Normen Müller

Change Management on Semi-Structured Documents

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Date of Defense: 13 July 2010

Jacobs University Bremen
School of Engineering and Science

Dissertation Committee

Prof. Michael Kohlhase, Jacobs University Bremen, Germany
Prof. Herbert Jäger, Jacobs University Bremen, Germany
Prof. Bernd Krieg-Brückner, Universität Bremen, Gremany

To my parents.

Executive Summary

Organizations in all conceivable areas have processes to manage their business activities, often referred to as business processes. In today's competitive global economy, automation of processes with appropriate technology is advantageous. However, the paradox of processes automation is the continuous evolution and change that occurs in the documents describing and specifying the business processes. Recent analyses of the Association for Information and Image Management, the international authority on enterprise content management, indicate that our globalized information society produces, maintains, and publishes about 5 petabyte (i.e. ca. 3 trillion pages) of documents a year. Thus, an overwhelming amount of documents is produced and changed every day in nearly all areas of our every day life. Even after a decade of research in the areas of document engineering, organizations still find it challenging to manage such an evolution of business processes inscribed in collections of intentionally related and intertwined documents. Therefore, this thesis finds answers to the question of "How can heterogeneous collections of semi-structured documents be accurately and effectively maintained?"

While dedicated authoring and maintenance tools may provide some assistance, they typically are restricted to single documents or documents of a specific type. Thus, the propagation of changes and the identification of the resulting necessary adaptations and adjustments is highly restricted. To resolve that discontinuity, we present a methodology that embraces existing document types, allows for the declarative specification of semantic annotation and propagation rules inside and across documents of different types, and on that basis define semantic annotation and change impact analysis for heterogeneous collections of semi-structured documents in order to improve the maintainability and reduce the maintenance efforts.

The enabling idea is to represent in a single graph all related semi-structured documents together with that part of their intentional semantics contained necessary to analyze specific semantic properties of the documents as well as to analyze the impact of changes. When a change takes place on either the syntactic or the semantic level of a document, our framework creates a propagating impact on the involved elements. This propagation of impact takes place due to constraints, associations, and dependencies among elements within and across documents. The whole methodology is based on document models defining the syntax, semantics and annotation languages for specific document types as well as graph transformations to obtain the semantic annotation and to propagate the effect of changes for documents of this type. For the interaction between documents of different types it builds on interaction models specifying graph transformations propagating semantic information over document boundaries. The methodology is implemented in the prