the execution of an transformation, through comparing those states the provenance information is gained. Annotation approaches behave differently: each datum can have annotations attached to it. When a transformation is performed upon the original datum the annotation is also transformed and allows reasoning of the newly datum's lineage. A problem with annotation systems is their target: most systems only allow annotating attribute values, tuples or relationships are not handled.

3.3 Provenance Storage

The main tasks of the storage system are data integration, deduplication and preprocessing. The refined provenance data is then made available to analysis applications for further usage.

3.3.1 Provenance Data

Provenance data describes how data objects came into being, it is used to clarify the actual state and linage of data artifacts.

Provenance data is graph data. The nodes in the graph represent actors, actions and artifacts, the edges describe the casual relationship between them. The graph is as immutable as the history of domain objects that it captures. The only destructive operations that should take place are removal of duplicate or derive-able provenance elements.

Provenance is used to clarify the current state and history of domain objects represented as artefacts within the provenance space. Important aspects include:

- lineage, e.g. who generated or changed an artifact
- the provenance's structure is important, e.g. for detecting common workflows
- changes in structure: how does the workflow change upon changed input artifacts

Each edge or node can have meta-data attached to it through annotations. In contrast to the graph itself those annotations might be changed or be user-supplied. In contrast to ordinary storage systems provenance data is mostly write-once/read-many. Once provenance data entered the system it is immutable (history should not change). Annotations can be added, but are seldom overwritten or deleted.

There are different opinions regarding the cyclicity of the graph. Within one relationship type the causal relationships between elements must be acyclic [2]. The graph consists of sub-graphs of various relationship types, when examined the total graph there can be acyclic relationships.

[2] e.g. one subsequent element cannot be the ancestor of it's ancestor

3.3.2 Provenance Data Structure

As already mentioned lineage data forms a conceptual graph[27]. While the conceptional graph might be used by the provenance system it can be transformed into another form to better suite storage. This only concerns itself with provenance data, the original artifact's data needs to be stored through separate means.

Data can be structured according to different paradigms. The following sections introduce the most important approaches with their corresponding query techniques and compare their suitability for provenance storage.

Relational Databases break down data into tuples and statically defined relationships between those. The standard access method is SQL and most software developers are fluent in this language. Alas there is no standardized support for graph queries (which is the natural expression form of provenance) thus requiring the developer to resolve graph queries programatically which is both design as well as execution resource intensive.

XML is another well known technology. Its representation is based upon a hierarchical tree which does not map to well into a provenance graph which can have multiple roots. Those multiple roots must be reduced to one if provenance should be modulated through XML. Another problem are automatic references between XML documents. This leads to similar complications as SQL: more work and complexity for the software developer.

Graph-based triple stores split data into triples consisting of subject, predicate and object. Their sum produces a (hopefully) adjacent graph. This situation mirrors provenance's Gestalt perfectly but the query language (SPARQL) does not treat paths through graphs as first-class citizens thus limiting their usefulness. While the model is good suited for storage it's usefulness as representation for user-access is thus limited.

Semi-structured Datastores represent data as structured records with relationships between records (which must be resolved manually). While this format is very suitable for network protocols and easy on users but is not as usable for automatic processing. Still it might be perfect for presenting analytical results to users, especially through the network.

3.3.3 Open Provenance Model

During the international Provenance and Annotation Workshop of 2006[29] a session on provenance standardization was held. To better understand the differences between provenance systems the First Provenance Challenge was initiated. Designed as non-competitive the expressiveness and capabilities of existing solutions were tested, alas the problem definition was too blurry so no comprehensive result was gained. One year after a second challenge was issued which clarified questions and expected results. This challenge was also focused on interoperability between provenance systems. As a result an object model was developed and published. Thus the Open Provenance Model[43] was born in late 2007. Its intend is data exchange between systems but not a definition of an object model used within implementations. In 2009 the Third Open Provenance Challenge will take place which will investigate the various Open Provenance exchange models proposed by the different contestants.

Provenance of objects is represented by an annotated acyclic graph enriched with annotations capturing execution data. Base objects are artifacts which describe an immutable piece of state and processes as actions resulting in new artifacts. Processes are started on behalf of an contextual entity called an actor.

Provenance Operations

The Open Provenance Model[43] identified various common provenance operations that must be expressed through a provenance system. Those operations are semantically equivalent to edges in the Open Provenance Model. A simple example of a provenance graph can be found at Figure 3.3, a real

Table 3.2: Open Provenance Model's Operations

Subject	Operation	Predicate
artefact	used-by	process
artefact	generated-by	process
process	controlled-by	user
process	triggered-by	process
artefact	derived-from	artefact

work graph gathered through monitoring an experiment execution can be seen in Figure 6.4.

The Open Provenance Model defines the following operations:

used-by

One or more artifacts were used by a process, i.e. are used as input parameters to an action in our execution engine.

This information is less useful as first thought: an action might consume much incoming data when producing only one tiny bit of outgoing data. In contrast to the "derived-from" edge we cannot create the casual relationship between incoming and outgoing data but can only state that an action had some interaction with some data element.

generated-by

A process generated one or more artifacts. This only captures the direct producer of an artifact (as they are created through actions they must have exactly one). It is not possible to identify ancestor artifacts through this relationship type alone.

controlled-by

Actions are always executed on behalf of an actor. This information is essential for creating audit trails of artifact's history.

While the execution engine is able to directly state who was the user responsible for an action our data-based sensors might have problems deducing this. In a typical application there is exactly one database user with whom all database operations are executed. The provenance module must employ context building techniques to match the user identities of the different systems.

triggered-by

A process was triggered by completion or request of another process. This should also correlate between different systems, e.g. an data store action in the action engine should trigger the store-action in the storage system sensor.

derived-from

This states that a datum was involved during creation of another datum. This implies:

1. there's exactly one action that performed the data transformations
2. the altering operation must know each input and the generated output datum, thus it should have "generated-by" and "used-by" relationships to the two nodes of the "derived-from" relationship.
3. in a provenance graph each artifact must take part in a provenance trail, i.e. must be connected to a process that generates it.

OPM-inspired data storage

While the Open Provenance model was primary designed as a data exchange format there are no function problems that would prevent its usability as storage model. We believe that through early design-

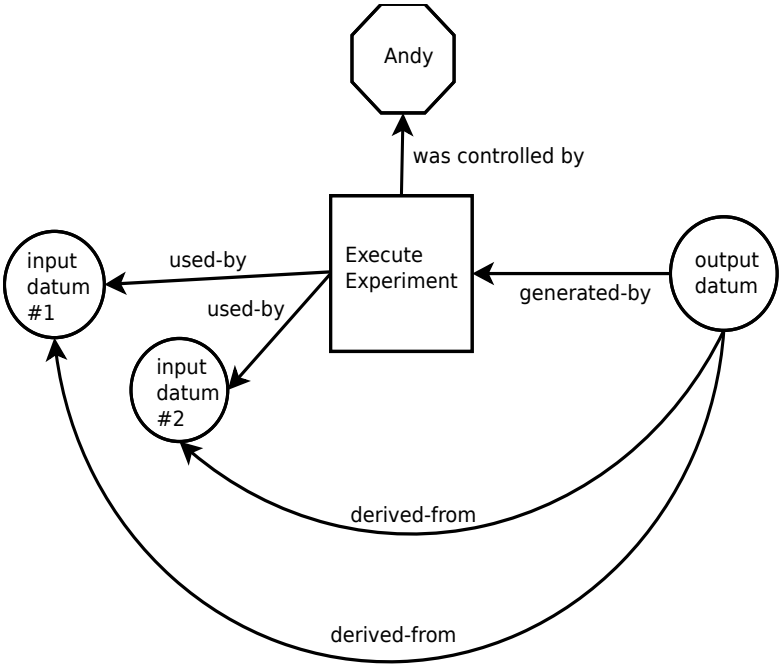

Figure 3.3: Example Provenance Graph of an Experiment Execution

decisions a OPM-based data backend should yield sufficient performance while maintaining complexity at a low level.

3.3.4 Efficient Provenance Storage

Gathered provenance information outgrows the original data by multitudes. An example given by Adriana P. Chapman [13] rates the provenance gathered through a short-time experiment at approximately 6 Gigabyte when performed upon a 200 Megabyte data set.

There are several approaches to improve storage requirements: sensors can be configured for a limited capturing scope, this reduces the amount of stored data but also removes potential provenance data. Deduplication ca be employed to reduce data redundancy but this introduces additional computing overhead for the storage system. The chosen storage backend technique also alters the performance characteristics and non-functional aspects as software and hardware selection can improve the situation.

Central vs. Decentralized Data Storage

Captured provenance data has historical as well as real-time value. As it is captured by distributed sensors it is inherently location-based. Comprehensive provenance information can only be extracted when those distributed provenance fragments are integrated into one graph. In a system where transportation between sensor and storage is lacking this leads to location-aware peer-to-peer storage with all known problems as consistent naming and distributed querying [36].

Depending upon one central storage component removes much of this complexity but introduces a severe trade-off: provenance data can only be integrated after a sensor was connected to the storage system. To improve this situation various techniques as transactions or sensor-side caches can be em-

ployed. A centralized storage system also allows concentration of analysis components. This removes duplicate functionality from sensor implementations.

Deduplication

Another way of reducing the overhead of provenance storage is deduplication which reduces the amount of data that is actually stored but might introduce additional overhead for analysis.

"Efficient Provenance Storage"[13] focuses on provenance deduplication when using XML as data store. Various techniques are introduced:

provenance inheritance describes the removal of one artefact's provenance information if it is just the subset of another artefact's provenance. In case of related artefacts this is commonly the case, use cases that consist of frequent updates upon the same artefacts can thus be reduced greatly.

provenance factorization is based upon the foundation that data manipulations can be represented as trees. Common trees or sub-trees can then be replaced (as done with provenance inheritance) to save storage. Actually this forms families of related actions that aids in further analysis.

structural inheritance is even more advanced. Provenance graphs can be dependent upon just some elements of an data object. If those dependencies can be detected those redundant graphs can be reduced into one.

The discussed techniques are based upon a XML-based storage system but can still be applied upon other storage technologies. In our case we will be using a graph-based storage as base for internal operations. This leads to reduced complexity for implementing *provenance inheritance* as provenance artefacts are inherently broken up into small triples before being distinctly stored. This automatically removes duplicate data.

Provenance factorization would also reap massive performance and storage gains but is not as easily implemented.

3.3.5 Security

Provenance deals with data and its changes. Not only the data artifacts are of importance but also the identity of collaborators[45]. Intelligence agencies hoard their informants, the identity of review authors might need to be authenticated but the contributions of each individual reviewer should stay secret. This situation, in which provenance is more important than the original artifact, is more common than initially thought of.

The security of the whole system consists of the security of the original data and the security of its attached provenance data[11]. There is manifold related work about securing traditional data. Provenance data cannot be treated as traditional data: it does not focus on singular data items but on causal relationships between items. Known security models for relational or tree-structured (XML) data model do not apply.

Another difference lies in its implicit generation. Provenance is mostly created as a side-effect while traditional data is created on direct behalf of an user. It's not clear how this should influence the chosen security model but the need for a fine-grained ACL system for newly created items can easily be seen.

Provenance is a directed acyclic causality graph and one query touches nodes along one path down that graph. The graph itself is immutable, there might be volatile annotations attached to it. A common need is to hide the participation of one user (node) from the graph. There is no comprehensive work available on the implications of this thread model or on security models that deal with directed acyclic graphs.

3.4 Provenance Access

This section investigates former work that relates to the analysis components and communication between them.

3.4.1 Provenance Access and Query Interface

Query languages can be adapted for accessing provenance data. While any generic query language can be utilized their usability might be lacking. While queries might be express-able their creation and handling can be cumbersome. This harms agile application development.

Provenance data is meta-data. It differs from other forms of meta-data because it is based on relationships between objects [27] as ancestry and identity information. Those relationships create a large provenance graph upon which analysis will be performed, thus graph-based query facilities are of high importance. The First Provenance Challenge confirms this: seven out of nine queries involved paths [29].

A useful query language should incorporate at least the following features:

regular expressions upon paths and it's elements (nodes/edges) are needed. Provenance queries describe multiple connected paths through the graph. Without an expressive query facility the formulation of these paths get out of hand.

path pattern matching it must be possible to define a path through pattern matching. The whole query path might not be known a priori, e.g. I might be interested in nodes where outgoing edges eventually touch another node but do not really know or care which edges have been traversed in between. Such queries are for example not easily doable in SPARQL as each element of a path must be known beforehand [27]. Early research for languages supporting path expressions can be found in [2].

paths as first-class citizens: graphs and paths are not the end result of queries but rather intermediate results upon which further analysis will be performed upon [27]. A query language that treads graphs as first-level citizens allows operations upon them, e.g. paths can be compared or set-level operations performed upon them.

aggregate functions and sub queries are not mandatory in a strict sense. Instead of discrete elements aggregate values are often the real goal of provenance queries. An ideal query language should support and aid those queries. The alternative would lead to increased round trips between storage system and analysis application: results of queries would be summarized or used to form subsequent dependent queries. Neither of which is an successful approach to a performant or efficient system.

There are various generic purpose languages that might be suited for provenance analysis as well as a couple of specialized languages, sequent sections will introduce some of them.

SQL

SQL performs upon relational data models which are the antithesis to graph models. To achieve graph-level queries the required query must be mentally transformed into its relational representation and then executed. The performance of larger queries is thus lacking (as they depend upon slow operations as joins).

Lately new revisions of the SQL-standard have seen wider use (SQL99). Those allow limited building of logical tree models through relational data: this can be utilized to provide a better emulation of a graph structure but as was already stated a provenance graph has distinctive features[3] which prevents

[3] i.e. no requirement that there is only one root element

perfect transformation into a tree.

Even when modeling the data structure as tree its relationships are not first-order citizens. Paths through graphs cannot be used as elemental data types: regular expressions upon them or using them as input for further analysis is not possible. A common use case is the comparison of two paths, as they are not first-level citizens this is not possible with build-in operators.

XML

XPath [14] and XQuery [9] are the two most common languages for querying XML documents. XML documents follow tree semantics (only one root) which does not mix well with provenance (which is a true graph). Which is a pity because paths otherwise are full first-class citizens in XML. Another drawback is that the result of a XQuery is always the final leaf node. Alas provenance queries often depend upon intermediate results (i.e. the graph nodes that a path traverses through).

Various provenance solutions tried to emulate full graph semantics through XML trees. Their conclusion was that it is possible but "it's unpleasant to use". Another approach was to use XML as a container format, i.e. using XML to store information for a fragment and utilize embedded XPath/XQuery expressions for relationships between those records. This prevents automatic processing of provenance data: software developers must repeatedly retrieve XML documents to solve their analytical problems.

SPARQL

As mentioned before graph-based triple stores are a good match for provenance models. The commonly used query language use in conjunction with RDF is SPARQL [49].

SPARQL lacks various important features: there is no support for subqueries, few aggregation functions and expressions are not support in select clauses. Even more problemous is that paths are not treated as first-level citizens, thus paths cannot easily be compared. Through the missing subqueries it is also not possible to query for paths where not all edges are known (or at least their count).

There exist various vendor specific extensions that add some of the missing features but they are incompatible with each others. A combination of various vendor specific extensions might fulfill all requirements but that combination would bind a potential software developer to a very narrow query and storage engine. Realistically speaking as it is not possible to mix extensions the "perfect mix" will never happen.

Custom Provenance Query Languages

All prior mentioned languages were general purpose query languages that were not designed especially for provenance queries. Their usability is reduced by missing features or mismatching assumptions about data structure. The results of the first Open Provenance Challenge support this: parties that based their backend upon XML or SQL did have to work around their short-comings or started to develop their own query interface. Recently dedicated provenance query languages as Lorel [2], PQL or the Provenance Query Protocol [40] (QPP) have been proposed.

QPP is a WSDL service definition for provenance query interfaces. As such it only defines the expected interface format and does not specify a full query language. As provenance queries can produce vast amounts of data querying and scoping capabilities are of high importance for QPP. A new provenance system might introduce a QPP network interface for analysis applications but so far no comprehensive query engine which could be employed has been presented.

PQL is a custom query language based upon the Lorel query language. It extends Lorel by more powerful query capabilities but maintains the base feature-set. Lorel was designed for usage with semi-

structured data. It satisfies all requirements for a provenance query language, especially it supports path expressions. In contrast to SPARQL paths can be constructed without prior knowledge of the detailed path structure. Lorel and PQL engines would suit provenance perfectly but alas currently there are no generic query engine implementations available.

While they provide all needed functionality those languages suffer from little publicity: software developers are not as fit with them as they are with established languages like SQL, XPath or SPARQL. There are also few existing query engines for provenance languages which are also lacking thorough evaluation. This implies that a provenance solution that utilizes a custom provenance query engine needs to provide additional more traditional query capabilities.

3.4.2 Interoperability

Businesses need provenance solutions that work through a long time: while their software solutions might change frequently the provenance solution might only change if the existing provenance data can be integrated into the new software.

This might be utilized by provenance solution vendors to tie their customers to their solutions but we do believe that such tactics are short-sighted and will not turn out successful in the long turn.

Data Structure

The Open Provenance Model[43] provides an intermediate representation of provenance information. It's graph based nature is good suited for provenance but leads to transformation costs for provenance solutions that choose a tree or relational based data backend. When choosing a graph-based backend the underlying data model should be modeled as close to the OPM as possible: this would ease the transformation between them.

Another benefit of staying near the OPM lies in better understandability. It is well documented and personell in the provenance area are commonly educated in it. As long as a provenance solution chooses an adapted version of it for data storage people will at least have a slight glimpse of the semantic meaning of the involved nodes, edges and relationships.

The OPM is a quite recent development but there are already algorithms emerging that are based upon its model. Those can be easily adapted to a system which model is a traditional subclass of OPM's model. This leads to long-term usability of the provenance system's backend model.

Provenance Data Access API

The communication protocol between the provenance system's components as well as between analytics users of the provenance protocol should depend upon well-known protocols. This will eases the integration with existing solutions and system designers can utilize already existing knowledge. Web services are the perfect solution to this problem.

Roy Thomas Fielding proposed the REST architecture for network access on resources in his dissertation[21]. The author was involved in the development of the HTTP protocol so it is not surprising that it utilizes the HTTP protocol for denouncing its operations. This leads to very natural data access pattern and lean solutions.

CHAPTER 4

The Agile Provenance System

We have created an architecture for a provenance system that should solve most stated problems. It focuses upon the following key problems:

- Multi-sensor Architecture

  Provenance must be captured throughout domain applications. This leads to an inherently multi-sensor design where one important provenance system's task is the integration of provenance fragments gained from different sensors.

  The interface between provenance system and sensors must be generic so different types of sensors can be connected. Within our reference implementation we focus upon unobtrusive automated provenance gathering but the interface is utilizable by sensors using other approaches. This will also aid adoption existing to our framework.

- Lightweight Sensors

  The multi-sensor paradigm implies that many different sensors are utilized for provenance gathering. Their used techniques and technologies cannot be determined by the provenance system so it must provide a generic sensor interface. In addition it needs to prevent duplicate redundant sensor implementations as this would not confirm to an agile approach. This leads to a core system concentrating shared functionality and multiple light-weight sensors.

- Modular Storage Engine

  Research has shown (Section 3.3.2) that there are various approaches for storing provenance data. From a software engineering point-of-view this implies that the storage engine shoudl be cleanly be separated from the provenance system.

- Generic Reusable Interfaces

  The provenance system connects multiple (different) sensors with one modular storage engine. It is the glue layer between sensors, storage and analysis engines. Clear and generic interfaces decouple these components and aids their parallel evolution.

- Suitable Interface for Analysis Applications

  Provenance can be stored within different structures through various stroage engines. Analysis applications need a stable interface to the provenance system that also presents the stored provenance information in a suitable way. This implies that the provenance system needs to transform data transmitted between storage and sensors as well as between storage and analysis applications.

  while internal access to the stored provenance might change with the used technology existing analysis applications need an stable interface. This unified interface should also present the

provenance information in a way suitable for analysis applications. The provenance system needs to transform provenance data gathered from the storage system into this representation.

- easy to use and modify
  The provenance system's architecture must be easy to comprehend and modify. We cannot forsee all future uses for this system so future developments are aided.

4.1 System Architecture

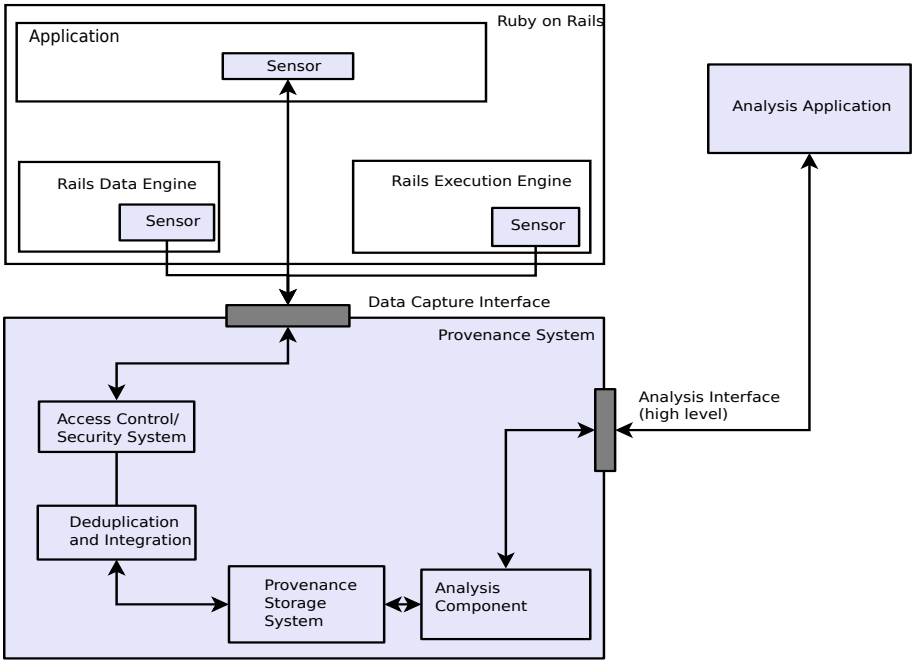

Figure 4.1: Conceptional System Architecture

The scope of a provenance system includes provenance gathering, storage and querying (for analysis purposes). Our resulting architecture can be seen in Figure 4.1. The functionality has been distributed in a way to aid decoupling. To detail the various parts and their components:

Data Gathering is responsible for providing provenance to the storage and analysis components. While we developed an advanced automated capturing sensor the architecture does not concern itself with the concrete implementation but only with their interactions. As the figure implies there aree various approaches to provenance gathering which all shoudl be supported by the storage system interface.

Provenance System concerns itself with data integration and storage. Various responsibilities have been moved from the sensors to the storage system as this allows a simplified sensor design.

As we are dealing with multiple sensors incoming data fragments need to be verified by a security and policy system as malicous users might try to corrupt the provenance system's data.

The provenance system is also responsible for integrating the different provenance fragments into one graph. The refined provenance graph is then exported to the potential external analsysis applications.

Provenance Analysis Components utilize the provenance data to gain valuable information. The provenance interface is a standardized way of gaining the needed knowledge thus hiding the complexities of provenance gathering and storage from analysis applications.

The following sections will delve into the different components.

4.2 Provenance Representation through Artefacts

Domain objects are data objects used by the domain applications. They store the current state of an conceptional domain object. Within the provenance system each change to an domain object is represented through artefacts.

4.2.1 Artefacts

The purpose of a provenance system is to store provenance information for domain objects. This information is represented through artefacts described through:

1. an artefact represents the state of a monitored domain object. For example a domain document has been changed during time: each change spawns one provenance artefact describing the state of the domain document after the change.
2. a domain object can have multiple artefacts associated to it, one for each operation that altered the monitored object
3. an artefact is identified internally by its unique id, for identification of its source a triple (domain-id, domain-class, timestamp) is utilized. The timestamp is added to capture subsequent updates upon the same object.

Changes to one domain object form a family of related artefacts. Such a family describes all alterations leading to the current domain object's state. The changes stored within the artefacts can be utilized to create any domain object's state.

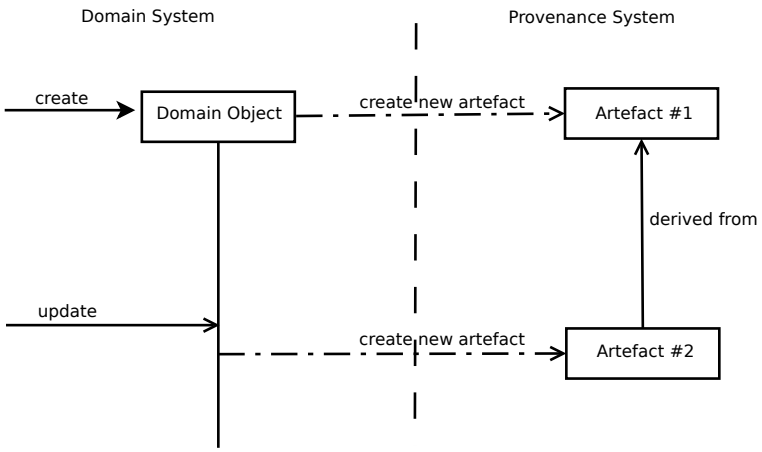

Figure 4.2: Mapping between domain objects and provenance artefacts

When applied to experiment management a recorded experiment would be an example for an domain object. Each experiment alteration leads to a co-related artefact within the provenance data continuum. The first artefact records the experiment objects creation, additional artefacts describe each subsequent change performed upon the experiment record.

4.2.2 Identity of Artefacts

Domain objects are identified by their type and identifier which commonly is a numeric value. As they only represent the current state of an object this is sufficient. Within the provenance domain the type and identifier pair only denotes the family of artefacts belonging to an domain object. To fully identify one artefact an additional identifier is needed which denotes the temporal dependency between artefacts. Any monotonic increasing value is sufficient, the most commonly used identifiers are timestamps or revision numbers.

While domain objects commonly are stateless there also exist versioned object models. With those a sufficient temporal object identifier can be extracted from the domain system. Our architecture can easily be adopted to this scheme but the reference implementation does not use this additional information as it attempts to be as independent of the domain application as possible.

Inter-System Identity Matching

A problem commonly found in distributed systems is that of divergent identities between systems. An artefact found in one system might denote the same entity as an artefact in another system alas their corresponding system-dependent identities do not match. After both have been imported into the provenance system they represent different objects with disjunctive provenance trails. This prevents comprehensive provenance analysis.

As long as Ruby on Rails systems are involved this is not problematic: Rails applications share data through shared databases, this implies that the vast majority of identities will match automatically. Our intra-system versioning mechanism also improves the matching probability through its greedy timestamp-based approach.

If duplicate object identities have already entered the provenance systems inference can be utilized to unify the different identities. Duplicates can be marked with the *owl:sameAs* relationship and the interference engine takes care of the rest: subsequent queries will not differate between the two identities but interpret those two objects as one.

4.2.3 Domain Object Alteration Semantics

Each data-changing event in the monitored application domain translates to newly created artefacts in the provenance system. While each artefact has an internal unique ID they also store enough information[1] to form families of artefacts representing versions of the same domain object. Ordering needs to be established within those families, suitable methods for this are monotonic increasing sub-IDs or time-stamp.

Establishing the families and their ordering within is a time- and resource-intensive task. Instead of performing this for each analytical request it can also be done once per artefact at import-time and its results be stored as a special relationship type. Later operations use this relationship instead of re-creating the families. There is actually not even a negative space impact as we are replacing the already existing *d*erived-from with a more specific *d*irect-ancestor-of relationship, the former can still be generated on the fly by inference so no duplicate information is stored.

Object destruction is an operation that is not captured by typical provenance systems. This can be described through an analogy to the real world: after an object was destroyed its current state does not matter any more and the already captured audit trail survives the now destroyed domain object. The provenance system captures object destruction by the *d*estroyed-by relationships analogous to the *g*enerated-by relationships. While not required for pure provenance the fact that an object ceased to

[1] domain-specific type and identity

be is of importance of various analytical questions. Ironically removal of domain objects within the application leads to creation of additional artefacts in the provenance domain.

4.2.4 Artefact Representation

Artefacts are stored in different representations during their lifetime. Proto-artefacts are generated by sensors and submitted to the provenance system. There they are processed and converted into triple form suitable for graph storage. Analysis applications access the stored information but would prefer a document-based interface. All communication endpoints should also conform to open standards to aid accessibility.

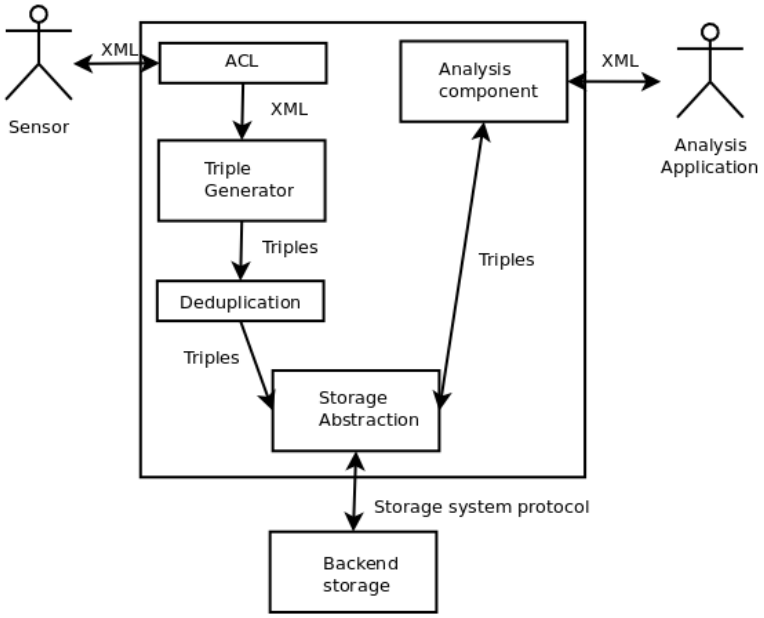

Figure 4.3: Communication between Sensors and Storage System

During **Provenance Gathering** proto-artefacts are stored in an implementation dependent manner. Our reference implementation creates objects which are stored in-memory and then are forwarded to the provenance system. The interface to the provenance system is object-oriented and not a graph-oriented triple interface. This allows the sensor to gather all provenance relevant information and submit it as an aggregated object. If each provenance aspect would be submitted as soon as it was gathered by the sensor the imposed performance impact would higher. In addition a transactional mechanism would have to be implemented: if an domain object has created provenance but is not persisted (i.e. due to an exception) its transmitted provenance must be reverted.

The **Storage System** stores artefacts in graph-conform triple form. As will be shown in later sections this aids integration of different provenance sources.

The stored information can finally be accessed by external analysis and display applications through the **Provenance Access Interface**. As shown before providing direct access to the graph-based triple store is not comfortable external applications as the associated query language is lacking and inconvenient to use. Instead a document-based approach would be preferred. The provenance data is transformed according to a logical model that is implemented through document-based accessible resources.

In contrast to the sensor format the data is not just one atomic provenance fragment but consists of interlinked provenance records that can easily be gathered by external provenance-using applications.

4.3 Data Gathering

Provenance data is generated through domain-specific applications that can be distributed throughout various computers situated within different companies. Provenance sensors must be integrated which each of those systems thus making our system an inherently multi-sensor based. To share functionality, compaction and compression are moved into the storage system - the one place shared by all sensors. This also yields the benefit of making sensors small, simple and easy to implement.

Different sensors posses different internal designs and implementations. While this thesis focuses upon an automatic monitoring sensor the interface between the storage system and sensors has been designed to be sensor implementation-agnostic. This allows easy addition of sensors and increases the life expectancy of the storage system. Future sensor developments can utilize the same provenance interface, thus reducing long-term migration costs.

4.3.1 Handling different Sensor Types

Existing provenance systems often differate between classes of sensors. Our system differs from this practice. A sensor is anything that talks through the standardized provenance sensor interface with our storage system. The storage and analysis is sensor agnostic, it does not differate between sensor data of different origin although administrative restrictions upon incoming data can be enforced through the security component.

This simple concept is very powerful as it allows multiple sensors to connect to our capturing interface. The consistency and integrity of pushed provenance fragments does not suffer as the ACL system for incoming fragments specify which fragment parts are acceptable for each known sensor.

Our prototype's sensor provides access to monitored as well as provided data. User-provided data is gathered through developer-provided callbacks that supply information that cannot be gathered through automatic means. The sensor performs data gathering in a hybrid way: while it is situated at workflow-level it captures provenance data in addition to workflow data. This is possible as we are monitoring data manipulation statements during service execution.

Through focusing on framework level provenance data of fine-enough granularity can be gathered while overhead is reduced. An additional data-centric sensor can still be integrated in the system and would aid the provenance data pool. Such a sensor would be of great benefit: data-centric sensors are the only ones that can capture the "real" data changes and would enable use to verify the captured provenance information.

4.3.2 Multi-Sensor System

The provenance system assumes the existence of multiple sensors. Common functionality was moved from sensors to the common storage and integration system. This concentrates most complex operations as provenance fragment deduplication, inference, storage and retrieval in the central storage system. Sensors themselves only provide few features. Through this separation of functionality the impact of sensors upon the domain system's performance is kept small.

The lightweight sensor design also allows easier separation of functionality between sensors. As the storage system was designed to deal with multi-sensor systems different capturing areas can be separated into different sensors, e.g. a file-system level sensor can be installed alongside a database monitoring sensor. This allows system administrators finer control about the type and granularity of

captured provenance.

This approach also leads to simpler schemes for multi-company provenance gathering. The storage and analysis system inherently deals with multiple sensors, it is agnostic to their location (e.g. how they are located throughout various companies). Administrative filtering is already possible through the ACL component which validates every incoming provenance fragment against a user-supplied rule set.

4.3.3 Provenance Transactions

Forwarding each captured provenance datum individually would introduce overhead, instead we're collecting provenance information and submitting it as batch before completion of each user interaction. The sensor implementation is responsible for collecting and aggregating the corresponding provenance fragments. Different application domains are differently suited for this approach.

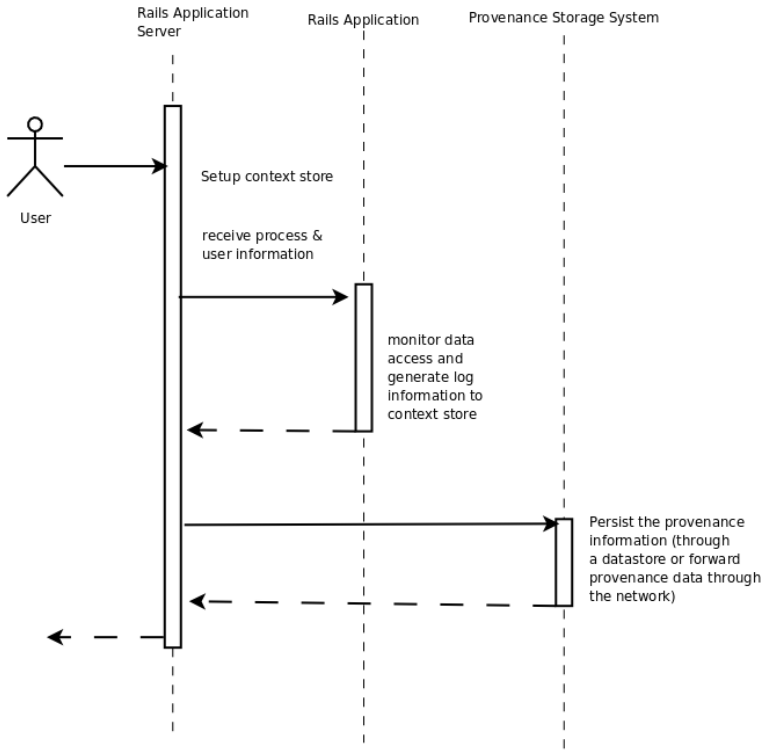

Figure 4.4: Provenance capture flow

To minimize the archived provenance data we only store provenance that lead to an domain object's creation, alteration or destruction. We do not store provenance information for operations that do not alter an domain object, for example we do not store a history of objects that have been read before another read access has taken place. If an application would need this functionality it can easily be implemented but would introduce an high overhead to the capturing and storage process.

4.4 Storage and Integration

The goal of the storage and analysis system is to accept provenance fragments and provide refined data to analysis applications. To achieve this various preprocessing steps are necessary: first the incoming fragments are checked against a user-supplied security policy. This prevents malicious data from entering the system. The fragments are then split up into atomic triples. Those triplets are then integrated into the existing provenance data pool. During this integration deduplication happens: duplicate triplets are removed from the data pool. The analysis interface allows external domain applications to access this refined data pool.

4.4.1 Sensor Interface

The interface between sensors and provenance system is based upon a defined record format. Its logical model can be seen in Figure 4.5, its elements are detailed in Table 4.1. It is more coherent than an provenance fragment based interface where the sensor would instantly forward each provenance fragment towards the storage system.

Access to the provenance resource is atomic: either the provenance information is accepted by the provenance server or not.

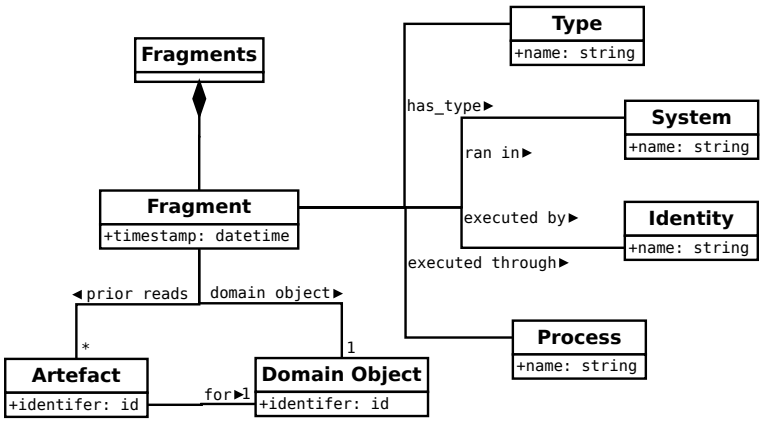

Figure 4.5: Format for Sensor Data

4.4.2 Access Control/Security System

All incoming sensor data is subject to consistency checking through the access control component. This component allows fine-grained policy decisions over which provenance fragments can be accepted. This prevents a malicious sensor from attacking the integrity of the overall provenance data continuum.

While the ACL system allows arbitrary rule sets the common reasons for rejecting data are:

- unknown submitting sensor
- the retrieved data does not comply to the provenance fragment schema, this might imply a transmission error
- A sensor does not deliver expected data: the ACL system can be used to restrict the different kinds of data that a sensor is allowed to transmit. For example a document sensor can be restricted to only support provenance for documents.

Element	Type	Description
type	Enumeration	describes the type of operation, valid types are create, update, destroy
provenance_artefact	Structure	the domain object for which this artefact was created for. It is identified by its type and identifier
timestamp	Time	when did the operation take place?
user	String	the user identifier that will be converted into an identity
process	String	the process identifier that will be used to describe the altering process
system	String	user-supplied identifier that can be used to differate between subsystems
prior_read	Array of Domain Objects	describes which other domain objects were used before the alteration operation took place

Table 4.1: Sensor Interface's Elements

4.4.3 Graph-based Storage Backend

The graph form has various advantages over storing and handling provenance as XML documents. First of all it is provenance's most native Gestalt thus "feels right". Provenance data inherently has a graph structure, if we use another form for storage the software developer will find himself with mentally translating graph queries into their SQL or XPath/XQuery equivalent. Often those query languages are not fit or expressive enough for solving questions, the developer has to work around their lack with custom logic in his application.

The only drawback lies within the XML-to-triple transformation process (and vice verse): this imposes a small performance overhead.

Provenance Integration and Deduplication

Provenance information is distributed over various fragments delivered by different sensors. The received fragments are integrated by the storage system. Subsequent analysis applications access this integrated provenance data pool.

The incoming fragments are split into elemental tuples consisting of subject, predicate and object. As long as objects and subjects delivered by different sensors share the same approach to identity forming their data is integrated into one graph. This approach also achieves initial deduplication as redundant data leads to duplicate tuples that can easily be detected and removed.

The only mandatory part is *p*rovenance_artefact which is used to identify the domain-specific document, all other elements can be omitted. Analysis operations might be dependent upon existence of more provenance fragment parts: for example the alteration trail of an artefact is less usable if no sensor provided information about the executing user. We do believe that it is within the domain of the software developer to provide sensors that retrieve sufficient provenance data for analysis operations.

Another benefit of using the triple form lies in automatic deduplication and integration of the prove-

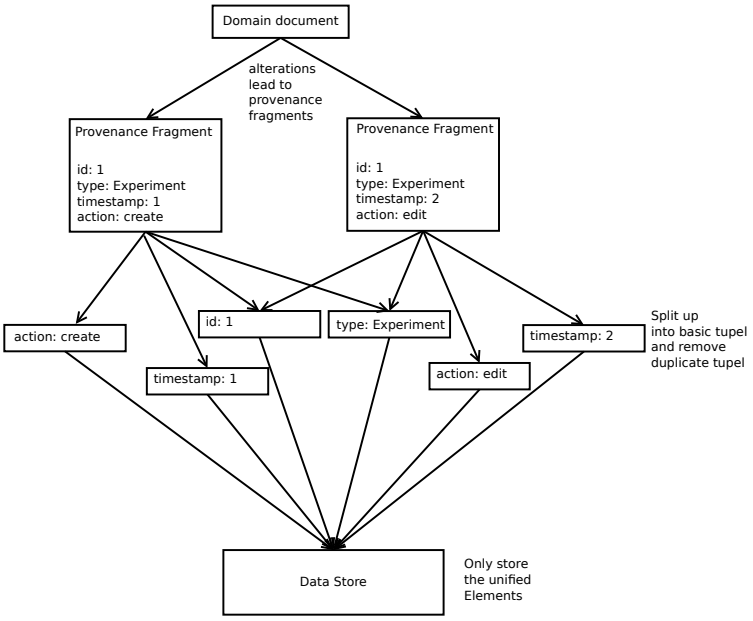

Figure 4.6: Provenance Fragment Deduplication and Integration

nance fragments. As the graph is split up into various atomic triples a very simple deduplication step is automatically achieved by prevent storage of duplicate triples. This is a form of structural compression. Process-related deduplication through sub-classing must still be implemented manually.

Integration is also aided: as soon as artefacts posses the same identifier their corresponding provenance fragments are merged. The only step needed is the unification of different document identifiers which point to the same original object. This is a non-problem in our current work domain as Ruby on Rails uses the back end database table name and identifier for object identification and this is shared across various implementations (as long as they use the same database). If we would integrate different data-sources another preprocessing step would be needed that transforms incoming objects of the same type (but with different identities) into a unified representation[2].

4.4.4 Inference

Graph stores support lazy materialization through inference rules. This allows for sparse graphs that still maintain expressiveness for queries. Inference rules specify which relationships can be dynamically be generated by analyzing existing relationships. The query engine can be parametrized to enable the inference engine on a per-query basis.

Inference allows lazy instantiation of tuples that can be automatically derived from existing relationships[3]. Redundant data can be removed while maintaining the comfort of expanded data for queries.

Enabled inference adds overhead for queries but the burden of developing, maintaining and optimization of the inference data and engine is moved from the provenance to the storage system and does

[2]this can be done analogous to the ACL system (but with an replacement operation instead of just checking regular expressions
[3]The same functionality within XML stores is called “lazy materialization”, its usage with XML is crumbled by patent laws.

not influence design or architecture decisions of the provenance system directly. Also I believe that inference should be closely coupled to storage as this allows further optimization of both.

Usage of Inference

Inference works best when applied upon a data model that posses many redundancies. The Open Provenance Model (and its descendants as my data model) posses few redundancies thus limiting its usefulness. With additional functionality this can change though and inference will get more important.

The storage and analysis system currently utilizes inference for:

duplicate marking and removal As the provenance environment spawns over multiple systems it is possible that the same domain object is represented through different images in those systems and thus artefacts are created for those distinct objects although they should describe the same domain object.

Traditional systems have problems coping with this situation: after duplicates are detected their identities must be unified into one. Subsequent additions and alterations must reflect this transformation i.e. a mapping between domain object and its various representations must be maintained. This can grow very space and time consuming. The identity unification step is also expensive and can impose severe delays when done periodically[4].

Through inference duplicate tuples can just be marked with the "sameAs" relationship and are thus treated as the same object by subsequent queries. This solves the duplicate situation in a simple and clean way, no manual mappings or update operations are prevented.

efficient relationship sub-typing when an artefact was involved in creation of another artefact the system puts both of them in a "derived-from" relationship. Updating an existing artefact follows the same procedure: a new artefact version is created and a "derived-from" relationship formed. Analysis queries often are highly interested in those close relationships, because of this a special relationship "direct-ancestor" was introduced. To allow "normal" queries that depend upon "derived-from" relationships to operate with the new system we would have to store both relationships. This introduces unwanted clutter in our provenance data continuum.

With inference we just specify "direct-ancestor-of" as subtype of "derived-from" and the inference engine does well-known polymorphic substitution thus removing the need for duplicate data storage.

4.5 Providing Provenance Information

Domain-specific analysis applications utilize the provenance data to extract meaningful information. This thesis focuses upon creating the larger framework and generic storage-side analysis components instead of focusing upon one concrete analysis application.

4.5.1 Logical Data Model

Experiments have shown that a graph based data model is well suited for storage but its usefulness is limited for communication purposes. To provide application developers with a better interface to the stored provenance data we introduce an RESTful provenance interface. The underlying data model can be seen in Figure 5.4.

[4] as database tables must be mutually exclusive locked during the unification operation to prevent consistency problems

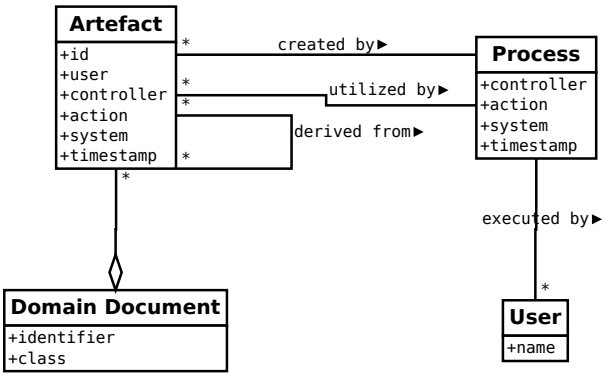

Figure 4.7: Conceptional Analysis Data Model

Changes to domain documents are reflected by artefacts which represent exactly one change in a domain document's state. Each change is performed through a process that was executed within a system on behalf of an user.

This simplified model provides the same data as the graph-based provenance model but incorporates a format that can easily be utilized by analysis applications. The RESTful API can be enriched by additional sub-resources to provide additional analysis capabilities[5] while keeping the existing API stable. An overview of provided resources can be seen at Table 5.1.

External applications utilize the offered resources as an programming interface. The various involved elements and their description can be found in the following section. Concrete examples of those resources can be found in the Experiment chapter.

4.5.2 Providing advanced analysis capabilities

When the incoming requests are converted to graph requests is a good point for implementing advanced query capabilities. Various graph queries do not provide comprehensive results and the gathered data must be refined before it is forwarded to the analysis application. This also decouples the analysis interface from the storage subsystems. This allows great flexibility for additional future query subsystems.

The object-oriented to graph-based translation offers a good point for security subsystem integration.

[5] as was done for process' descendants or artefact's trails

CHAPTER 5

Reference Implementation

Based upon the detailed architecture we created a reference prototype of our provenance system. As an environment for our sensor we have selected Ruby on Rails web applications.

5.1 Environment

Our provenance solution resides within the field of web applications developed and maintained through an agile mechanism. The principles of agile development are stated in the Agile Manifesto[26]:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

This approach often leads to fast-changing domain applications. Software developers work co-located at their customers site and develop the applications with direct customer feedback and fast release cycles. This fast-paced approach leads to dedicated problems for provenance gathering which are shown in section 1.3.

We focus on capturing provenance for applications that are written in web frameworks whose intended usage is this project management technique, e.g. Ruby on Rails[58] or Django. Those frameworks employ dynamic high-level script languages such as Ruby or Python and provide in-framework support for backend data access, especially for interfacing database data.

Larger applications consist of invocation of atomic services. Services are accessed by users through web browsers. Complex workflows are created by an user consuming various discrete services. Each service encapsulates domain-specific knowledge and follows a basic request/response architecture: the requested service accepts its input data, executes and its results are returned to the user. For example a human browses through various dynamically generated web pages: through the selection of viewed sites he implicitly creates a workflow consisting of the actions that are responsible for creating his accessed web sites. The combination of consumed services is highly dynamic and will change over time.

The involved web systems need not to be located geographically near or be owned by the same individuals. Security and Policy Enforcement are of importance, especially when business data between different companies is shared.

The consumed and to be monitored web systems have been developed to fulfill domain-specific needs and not for provenance gathering. Their function must not be impaired by provenance gathering. Also the effort for integrating provenance gathering and analysis must not impact application developer's performance.

5.2 Used Technology

We are all standing upon the shoulders of giants when it comes to writing software systems. Through intelligent choices of underlying techniques and mechanisms work can be reused and the resulting system is easier to comprehend by developers.

5.2.1 Programming Language and Base Framework

We've chosen the Ruby on Rails web framework for both the storage system and as testbed for the initial provenance sensor implementation. It is a rapid growing agile and dynamic framework that is usually used for web 2.0 applications. Typical Rails applications are business-oriented, situated mostly in the project management, communication and productivity sector. Example web-based applications include Twitter, Redmine, Bagback, Basecamp, Campfire.

The selection of this framework does yields various benefits:

Ruby and the Rails Framework are very dynamic environments. It is possible to inject the needed sensor hooks through plug-ins, thus preventing time consuming patches to the framework

Rails provides a database abstraction layer. Through augmenting this capture of all database activity is possible.

Typical Rails installation consist of multiple application servers that do not share anything[1]. A system that can cope with this environment should be easily adaptable to multi-application environments.

There is currently no provenance solution that was adapted for Ruby on Rails.

Those features are commonly shared between agile high-level web-based development frameworks.

All interfaces with sensors, storage and analysis systems are using standard open technologies, through this decision additional systems and frameworks can easily adapted to the provenance system.

Extending the core System

Ruby on Rails supports the "monkey patching" approach[2] for dynamically modified runtime application code. The basic approach is best described by the following quote by Patrick Ewing given at RailsConf 2007:

> Well, I was just totally sold by Adam, the idea being that if it walks like a duck and talks like a duck, it's a duck, right? So if this duck is not giving you the noise that you want, you've got to just punch that duck until it returns what you expect.

Monkey patching allows to augment Rails' functionality with provenance-enhanced versions by replacing the original objects with augmented ones as long as their signatures match.

5.2.2 Transport and Interfaces

Our provenance system is inherently a distributed system: sensors are distributed with the domain system and communicate with the core provenance system through the network. Analysis applications access the refined data through the network even if they might be employed on the same computer

[1] sharing happens through a shared database

[2] often also called "duck punching" or "duck typing"

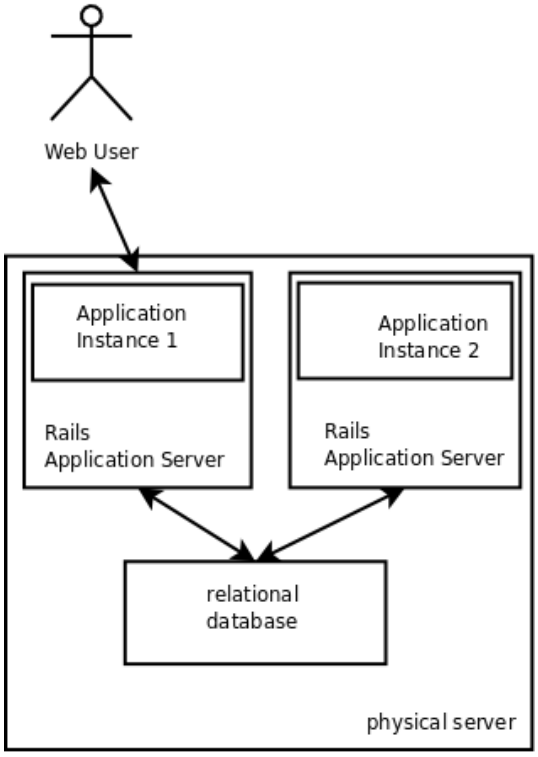

Figure 5.1: Typical Ruby on Rails Stack

system. Internally the storage system is also communicated with through a network protocol to further detangle the involved systems.

Transport Mechanism

We are utilizing the HTTP protocol as base transport means. It is a well known mechanism and its adoption allows us to benefit from existing knowledge and focus on provenance problems. This reduces development costs and eases deployment of our provenance system.

For Authentication and Transport Level Security schemes as SSL/TLS can be reused. They provide both client and server authentication as well as security of transported data. In addition to the provenance system's provided policy mechanisms existing application-level firewalls can be adapted to filter transported data. The firewall exist outside our provenance application's realm thus providing administrators the security that this security cannot be compromised by faults within the provenance system. There exist multiple solutions based on HTTP for performance and fault tolerance. Examples include load-balancers and caching proxy systems. And finally an additinoal benefit of using the HTTP protocol lies in the good client-side library support.

Data Representation

We provide services through the network over HTTP. The services themselves adhere to the REST approach. Each document type is represented through a resource exported through an identifiable HTTP object. Upon those resources common operations as CREATE, READ, etc. can be performed. We do

not support DELETE or UPDATE semantics as once gathered provenance data should not change.

Data within a resource is formatted either through JSON or XML, the client application can chose it through the accessed resource. Data marshaling to and fro XML or JSON is provided by the underlying Ruby on Rails framework.

5.2.3 Sensor

We implemented the sensor as a Ruby on Rails plugin. This allows us to extend the Ruby on Rails framework through a technique called "monkey-patching" without imposing change requirements upon domain applications.

The sensor utilizes Ruby on Rails' ActiveResource framework to access the provenance system. It allows the sensor to transparently perform the Ruby Object to HTTP request mapping. This is performed without prior schema information the structure of the resource is created within the Ruby source code itself.

5.2.4 Back-End

The backend is written in a minimized Ruby in Rails system. We removed most of the database, email and HTML capabilities as we depend solely upon our graph storage engine and XML/JSON formated REST resources. Future Rails versions (post 3.0) will make this easier as the framework is redesigned to be more modular.

We create the SPARQL queries to the storage system without any support library as this enables us to include vendor-specific language extensions.

5.2.5 Storage Technology

As a back-end store we have chosen Franz's AllegraGraph which is a graph-based tuple store. For information why we prefer graph stores over traditional storage solutions please consult the prior chapters.

The reasons for choosing this storage solution were:

- existence of an free edition (which 50 million tuple limit is not a problem for short-term tests with this prototype).
- wide interface options: the web site states that there are various libraries available for Java, Python, Ruby, etc. In addition there are also various HTTP interfaces to the storage system.
- Availability of a graphical query tool that helps with rapid application development

During implementation of the prototype we discovered various problems and shortcomings with Allegra OpenGraph. The ruby bindings (which are implemented as an extension to ActiveRDF) were unstable and did not provide all needed functionality (as data insert). In response to this we moved to using the "new HTTP" interface which provides a simple and efficient interface for insertion and querying through SPARQL. Alas this led to a tuple store that made usage of the graphical query tool impossible due to database format version conflicts.

While this seems rather disappointing at first it has lead to long-term benefits for the storage system. As we just depend upon HTTP commands with SPARQL payloads for communication with the storage system it is rather independent of the back-end store. By usage of a backend-dependent library the storage system would be closely tied to a specific backend solution. Also the network boundary acts as a legal limit with regard to license and copyright concerns.

Storage Interface Within the storage and integration system a small layer abstracts storage system access. The current system assumes a graph-based storage backend but interfaces to alternate backend technologies can also be implemented.

To access the graph store (AllegraGraph) the new HTTP interface was selected. Franz also provides Ruby ActiveRDF bindings for its database system but their stability was unsatisfactory w.r.t. the tests performed. That all communication with the graph system happens through a standardized HTTP protocol also prevents any software license problems[3].

5.3 Provenance Gathering

Ruby on Rails uses a simple process model: there is one process or thread per user request. This allows simple storing of provenance information in a per-thread store which is setup at the beginning of each request and forwarded to the storage system at the end of a request.

This send operation can be done asynchronously to speed up performance but the system currently does this synchronously as this eases development and allows interactions from the provenance system to the application system.

The prototype's data gathering is tightly integrated within the Ruby on Rails execution engine. It works synchronously, i.e. it delays an action's completion until the provenance operation is completed. A failure within the provenance system is propagated back into Ruby on Rails and prevents data storage.

This allows the provenance system to interact with the application domain system. An advanced prototype might utilize this behaviour to prevent fraudulent data from entering the domain system. It also introduces a performance drawback.

Another approach would be an asynchronous system. Provenance data would be passed from the provenance sensor to the storage system without waiting for the later's response. This would improve performance but also endangers provenance data integrity. A reliable storage and transport mechanism introduces most of the overhead of a synchronous system.

A context store is used for provenance information saving. During operation execution provenance related information is stored therein and after the operation is completed its contents forwarded to the provenance storage system. All data operations happening during on service call gets aggregated and sent to the storage system.

5.3.1 Sensor Interface

As we expect various sensors to interact with the storage system a simple and easily accessible interface is important. To achieve this we opted for a RESTful JSON or XML interface transported over HTTP. One reason for choosing this type of interface is good support by programming languages.

The top-level element is fragments which consists of a list of one or more fragments. Sensors can capture multiple provenance fragments (e.g. multiple update operations) and transfer them in batch to improve performance. A fragment describes an altering operation within the application domain. Its contents can be seen in Figure 4.1, a conceptional overview is given in Figure 4.5.

The focus on well-known protocols such as HTTP yields additional benefits. Transport-level security and safety procedures are well known and can be adopted to use with our system. The same goes for redundancy and fault tolerance. Standard HTTP-aware equipment as Proxies, Caches, Firewalls or

[3] that might have happened when including a patent-encumbered software library

Load-Balancers can be used in conjunction with the sensor and analysis interface.

Our first implementation opted for sensors that generated triples that were forwarded to the storage system. This lead to complex sensor implementations especially when dealing with identity detection or deduplication and lead to provenance problems: many tuples had to be transported from the sensor to the storage system, collecting and batch-transferring would require explicit transaction support.

The final RESTful interface solves that problem. The sensor just captures provenance data and forms one XML document per observed object update. This in turn is analyzed by the storage system and then split into multiple tuples.

5.3.2 Generic Pattern for unobtrusive Provenance Gathering

One obstacle that prevents wide-scale provenance adoption is the effort needed on part of the domain software developer. This effort is centered upon integration of provenance gathering means with the domain-specific application. Sensors that generate monitored provenance lift this burden as long as they act unobtrusively with regard to the implementing software developer.

Our sensor implements automated unobtrusive gathering of monitored provenance. This can be achieved without any user or application developer intervention. There are two data pieces that cannot be gathered through automatic means: those are supplied by user-provided callbacks, making the sensor a hybrid monitoring and disclosed provenance sensor.

Architecture of the Automated Sensor Subsystem

The storage system is oblivious to the employed sensor architecture. Any sensor must confirm to the storage-sensor protocol and its supplied data is verified by the security and policy system. This allows flexibility with regard to utilized sensors.

Our sensor integrates with the domain framework and monitors data access through this framework data access system. This allows it to gather data provenance while performing on the higher workflow level.

For each performed domain data manipulation operation a provenance log with all prior data access leading to the operation is recorded. After the monitored service has completed execution but before its result is returned to the consuming client the provenance logs are enriched with process and user information. This final provenance fragment is then forwarded to the storage and analysis application for further inspection and integration with the already gathered provenance information. This leads to a natural transactional communication pattern between sensor and storage system.

Our sensor utilizes much of Ruby's dynamic nature and is able to provide almost all provenance information,small additional information is gathered through software callbacks implemented by the application's software designers. While this is aided Ruby's dynamic nature its fundamental techniques can easily be adapted to other languages.

5.3.3 Augmenting Ruby through Monkey Patching

The plugin utilizes monkey-patching [17;30;48] to enhance Ruby on Rails' Data Access Layer (*ActiveRecord*[5]) and request processing mechanism (*ActionController*[4]) by capturing means. This allows detection of all object data manipulation actions. The resulting log files are forwarded to the Provenance System.

5.3.4 Integration with Ruby on Rails' Data Layer

Ruby on Rails' default data access layer consists of ActiveRecord which is based upon the access technique of the same name[22]. The following object manipulation statements are monitored for provenance:

item instantiation is done for each loaded or queried item. This kind of access is interpreted as read-access upon the loaded object. Logging access on each object's attribute would impose a too high performance hit. Ruby on Rails also supports dirty-marking of changed data: this allows us to detect the changed attributes when they are stored without additional overhead.

item.create is called when a newly artefact is stored for the first time. This action generates most Open Provenance Model relationships with help of the information in the context store.

item.delete notes which user is responsible for the destruction of an artefact through which process. This operation is not foreseen by the Open Provenance Model and is a custom addition by me.

item.update is called when an artefact is updated. It generates almost the same relationships as the *i*tem.create operation, the only difference is the addition of a special relationship (*d*irect-ancestor-of) to specify that the parent and child artefact are closely related.

There are also various relationships that are captured on a per-request base. Those do not change with each data access and thus are only calculated when the context store is initialized:

process name stores a short description of the controller and action that executed the current operations

user cannot be determined automatically. In a typical web application the system user does not reflect the "domain" user that executes an operation. To solve this the developer can fill in a callback that supplies this information.

system this user-definable callback allows a simple human-readable name to better identify involved systems in the provenance graph.

Currently the following Rails operations and queries that bypass Ruby on Rails' Data Access Layer are not supported. Support for the former could be implemented by augmenting the data access layer while comprehensive capture of the later information would require an additional low-level SQL sensor. For example Trio could be utilized to provide low-level provenance data based upon SQL queries for a database.

5.3.5 Ease of Use

One problem of provenance system is their intrusiveness and the associated costs of creating adequate instrumentation in existing solutions. Our System was written with minimal developer overhead in mind.

We implemented the capturing component as Ruby on Rails plugin which can easily be added to an existing Rails application like any other plugin:

Listing 5.1: Plugin Setup Procedure

```
$ git submodule add git://github.com/andreashappe/open-provenance-sensor
    vendor/plugins/provenance
$ git submodule init
$ git submodule update
```

By default the capturing system forwards provenance information to *localhost:4000*. More developer interaction is not needed for enabling provenance gathering. To improve the quality of the provenance information developers can add limited data about user and system information. This can be supplied by augmenting models inheriting from *A*ctiveRecord::Base.

Listing 5.2: Providing additional Information

```
class SomeDataModel < ActiveRecord::Base
  def get_user_information
    "this is the user-id that will be used"
  end

  def get_system_information
    "human readable description of the system"
  end
end
```

The system information is primary for human usage: it helps the analytics user to differate between involved systems. The user information is of high importance, many analytics operations depend upon it and it cannot be gathered automatically (the user seen by the sensor will be the same, mostly *w*ww-data for web based systems and not the user on which behalf an operation was carried out).

5.3.6 Policy and Access Control

Our approach automatically captures provenance information for each persisted domain object. Various reasons exist for non-comprehensive monitoring:

- some sensitive domain data should not be auditioned or shared with the provenance system
- there's just no need for provenance of some objects
- performance considerations prevent provenance gathering for some objects

To suite all needs our system provides two capturing schemes. When using the "eager" scheme all data access objects that are derived from ActiveRecord::Base are monitored for provenance. In an Ruby on Rails application this captures all database access. With the "lazy" scheme no automatic augmenting of data access objects is performed. Application developers need to extend the to be monitored data objects with the capturing module.

Listing 5.3: Different Capturing Modes

```
class EagerCapturedDomainObject < ActiveRecord::Base
  # when the eager-mode is activated every data access
  # originating from descendants for ActiveRecord::Base
  # is captured
end

class LazyCapturedDomainObject < ActiveRecord::Base
  has_provenance

  # with the lazy mode only domain objects that include
  # the has_provenance extension are monitored for provenance
end
```

Domain applications differ through their sensor requirements. For example, most web-based applications primary depend upon databases for file storage while others prefer files for persistence. Which data sources are monitored by the provenance solution must be configurable. We achieve this through the multi-sensor approach. As our system is designed for multiple sensors the different scopes, e.g., database- or file-based capturing, can be extracted into different sensors. The domain application only includes plugins for sources that it is interested in. This also minimizes the impact upon domain applications: as each sensor is designed for exactly one capturing type and technique unneeded functionality is not included into the domain application.

Access control is given through an accept and a block list. Each lists consists of provenance fragments, regular expressions are employed to provide some filtering. This concept is powerful enough for describing all possible occurrences of provenance fragments, even complex ones, e.g.:

Listing 5.4: Example ACL configuration

```
whitelist:
  document_fragment:
  id: ".*"
  class: "document"
  user: "andy"
  system: "rails"
  process: ".*"
capture_all_of_user_linh:
  id: ".*"
  class: ".*"
  user: "linh"
  system: ".*"
  process: ".*"
```

This allows the user "andy" to submit provenance fragments of class "document" through the "rails" system. It would also be possible (through the "process"-hook) to limit the controllers or actions that a user might execute for provenance gathering, this allows very selective provenance gathering. The other rule allows the user linh to capture provenance for everything.

While this is possible it might not be too user-friendly, but this can be solved by a separate application that is used to model and create those access control rules. This has the advantage that the core system can still be kept small and simple while system developers can still be provided with the means of implementing a complex rule set.

Detection and Verification of an sensor's identity is not scope of the provenance system itself. We are relying upon well known technologies for data transportation and those lend well tested means for achieving this. If sensors need to be identified and the transported data secured SSL can be employed. Client-side SSL certificates also provide identity validation while it's transport level encryption provides security of transmitted data. This design decision allows concentration upon provenance problems and prevents redundant implementation of well known techniques.

5.3.7 Sensor Interface Format

Using the tuple format for sensor data transportation leads to various problems. As each tuple creates a single HTTP request transport overhead is high. To solve this we would need to introduce a transaction mechanism to the data capturing interface. This would violate the "stateless" approach of the REST paradigm.

To solve this problem we introduced a structured capture interface for sensor data detailed in Figure 4.5, its elements are described in Table 4.1. This format encapsulates data of various monitored applications into a inherently transactional format. As it captures a request into one atomic message it aids third-party filtering through application-level firewalls. With the prior format a filtering proxy would need to assemble the data of a transaction itself which is an error-prone and expensive operation that is avoided with the later format.

An example of transmitted provenance data can be seen in Figure 5.2. This example shows provenance for a newly created "experiment" with identifier "1". It shows which process was responsible for the new document and on behalf of which user it was executed. The "prior_read" array lists all other

Listing 5.5: Example of a Sensor Provenance Fragment

```
fragments:
  fragment_1:
    type: create
    provenance_artefact:
      id: 1
      class: experiment
    timestamp: Sep 27, 18:14
    user: andy
    process: experiment-create-111
    system: rails
    prior_read:
      artefact_1:
        id: 1
        type: data
      artefact_2:
        id: 4
        type: data
  fragment_2:
    ..
```

Figure 5.2: Example of a Sensor Provenance Fragment

artefacts that were involved in the creation of the new domain document. Multiple fragments can be grouped together and transmitted with one HTTP request.

5.4 Integration and Storage

Ruby on Rails was also used for the integration system. This allowed rapid development of needed functionality. The storage itself was Franz AllegraGraph which will be described in a later section.

Recent developments have shown that we could remove the Rails dependency of the storage system and write a pure Ruby system. While this would remove software dependencies for the storage system itself the benefits have not been enough to warrant this rewrite: the Rails' sensor as well as the analysis system still would depend upon Rails, thus again creating the dependencies that the storage system would have lost.

5.4.1 Extending the core System

Domain-specific extensions need means of extending the core system while staying generic. These standardized interface also keeps extensions modular and enable interchangeability between extensions.

The storage and analysis system was written utilizing Ruby on Rails and inherits its extension mechanisms. Plugins are allowed to change almost all internals of existing code and can provide additional analysis methods and interfaces.

We distributed the functionality of our provenance system to better suit subsequent monkey patching. This allows software developers to replace our default capabilities with specialized functionality by adding plugins to the provenance system's Ruby on Rails system.

5.4.2 Distributed Back-end Storage

Provenance analysis must be performed centrally. The integration steps need atomic access to the full provenance data continuum. While processing needs to be centralized the underlying storage can still be distributed. This yields benefits as improved performance or higher data safety through redundancy.

As the storage and analysis system encapsulates all data access exchange of the underlying data access technology is possible, candidates have been shown in Section 3.3.2. We focus on graph-based storage accessed through a HTTP protocol. This allows usage of existing solutions as web proxies or HTTP load balancers for performance and safety gains.

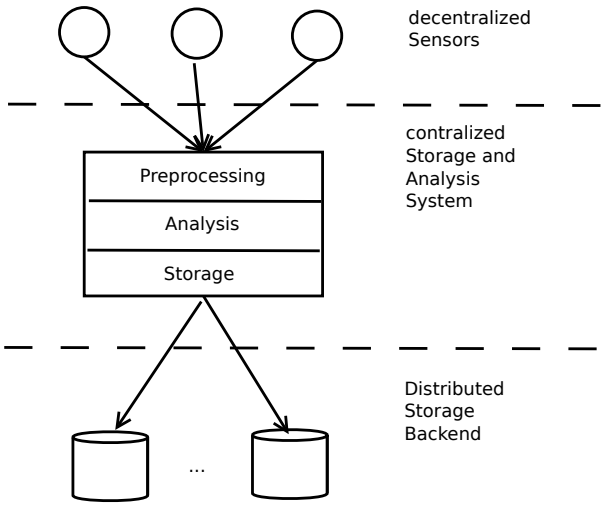

Figure 5.3: Storage System and its relationship to Sensors and Storage

5.5 Provenance Access Interface

This sections describes some details that concern themself with analysis application access to the provenance storage system.

5.5.1 Access Control

The analysis interface allows extensive reasoning about a systems data and the processes altering that data it must be protected from malicious users. As there is currently no standardized security model for graph-based data the security checking must be done on access level and might be later be extended through an intra-data security model.

We do not provide any means of protecting the HTTP interface. In a productive system this can be easily be done through employing accounting reverse proxies (e.g. apache) or application-level firewalls. This methods are well known and tried and can easily extend the provenance system's functionality without adding further complexity to the provenance core.

This is another area where my reliance upon open and well known protocols and standards significantly aids the development of the provenance system by delegating tasks to tried and accepted systems.

5.5.2 Logical Data Model

Experiments have shown that a graph based data model is well suited for storage but its usefulness is limited for communication purposes. To provide application developers with a better interface to the stored provenance data we introduce an RESTful provenance interface. The underlying data model can be seen in Figure 5.4.

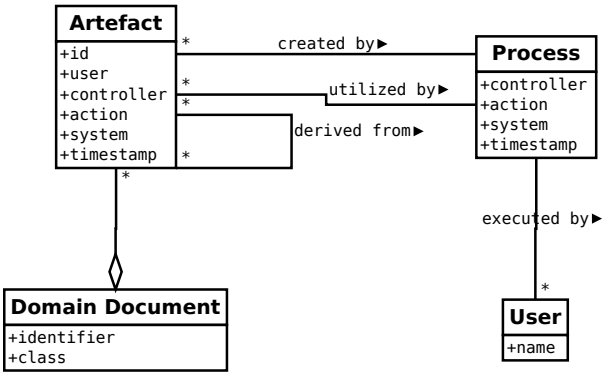

Figure 5.4: Conceptional Analysis Data Model

Path	Description
/artifacts	high-level overview about artefacts
/artifacts/*id*	provenance for one artefact
/artifacts/*id*/trails	provenance family for the domain object
/processes/	high-level overview about processes
/processes/*id*	information about one process
/processes/*id*/descendants	information about processes that are logically dependent upon this process
/identities/	high-level overview about known users
/identities/*id*	provenance information for one user identity

Table 5.1: Analysis Interface Resources

Changes to domain documents are reflected by artefacts which represent exactly one change in a domain document's state. Each change is performed through a process that was executed within a system on behalf of exactly one user.

This simplified model provides the same data as the graph-based provenance model but incorporates a format that can easily be utilized by analysis applications. The RESTful API can be enriched by additional sub-resources to provide additional analysis capabilities[4] while keeping the existing API stable. An overview of provided resources can be seen at Table 5.1.

External applications utilize the offered resources as an programming interface. The various involved elements and their description can be found in the following section. Concrete examples of those resources can be found in the Experiment chapter.

[4] as was done for process' descendants or artefact's trails

Artefacts

Each artefact describes the creation, alteration or destruction of one domain object, its elements are:

Element	Type	Description
artifact-id	numeric identifier	in conjunction with type describes the domain object
artifact-class	String	the type of the domain object
timestamp	Time	when was this artefact created?
generated-by	Structure	describes the process and user that generated this artefact
derived-from	List of Artefacts	shows artefacts that utilized this artefact during their creation
was-derived-from	List of Artefacts	shows each artefact that was involved in the creation of this new artefact
direct-children	List of Artefacts	show direct children: these are new versions of the artefact that describe subsequent updates to the domain object

The retrieved information includes:

which other artefacts were involved its creation: this is shown through the *d*erived-from collection which includes all artefact that were utilized during creation.

direct artefact metadata: this describes the domain object, this includes its *a*rtifact-id, *a*rtifact-class and *t*imestamp

which other artefacts used this artefact: this information is provided by the *w*as-derived-from collection. The record also includes *D*irect-child relationships that show that derived artefacts are direct children, i.e. the child is a new version of the same domain-specific object. This denotes an updated domain object.

generated-by provides information how this change came to be: as shown by *g*enerated-by. This shows which Ruby on Rails controller was responsible for the artefacts creation (*c*ontroller, *a*ction, *s*ystem), the process that was used (*p*rocess) and the user in charge of the operation (*u*ser).

Indirectly referenced provenance is not shown within the fragment to preserve efficiency and must be queried through the referenced object's resources. For example to retrieve process information the process resource ("/processes", see Listing 5.7) can be retrieved. This is especially needed for user information as this is only contained within the detailed process information and not within the artefact.

The following example shows a provenance artefact encoded in XML:

Listing 5.6: Example Provenance Artefact

```
<hash>
  <was-derived-from type="array">
    <was-derived-from>
      <artifact-type>Artifact</artifact-type>
      <subject>artifact_64359</subject>
    </was-derived-from>
  </was-derived-from>
  <artifact-id>"4"</artifact-id>
  <artifact-type>Artifact</artifact-type>
  <generated-by>
    <controller>"documents"</controller>
    <action>"create"</action>
    <user>andy</user>
    <process>
```

```
      documents-create-38600a7934b384aef21ca1825d8f25ffb38629ef
    </process>
    <system>Rails</system>
  </generated-by>
  <derived-from type="array">
    <derived-from>
      <artifact-type>Artifact</artifact-type>
      <subject>artifact_2506</subject>
    </derived-from>
  </derived-from>
  <direct-children type="array">
    <direct-child>
      <artifact>artifact_64359</artifact>
    </direct-child>
  </direct-children>
  <artifact-class>"Document"</artifact-class>
  <timestamp>"Wed Jul 22 08:41:46 UTC 2009"</timestamp>
</hash>
```

Processes

Each artefact must be created through a process that was executed on behalf of an user. The provenance information describing each process is:

Element	Type	Description
process-id	identifier	unique identifier for this process
system	String	user-supplied identifier of the involved system
controlled-by	User Identity	who executed this process
controller	String	describes the rails controller that was used
action	String	describes the action that was executed
used-artifacts	List of Artifacts	which artifacts were read during process execution
deleted-artifacts	List of Artifacts	which artifacts were destroyed during process execution
created-artifacts	List of Artifacts	which artifacts were created during process execution. These describe all created and updated domain objects.

Table 5.2: Process' Elements

This data allows to trace process execution as well as monitor all created, altered or destroyed artefacts. Artefacts are referenced by their provenance store internal identifier and further information can easily be accessed by accessing them through their REST resources.

The following example shows a simple process provenance record:

Listing 5.7: Example Provenance Process

```
<hash>
```

```
  <created-artifacts type="array">
    <created-artifact>
      <artifact-type>Artifact</artifact-type>
      <subject>artifact_34870</subject>
    </created-artifact>
  </created-artifacts>
  <controller>"documents"</controller>
  <artifact-type>Process</artifact-type>
  <process-id>"9cdd5dfb7f8d53b1dc7b0ba168caf797c8ad691c"</process-id>
  <deleted-artifacts type="array"/>
  <action>"create"</action>
  <controlled-by>andy</controlled-by>
  <used-artifacts type="array"/>
  <system>Rails</system>
</hash>
```

Identities

Identities describe user-specific information.

Element	Type	Description
name	String	identifier for this user
processes	List of Processes	recently executed processes that were executed on behalf of this user
generated-artifacts	List of Artifacts	recently generated artefacts

Table 5.3: Identity's Elements

5.5.3 Work-Around SPARQL shortcomings

SPARQL does not support advanced features as sub-queries or path-based expressions. Various queries' efficiency depend upon those features. For example, when searching for all descendants of an artefact (all direct and indirect other artefacts that depend upon the artefact) the program flow resembles Figure 5.5. Actions written in italics are send to the backend storage system, boxed actions interact with the analysis application.

If the query language would support advanced features for defining paths between nodes the program flow would be simplified to the flow pictures in Figure 5.6. This would remove the need for the retrieval loop within the storage and analysis system and would spare round trips thus improving performance. As the analysis component encapsulates data access technology future versions of the storage system can switch to another query engine supporting this functionality without imposing changes upon existing analysis applications.

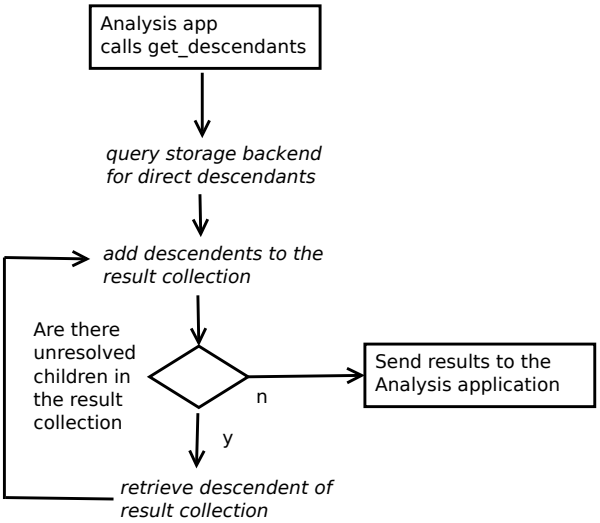

Figure 5.5: Program flow for gathering an action's descendants

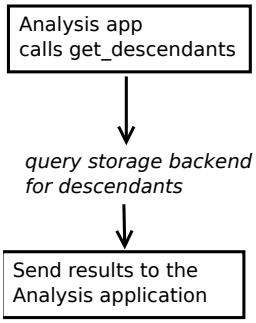

Figure 5.6: Program flow for gathering an action's descendants with path expressions

CHAPTER 6

Evaluation

Scientific method consists of collection of data through observation and experimentation, and the formation and testing of hypothesis[18]. An important part of observation and testing is documentation. It allows subsequent verification and reenactment of executed experiments. Scientists are responsible for creating those documents: this imposes tedious work upon them and its results are subject to fluctuations in quality. Automated systems can overcome this problems.

We introduce an experiment management system that utilizes Genesis2 for experiment testbed deployment and Agile Provenance for history gathering and storage. We show that our software stack avoids time-consuming user interactions for documentation and provenance gathering while providing a rich dataset for later experiment evaluation and verification.

6.1 Experiment Management

Experiments are an important part of scientific workflows as they allow falsification of theses[53]. Their documentation is paramount for presenting scientific claims. Experiment Management Systems (EMS) are used to automate experiment execution and documentation; they concern themselves with the following phases of an experiment's life cycle[38;39]:

experiment design describes the systems and their behaviour during experiment execution

experiment deployment uses the entered testbed configuration to create and deploy the testbed. The test application must be able to interact with the deployed testbed without modifications that would taint the tested behaviour.

data collection must be performed during experiment execution. The data should be preprocessed and presented to the end user in an usable manner.

The origins of experiment management systems were simple log entry tools as scientists' notepads. Automation, esp. computers, has increased the amount of generated data so that integrated and automated experiment management systems become necessary[38].

6.1.1 Responsibilities

Experiment Management Systems remove repetitive tedious tasks from scientists. They also guarantee constant quality of captured experiment data. An EMS is employed during the whole experiment life cycle.

An experiment is created by a scientist by definition of a testbed. The testbed consists of one or more communication endpoints that are consumable by the testbed application. After each endpoint's behaviour was specified by the scientist the EMS transforms the configuration data into concrete services and deploys them as testbed.

After the testbed was deployed the test application can be executed. During its life cycle its communication to and fro the testbed needs to be monitored and captured for later analysis. The sensor mechanism must be designed and employed in a way that does not taint the observed behaviour: otherwise the resulting captured data would not allow reasoning about the original test application.

The captured data must be presented to the scientist in an usable way. In addition the experiment management system should allow third-party applications to access the retrieved capture data. The configuration stored within the EMS must be an image of the current configuration within the real testbed.

Experiments evolve over time. They must change to altered test parameters or are adapted by other scientist to suit their distinct experiments. The EMS needs to store information about all evolutionary steps and must be able to recreate any former state. This can be employed to recreate and verify old experiments and provides the experiment documentation and provenance needed by the scientific method.

6.1.2 Existing Components

We did not write the Experiment Management System from scratch but were able to reuse existing components:

Genesis creates and deploys testbeds for service-oriented architectures[34] . Testbeds are modeled through a scripting language on the front-end host, their services are distributed throughout one or more backend hosts.

The model is not persisted and interactions between the test application and testbed services are neither traced or logged. No provenance information is collected or stored.

Provenance System is the system described within this master thesis. It consists of sensors forwarding provenance fragments to a central integration and storage system. The sensors capture provenance in an automatic and unobtrusive manner, the collected and refined information is provided to external applications through an separate interface.

Management System is used to bridge the gap between domain application (Genesis) and the provenance system. Its task is configuration of testbeds as well as storing experiment execution data. It utilizes Genesis to generate testbeds from stored configuration and the provenance system for gaining historical data.

6.1.3 Contributions to the Provenance System

Supplementary to our Provenance System additional components were needed to integrate Genesis into our solution. Those components are:

Genesis2 plugin for model observation captures any model modification and forwards new testbed models to the management system. This provides monitoring of experiment configuration.

Genesis2 communication monitor captures all communication between the testbed and the tested application. The communication logs are forwarded to the Management System.

Management System allows configuration and deployment of testbeds. In addition it allows analysis of captured testbed logs. Through provenance changes within models or communication patterns can be detected and presented to the user.

The *Genesis2* plugins should enable data capturing without any modifications of existing Genesis2 systems nor code base.

6.2 Architecture

The overall architecture can be seen in Figure 6.1. The Management System bridges Genesis (seen through its front and backend systems) and the Provenance System. Entered testbed configuration is used through Genesis to create and deploy a testbed. Within this testbed a Test Application can be run, all occurring communication is forwarded to the Management System. The Management System's data pool is monitored by the Provenance System, which captures provenance trails for all occurring operations. The user only interacts with the Management System for testbed configuration and log visualization.

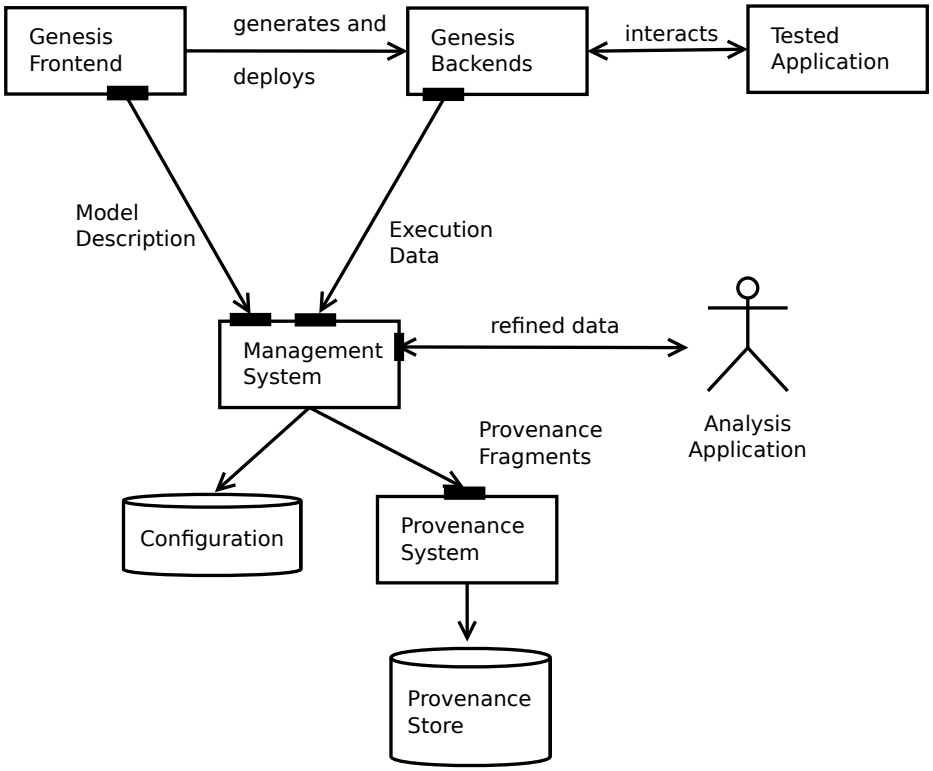

Figure 6.1: System Architecture

6.2.1 Genesis

Genesis consists of one front-end and various back-end systems. Model configuration takes place at the front-end component. This generates service endpoints containing operations which are distributed throughout the backend components. Testbed model configuration is only transient, no information is persisted.

Configuration Language Genesis uses a Groovy[35]-based language for configuration (as can be seen in listing 6.2). Configuration scripts configure hosts, services and operations contained within those services. The concrete implementation of one operation is given directly as Groovy program.

An example genesis configuration script can be seen in Figure 6.2. This short code segment defines a host containing a simple operation *helloworld* that returns the sum of its input parameters. In addition

Listing 6.1: Example Genesis Configuration

```
def h = host.create("testhost",8181)

def sl = webservice.build {
  testservice() {
    onDeploy = { println "deployed" }
    onUndeploy = { println "undeployed" }

    helloworld(a: long, b: double, response: String) {
      "got $a and $b"
    }
  }
}

def s = sl [0]
s.bindTo(h)
s.deploy()
```

Figure 6.2: Example Genesis Configuration

functionality for callbacks (*onDeploy* and *onUndeploy*) is provided. Finally the service is bound and deployed to the backend host *testhost*.

Implementation Genesis is implemented in Java with an embedded Groovy interpreter for system configuration. Functionality is provided through plugins, this allows a very modular system.

6.2.2 Provenance System

The provenance system integrates and stores provenance fragments. Its provenance data is utilized by the Management System for retrieving experiments' and models' histories. This enables retroactive inspection of an element's change history, showing which users altered which aspects of a testbed. As the captured data is also stored by the provenance system changes in the test application's communication patterns can be related to testbed configuration changes.

Data Format Internally data is stored as a graph confirming to the Open Provenance Model[42]. For better accessibility the Provenance System provides an object-oriented REST interface[20;25] to stored data. Data is grouped into the following resources:

artefacts describe one concrete state of an stored object. As an object is changed through time it generates multiple artefacts, their sum describes how an object came into being.

processes which software processes are responsible for artefacts?

users which users are responsible for the execution of software processes?

The "Management System" uses those resources to gain historic data about experiment configuration and its related execution logs.

6.2.3 Management System

The Genesis system does not persist information about the current model after deployment finishes. The provenance system describes changes upon persisted domain objects. To bridge this gap the Management System persists the current configuration and captured experiment data within its database which can be monitored by a provenance gathering sensor.

Data Model

Configuration and trace data is persisted through ActiveRecord[37] objects. The data model includes experiment configuration as well as experiment trace data. The former is gained from Genesis' Groovy experiment definition while the later is gathered during experiment execution.

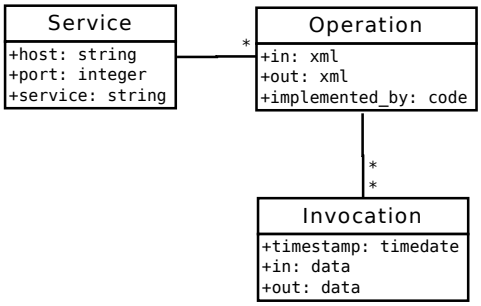

Figure 6.3: Management System's Data Model

The data model (as seen in Figure 6.3) consists of few database tables. A Service uniquely identifies one communication endpoint which can be contacted by the tested application. Each endpoint consists of one or more operations. An operation has an incoming and outgoing signature describing parameters and their format. In addition we store information about the operation's implementation or configuration. During experiment execution an operation can be invoked, in that case we store all messages going from and to the service as well as their timing information.

The data continuum is monitored by a provenance sensor. This allows reflection upon changes within one data object, e.g. we can detect configuration changes and changes within the trace data. Our data model lacks most typical provenance information as timestamps, revision history for stored objects or user operation trails as those can be automatically provided by the provenance system.

User Interface

The Management System also provides a simple end-user interface implemented as a traditional web application. It allows creation and modification of experiments as well as browsing through corresponding capture logs. Examples of the user interface's output will be shown throughout the Evaluation chapter.

6.3 Integrating Genesis

Interactions between GEMs and Genesis consist of three distinct variations: first the testbed configuration entered into the Management System must be converted into a Genesis-compatible form and then utilized by Genesis to deploy a testbed. Second the Genesis-internal testbed model must be monitored for changes. This detects alterations which happened because of direct user interactions as well as mis-transformed data models from the Management System. And last communication data during experiment execution must be captured and persisted to document the experiment. Additions to Genesis should leverage its plugin mechanism to keep the Genesis impact minimal. Each following section will detail on of those aspects:

6.3.1 Testbed Configuration Monitor

The front-end is augmented with a small sensor component which forwards model information to the Management System. The testbed model itself is not stored at the front-end component.

The *ModelWatcherPlugin* installs a Genesis-wide plugin watcher. Each instantiated plugin is observed by an unique *PluginWatcher* instance. A plugin can export various model types, a service consists of various of those instantiated types. The Plugin investigates all data types exported by a plugin for possible instances of the *WebService* type. If found the *PluginWatcher* installs one *WebServiceObserver* which will be called for each model change.

The plugin traverses through all affected hosts and their deployed operations. It creates a snapshot of each current operation and forwards this information to the Management System. This traversal also resolves a discrepancy between Genesis' and the Management System's data model. In the former one operation can be deployed on multiple hosts while the later thinks one operation to reside exactly upon one host. To keep the implementation simple a REST interface is utilized for this.

Implementation-wise our plugin registers a watcher on *ModelChange*-Events. If the actual event is a *DEPLOY* event the plugin investigates all involved web services and extracts their information as signature, implementation, deployed locations etc. While the configure script can employ programming techniques as loops or recursions the deployed operations are always atomic; their definition does not contain external dependencies upon other web services.

Genesis2 creates a configured operation at the configured hostname but places the operation within a canonical path that includes the operation's type and an UUID. The UUID is not configurable but can be detected by our sensor and is transmitted to the management system where it is presented to scientists as read-only data attribute.

6.3.2 Testbed Communication Monitor

Testbed provenance describes how a testbed was configured, deployed and altered through time but does not describe what happened during experiment execution. The tested application's behaviour is defined through messages that it exchanges with the deployed testbed.

We introduce the *CommunicationInterceptor*. It augments each deployed *WebService* with a slim layer that captures all communication to and fro the encapsulated web service. The proxy approach removes the need for direct instrumentation of the tested program, thus removing a possible source of corrupted experiment logs.

The captured messages are encoded for usage within XML messages and forwarded to the management system. There they are analyzed, related to configured endpoints and stored within the database. As the management system keeps the actual testbed configuration incoming messages must always be relate-able to one endpoint.

The management system also performs simple session reassembly: the communication interceptor only delivers atomic messages (i.e., one incoming or one outgoing message). Through message ordering the management system is able to relate outgoing messages to prior causal retrieved messages.

6.4 Providing Provenance

Provenance is an important aspect for scientific workflow systems. It allows automatic documentation thus supporting reproducing, sharing knowledge reuse of or within experiments. It also allows reasoning about unexpected experiment results and preparation of results for publication[12;54]. Integrated

provenance solutions promises to be a chief advantage of scientific workflow solutions over traditional systems [16:47].

Through usage of the existing Provenance System we hope to reduce redundant work for implementing provenance gathering, storage and analysis facilities.

6.4.1 Provenance Capture

Our provenance solution provides sensors that integrate into existing Ruby on Rails [51] applications. We employed those to augment the Management System with provenance capabilities.

The sensor is installed as any ordinary Ruby on Rails plugin [55] and needs minimal configuration, mostly the location of the Provenance System. In addition two callbacks have been provided, one to provide user information and one to identify the Management system through a simple human-readable name. Through techniques described in 5.3.2 alterations to the management application were prevented.

The inclusion of provenance capturing through the plugin-concept minimizes its intrusiveness upon the domain application[1]. The application developer was able to focus upon other domain aspects and treat provenance as an automatically supplied aspect.

6.4.2 Provenance Usage

The Management System needs to access the provenance system for needed information. Provenance is represented through the REST paradigm [20]. Ruby on Rails provides the *ActiveResource* [6] framework for accessing remote resources. It is a thin dynamic layer that hides network interactions and dynamically generates client-side methods for accessing remote resources.

The provenance system forbids direct access to the graph-based persistence store as there currently is no security system in place. Direct SPARQL-based access might be allowed when generic graph-based security systems are available. Complex queries are implemented as resources within the Management System.

Provenance information is provided through an REST interface. The management system utilizes the Ruby *ActiveResource*-interface to automatically generate client-side access code. *ActiveResource* also transparently (de-)serializes incoming and outgoing data between its Ruby and XML representation.

The stored information is represented through resources. Through searching through the *artifacts* resource all revisions of a specific domain object can be found. To retrieve the full history trail can be retrieved by accessing sub-resources of an specific artefact. An example of a Ruby script accessing provenance information for an operation can be found at listing 6.5.

6.5 Evaluation

The evaluation is based upon a storyline detailing a workflow in the software engineering field. Through the use cases benefits of GEMS and its components will be shown. The storyline is broken down into various distinct operations. Each of them will be detailed through a separate section within this chapter.

[1] which is the Management System in this case

6.5.1 Illustrative Example

A software developer has designed a new service-oriented application. To test it and its interactions with other services GEMS is employed. The software developer describes a testbed through the experiment management system. He creates multiple endpoints which contain operations with defined semantics. After the testbed configuration is finished the testbed configuration is transformed into running services and deployed. Throughout the experiment's life cycle the testbed and test application are deployed and verified. Each corresponding testbed change must be documented. This documentation can be consulted to retrieve a high or low level overview of executed test applications and testbeds. Through its various revisions the evolution of th test environment can be observed.

The software developer expects the following capabilities from the EMS system:

1. deployment of configured testbeds
2. automatic capturing of all communication within the testbed as well as capturing of testbed configuration changes. This documents the test case without any user interactions.
3. display of the current testbed's state
4. historical information about stored provenance configuration
5. capture and show all communication to and fro the testbed. This is valuable information and can be seen as input and output to black box testing.
6. it must be possible to revert any prior testbed configuration to replay prior experiments

6.5.2 Experiment Configuration and Deployment

The management system provides testbed configuration through a simple web interface. Its declarative nature allows easy listing and modification of already configured endpoints and operations. Another benefit are the inherently multi-user capabilities: the configuration is stored on the central web server, no additional document management system is needed for the configuration script.

After the user requests a testbed deployment the management system transforms the entered configuration into a standardized Genesis2 Groovy script. The generated scripts possesses the advantage of adhering to consistent structure and style guidelines.

The Genesis2 script is written to a temporary file which is passed on to the Genesis2 Java interpreter. It generates the testbed web services and deploys them to the configured hosts.

6.5.3 Experiment Configuration Capturing

We expect a management system to transparently detect and store any configuration change. It must be agnostic to the change's source: it should detect configured changes as well as ad-hoc changes done during experiment execution. This approach enables various use cases:

- configured testbed models can be verified against their configuration: any derivations should be detectable through a change to the stored testbed model.
- detect changes within the testbed during experiment execution. Testbed configuration changes during testing. The automatic capturing of testbed changes enables automatic logging of changes within the management system.
- transparently document testbeds. As missing endpoints and operations are automatically generated within the management system our solution transparently documents testbeds.

The Genesis2 front-end is augmented with a testbed model sensor. It detects deployed testbed elements and forwards configuration changes to the management system. There, the captured information is split up and integrated with the currently stored testbed model. If deployed endpoints, services or

operations are not already registered the management system creates new configuration entries.

We evaluated this claim by running a simple test: first we configured a small test case within the management system and deployed it. The deployment was monitored by our sensor, but as there were no derivations from the configuration no change was detectable. To test the sensor we exported the *Genesis2* deployment script, altered it by hand and deployed it. Every change was forwarded to the management system and integrated into the testbed's configuration. To investigate an extreme case we deleted all testbed configuration and executed various versions of our testbed scripts. After execution was finished the management system contained the current configuration of the deployed testbed.

After executing the test case we investigate the stored provenance graph in Figure 6.4. While all needed information is captured the large size of the provenance graph shows that queries upon it could be tedious.

6.5.4 Testbed Provenance

The management system only stores an image of the current state of an experiment. We cannot compare this to any former state as we lack that former state's information. The evolution of the testbed and its corresponding observed messages are lost with each newly deployed experiment. Alas this is a very important part of the scientific method. We expect an experiment management system to provide all this information.

Traditionally this functionality is provided manually by the EMS developer. We used a different approach: as part my master thesis I developed an unobtrusive plugable provenance system that automatically captures needed information from the domain application (in our case this is the EMS). We added this capturing plugin and utilized the provenance system's data interface for provenance queries. This removed reduced the costs of implementing provenance capturing and storage completely, the only remaining provenance effort were the domain-specific analysis queries.

The management system utilizes provenance for the following tasks:

- show history of endpoints and operations. Each modifying operation (as *create* or *update*) is logged with its meta-information (timestamp, executing user and process, data dependencies). This allows to distinct between changes entered within the management system frontend and changes done through Genesis2 scripting.
- show detailed information for each object's revision. As the provenance system is written with object systems in mind it is able to provide change history on a per-attribute level. In conjunction with the captured meta-information this generates a thorough "paper trail" of each monitored object.

We were able to retrieve the wanted provenance information in an efficient manner. Examples of querying the provenance system can be seen in 6.5.

6.5.5 Capturing Communication

The experiment management system cannot concern itself only with configuration data but must also monitor all communication within the testbed. In our use case this involves capturing of transported messages, othe user cases can involve ellaborate measurement mechanisms as physical bio-informatics sensors.

The observation must no influence the experiment. Following this guideline we employed sensors that transparently augment *Genesis2*. No direct alteration of *Genesis2* or the to be tested application is needed.

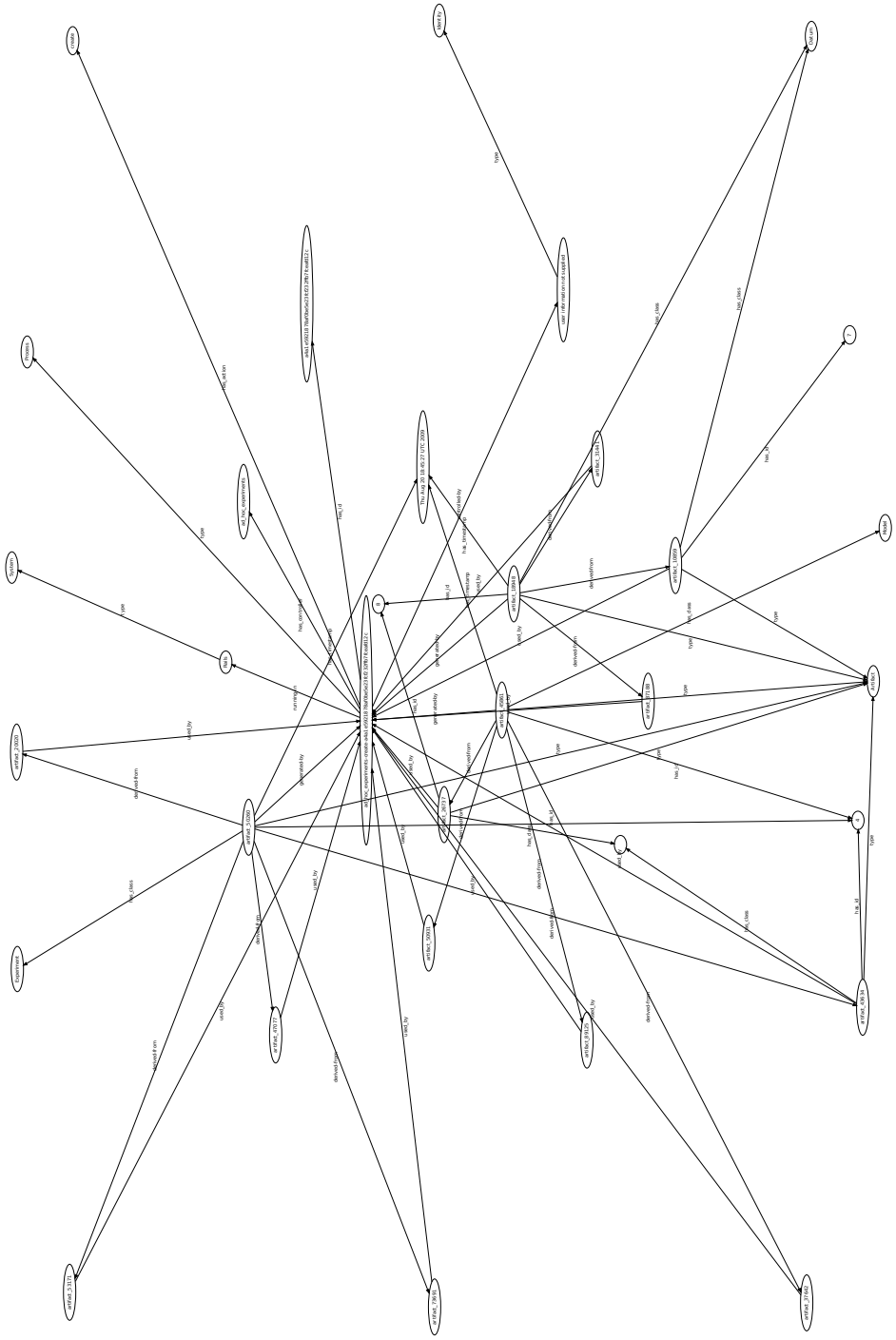

Figure 6.4: Ad-Hoc Experiment Provenance result

Listing 6.2: Example Provenance Access

```
# search for all artefacts for an Object of class Operation with id 2
# an artefact describes one change that happened to one operation.
# this generates /artifacts.xml?type=Operation&id=2
@artifacts = Artifact.find(:all, :params => { :type => Operation,
                                              :id => 2})

# retrieve all trail subresources for the first retrieved artefact. This
# resource provides a summary about all changes done to one operation
# this accesses /artifacts/artifact_123/trails.xml
id = @artifacts.first.id
@trails = Trail.find(:all, :from => "/artifacts/#{id}/trails.xml")

# take the first revision from the @trails collection and show its changes
@revision = Artifact.find(@trails.current_artifact_id).first
@revision.attributes.each do |name, change|
    print "attribute #{name} #{change.before} -> #{change.after}"
end
```

Figure 6.5: Example Provenance Access through *ActiveResource*

To test the capturing plugin we deployed a testbed and executed a simple test application within it. The test application periodically invokes the deployed webservices. The sensor generates communication fragments which are forwarded to the management system where they are associated with configured endpoints. Within the management system's user interface the captured XML communication fragments can be listed and examined. This allows relating endpoints (their configuration) and exchanged messages (i.e. their experiment behaviour). The provenance system adds versioning to the storage: older versions of endpoints and communication fragments can be retrieved for alter examination.

6.5.6 Testbed Configuration Replay

A scientist has worked on an experiment through the last couple of weeks. To locate an error the testbed was specialized to better trap the problem. After it was fixed the scientist wants to evaluate his solution with his prior testbeds.

With a manual provenance solution (e.g. hand-recorded documentation) the scientist would need to retrace all changes to the testbed and recreate the wanted testbed version. Through our automatic provenance solution we can recreate any prior state of the testbed. The scientist can select one artefact (as seen in an object's history trail) or any point in time as reference point. The provenance solution then creates a Groovy Genesis2 script that can be deployed.

To achieve this we access a special resources within the management system; the */archives/date.xml* resources. For each entered date it produces a Groovy representation of the testbed's state at time *date*. To generate this the management system performs the following steps:

1. retrieve a list of all testbed objects from the provenance system
2. get the last valid object version prior to the entered date (this has to be done for each object).
3. verify that all inter-object references are valid. When an object references another object the provenance system must make sure that the referenced object's version is valid during the referencing object's time frame.

4. convert the object collections into valid Genesis2 Groovy scripts. This functionality is shared with “normal” testbed generation and deployment

This use case shows that the provenance system can be utilized to automatically create any former state of an observed object model. While we utilize this ability to redeploy testbeds additional use cases can easily be seen. Generally speaking any software system that needs a versioned or logged state benefits from an automatic provenance solution

6.6 Conclusion

The combined software stack has been shown that automatic unobtrusive experiment documentation is possible. The effort of creating the management system was concentrated upon creating and integrating the management system with Genesis2, adapting the provenance solution did consume less than eight percent of the overall project time and effort.

The provenance solution allowed us to unobtrusively capture the history of all monitored and persisted objects. This provided the base data layer for experiment’s documentation and replay.

Within the provenance system the current graph-based storage engine and its query language (SPARQL) were the biggest problems: due to SPARQL’s inherent limit[28] advanced queries had to be broken down into simpler sub-queries; each of them needs a separate round-trip to the data store thus reducing overall provenance performance. There is ongoing work on specialized provenance query engines (i.e. Lorel[3]) which will solve this drawback in the short term.

CHAPTER 7

Conclusion and Future Work

We have spilt the conclusion in four parts consisting of a general overview of the chosen architecture and an examination of its larger components. Future steps are detailed for each part.

7.1 General Architecture

Our solution uses a centralized storage system for provenance compaction, persistence and analysis (see Section 4.1). This concentrates provenance processing into one central component while simplifying sensor design. The central system was augmented with a generic security subsystem (as can be seen in Section 4.4.2) which is sufficient for multi-sensor architectures.

The architecture and its reference implementation depend upon well-published and open standards as far as possible. Development tools, as protocol or log analyzers, were easily adapted for usage with our system. Production environments gain access to security and distribution tools.

The architecture provides a clear separation between generic and domain-specific functionality. The former is centered at the core while the later can be added through external applications utilizing the analysis interface or through extensions to the core system. This leads to a clean software design.

7.2 Unobtrusive Provenance Gathering

Our sensor design provides unobtrusive provenance gathering through automatic capturing techniques. This approach allows domain applications to focus upon their domain-specific tasks in a provenance-agnostic manner. Experiments have shown (see Section 6.5.3) that the quality and quantity of automated gathered data within the Ruby on Rails framework is sufficient for subsequent provenance analysis.

The captured provenance data is detailed enough for subsequent provenance analysis. Its quality is independant from user activities as it is captured without direct user interactions. This is especially important if users might be considered malicious. Another advantage is the sensor's independence from the domain application implementation: through this the captured provenance data quality is guaranteed accross different domain use cases. The sensor's scope can be configured by the application software developer, this allows exclusion of data that should not be subject to provenance gathering. This standardized format also allows easy re-combinations of storage systems and analysis applications.

Our implementation can be used as base for future sensors. This is aided through the focus upon simplicity and open technologies. Ideas for initial beneficial additions could be:

Integrate direct provenance API into the instrumentation sensor. The gathered provenance data could be presented in a language native format, for example in Ruby the provenance data could be implemented as dynamic object attributes. Currently provenance information is represented through objects separated from the domain's objects.

Add additional sensors to the domain system. We provided a workflow-level database capturing sensor but additional sensors can be added to domain systems in dependence of the business domain's needs. A low-level SQL sensor or a file-level sensor would be examples of different gathering scopes implemented through add-on sensors.

Utilize unit tests to gather prospective provenance. Agile systems commonly depend upon testing frameworks for verification of their codebase. Those unit tests could be utilized to provide a set-actual comparison upon the provenance sphere. This would enable anomaly or fraud detection.

7.3 Storage System

Provenance information is provided through different formats within our solution: internal data handling uses a graph-based form for storage and integration while a transformed object-oriented interface format is provided for interactions between the storage system and external components as sensors or analysis applications. Graph-based storage is well suited for persisting provenance information. Provenance fragments are integrated without user intervention as long as identifiers between different domain-systems can be matched.

Figure 6.4 shows one downside of graph-based provenance storage: the graphs grow large and are hard to navigate. This representation is inefficient for network transport and hard to utilize by external analysis applications.

The provenance graph's security was not within the scope of this thesis. While basic security enforcement was imposed upon incoming and outgoing interfaces the stored data itself is not secured. We hope that recent work on graph-based security will yield storage systems that will provide the needed functionality.

The Evaluation has shown that the current graph-based storage engine and its query language (SPARQL) were the biggest problems: due to SPARQL's inherent limits [28] advanced queries had to be broken down into simpler sub-queries; each of them needs a separate round-trip to the data store thus reducing overall provenance performance.

Future steps should focus upon the storage component and its deduplication and query capabilities. Techniques as common sub-expression elimination that would decrease the amount of stored provenance but might degrade the analysis systems performance. Another approach is putting an compiler optimization engine on top of the gathering subsystem: provenance data optimization and compiler optimization share common characteristics. Inference engines should also be explored further. They provide means of replacing redundancies while keeping the logical data model intact. Inference engines are currently tightly bonded into the underlying storage engine: a generic inference engine designed as part of the provenance system would be needed to keep the core system generic.

The former suggestions contain themselves within the storage system implementation, another weak point is the current query interface. SPARQL is too limited in the long term and needs to be replaced with a more powerful query language. A future step would be the development of a generic query engine (using Lorel [3] or PQL) on top of our storage system.

7.4 Analysis Interface

Our reference imlementation stores provenance in a graph resembling the Open Provenance model. This form mirrors provenance's structure and leads to natural access patterns. In addition we gain automatic deduplication and integration of provenance fragemnts. The graph form is not perfect for providing client-application access to to the provenance pool: it's cumbersome to use, clients need to create complex queries for gaining standard information. We provide a simple networked resource-based interface which allows easy client-side interaction with the analysis data, further query functionality can be implemented conforming to this method.

SPARQL lacks various features (most important sub-queries and variable paths) that had to be emulated by the analysis subsystem. Vendor-specific extensions exist but taint the core's generic principles. Another approach would be the implementation of a dedicated generic stand-alone provenance query language. This has been started by PQL and Lorel but there are currently no query engine implementations available that could be integrated.

Advanced analysis techniques need to be employed over the gathered provenance data. The graph-based nature of the provenance graph leads to the applying of social network techniques.

Bibliography

[1] A provenance project without an Author, f. n.d., 'Applying provenance in organ transplant management'.
URL: *http://www.gridprovenance.org/applications/OTM.html*

[2] Abiteboul, S., Quass, D., Mchugh, J., Widom, J. and Wiener, J. L. 1997*a*, 'The lorel query language for semistructured data'.

[3] Abiteboul, S., Quass, D., McHugh, J., Widom, J. and Wiener, J. L. 1997*b*, 'The lorel query language for semistructured data', *International Journal on Digital Libraries* **1**(1), 68–88.

[4] *ActionController API documentation* n.d., `http://api.rubyonrails.org/classes/ActionController/Base.html`.

[5] *ActiveRecord API documentation* n.d., `http://api.rubyonrails.org/classes/ActiveRecord/Base.html`.

[6] *ActiveResource API documentation* n.d., `http://api.rubyonrails.org/classes/ActiveResource/Base.html`.

[7] Barga, R. S. 2006, Automatic generation of workflow execution provenance, Technical report, Microsoft Research.

[8] Barga, R. S. and Digiampietri, L. A. 2008, 'Automatic capture and efficient storage of e-science experiment provenance', *Concurr. Comput. : Pract. Exper.* **2**0(5), 419–429.

[9] Boag, S., Chamberlin, D. D., Fernández, M. F., Florescu, D., Robie, J. and Siméon, J. 2007, 'Xquery 1.0: An xml query language', World Wide Web Consortium, Recommendation REC-xquery-20070123.

[10] Braun, U., Garfinkel, S., Holland, D. A., Muniswamy-Reddy, K.-K. and Seltzer, M. I. n.d., 'Issues in automatic provenance collection'.

[11] Braun, U., Shinnar, A. and Seltzer, M. n.d., Securing provenance, Technical report, Havard School of Engineering and Applied Sciences.

[12] Buneman, P., Khanna, S. and Tan, W.-c. 2001, Why and where: A characterization of data provenance, *in* 'In ICDT', pp. 316–330.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.1848*

[13] Chapman, A. P., Jagadish, H. and Ramanan, P. n.d., Efficient provenance storage, Technical report, University of Michigan, Wichita State University.

[14] Clark, J. and DeRose, S. J. 1999, 'Xml path language (xpath) version 1.0', World Wide Web Consortium, Recommendation REC-xpath-19991116.

[15] Dang, Y. B., Cheng, P., Luo, L. and Cho, A. 2008, A code provenance management tool for ip-aware software development, *in* 'ICSE Companion '08: Companion of the 30th international conference on Software engineering', ACM, New York, NY, USA, pp. 975–976.

[16] Davidson, S. B. and Freire, J. 2008, Provenance and scientific workflows: challenges and opportunities, *in* 'SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data', ACM, New York, NY, USA, pp. 1345–1350.

[17] Delabar, E. 2008, 'Duck punching javascript - metaprogramming with prototype', `http://www.ericdelabar.com/2008/05/metaprogramming-javascript.html`.

[18] Dictionary, M.-W. O. 2010, 'Definition of scientific method'.
URL: *http://www.merriam-webster.com/dictionary/scientifichod*

[19] Ding, L., Kolari, P., Finin, T., Joshi, A., Peng, Y. and Yesha, Y. 2005, On Homeland Security and the Semantic Web: A Provenance and Trust Aware Inference Framework, *in* 'Proceedings of the AAAI SPring Symposium on AI Technologies for Homeland Security', AAAI Press. (poster paper).

[20] Fielding, R. T. 2000, REST: Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, Irvine.
URL: *http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm*

[21] Fielding, R. T. n.d., 'Architectural styles and the design of network-based software architectures'.

[22] Fowler, M. 2002, *Patterns of Enterprise Application Architecture*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[23] Freire, J., Koop, D., Santos, E. and Silva, C. T. 2008, 'Provenance for computational tasks: A survey', *Computing in Science and Engineering* **1**0(3), 11–21.

[24] Groth, P., Munroe, S., Miles, S. and Moreau, L. 2008, *In Lucio Grandinetti (ed.), HPC and Grids in Action*, IOS Press, chapter Applying the Provenance Data Model to a Bioinformatics Case.
URL: *http://www.ecs.soton.ac.uk/ lavm/papers/hpc08.pdf*

[25] Happe, A., Truong, H. and Dustdar, S. 2010, 'Unobstrusive provenance'.

[26] Highsmith, J. and Fowler, M. 2001, 'The agile manifesto', *Software Development Magazine* **9**(8), 29–30.

[27] Holland, D. A., Braun, U., Maclean, D., Muniswamy-Reddy, K.-K. and Seltzer, M. I. n.d., Choosing a data model and query language for provenance, Technical report, Havard University.

[28] Holland, D. A., Braun, U., Maclean, D., Reddy, K. K. M. and Seltzer, M. 2008, Choosing a Data Model and Query Language for Provenance, *in* 'Second International Provenance and Annotation Workshop (IPAW'08)'.

[29] Holland, D. A., Seltzer, M. I., Braun, U. and Muniswamy-Reddy, K.-K. 2008, 'Passing the provenance challenge', *Concurr. Comput. : Pract. Exper.* **2**0(5), 531–540.

[30] *Interviews from RailsConf 2007 in Portland* 2007, `http://podcast.rubyonrails.org/programs/1/episodes/railsconf-2007`.

[31] Jul, M. n.d., 'acts_as_versioned documentation'.
URL: *http://ar-versioned.rubyforge.org/*

[32] Kifor, T., Varga, L. Z., Vazquez-Salceda, J., Álvarez, S. and Willmott, S. 2006, Ehcr: An eu provenance case study, Technical report, EU Research Project, Contract Number: 511085.

[33] Kifor, T., Varga, L. Z., V?zquez-Salceda, J., ?lvarez, S., Willmott, S., Miles, S. and Moreau, L. 2006, 'Provenance in agent-mediated healthcare systems', *IEEE Intelligent Systems* **2**1(6), 38–46.

[34] L., J., H., T. and S., D. 2008, 'Genesis - a framework for automatic generation and steering of testbeds of complex web services'. 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'08).

[35] Laforge, G. 2008, 'Groovy: An agile dynamic language for the java platform'.
URL: *http://groovy.codehaus.org/*

[36] Ledlie, J., Ng, C., Holland, D. A., Reddy, K. K. M., Braun, U. and Seltzer, M. I. n.d., Provenance aware sensor data storage, Technical report, Division of Engineering and Applied Science, Havard Unversity.

[37] Lerner, R. M. 2005, 'At the forge: Working with activerecord', *Linux J.* **2**005(140), 12.

[38] Livny, I., Ioannidis, Y., Livny, M., Haber, E., Miller, R., Tsatalos, O. and Wiener, J. 1994, 'Desktop experiment management', *IEEE Data Engineering Bulletin* **1**6.

[39] Mayer, C. P. and Hübsch, C. 2009, 'Distributed Experiment Management for Large-Scale Testbeds'. Presentation at the 9th Würzburg Workshop on IP: Joint EuroNF, ITC, and ITG Workshop on 'Visions of Future Generation Networks' (EuroView2009).
URL: *http://doc.tm.uka.de/2009/HuebschMayerEuroView09-camerarady.pdf*

[40] Miles, S., Mureau, L. and et al., P. G. n.d., 'Provenance query protocol'.

[41] Miles, S., Victor Tan, U. o. S., Turi, D., Wolstencroft, K. and Jun Zhao, U. o. M. 2006, 'mygrid: An eu provenance case study'.

[42] Moreau, L., Freire, J., Futrelle, J., McGrath, R. E., Myers, J. and Paulson, P. 2008, The open provenance model: An overview., *in* J. Freire, D. Koop and L. Moreau, eds, 'IPAW', Vol. 5272 of *Lecture Notes in Computer Science*, Springer, pp. 323–326.
URL: *http://dblp.uni-trier.de/db/conf/ipaw/ipaw2008.html#MoreauFFMMP08*

[43] Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J. and Paulson, P. 2007, 'The open provenance model'.
URL: *http://eprints.ecs.soton.ac.uk/14979/*

[44] Munroe, S., Miles, S., Moreau, L. and Vazquez-Salceda, J. 2006, Prime: A software engineering methodology for developing provenance-aware applications, *in* 'Sixth International Workshop on Software Engineering and Middleware', ACM Digital.
URL: *http://eprints.ecs.soton.ac.uk/13062/*

[45] Ni, Q., Xu, S., Bertino, E., Sandhu, R. S. and Han, W. 2009, An access control language for a general provenance model., *in* W. Jonker and M. Petkovic, eds, 'Secure Data Management', Vol. 5776 of *Lecture Notes in Computer Science*, Springer, pp. 68–88.
URL: *http://dblp.uni-trier.de/db/conf/sdmw/sdmw2009.html*

[46] Olson, R. n.d., 'acts_as_paranoid documentation'.
URL: *http://ar-paranoid.rubyforge.org/*

[47] Osterweil, L. J., Clarke, L. A., Ellison, A. M., Podorozhny, R., Wise, A., Boose, E. and Hadley, J. 2008, Experience in using a process language to define scientific workflow and generate dataset provenance, *in* 'SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering', ACM, New York, NY, USA, pp. 319–329.

[48] *Plone's glossary on monkey-patching* n.d., `http://plone.org/documentation/glossary/monkeypatch`.

[49] Prud'Hommeaux, E. and Seaborne, A. 2008, 'SPARQL query language for RDF', World Wide Web Consortium, Recommendation REC-rdf-sparql-query-20080115.

[50] Reilly, C. F. and Naughton, J. F. 2009, 'Transparently gathering provenance with provenance aware condor'.

[51] *Ruby on Rails* n.d., `http://www.rubyonrails.rog`.

[52] Simmhan, Y. L., Plale, B. and Gannon, D. n.d., A survey of data provenance in escience, Technical report, Computer Science Department, Indiana University.

[53] Singh, M. P. and Vouk, M. A. n.d., 'Scientific workflows: Scientific computing meets transactional workflows'.

[54] Sproull, R. and Eisenberg, J. 2005, 'Building an electronic records archive at the national archives and records administration: Recommendations for a long-term strategy'.

[55] Steward, A. 2008, 'Peepcode press: Rails plugin patterns', `http://www.peepcode.com`.

[56] Tan, W.-C. n.d., 'Provenance in databases: Past, current and future'.

[57] Widom, J. 2008, Trio: A system for data, uncertainty, and lineage, *in* 'Managing and Mining Uncertain Data', Springer.
URL: *http://ilpubs.stanford.edu:8090/843/*

[58] Wirdemann, R. and Baustert, T. 2006, *Rapid web development mit Ruby on Rails*, Hanser, München (u.a.).

License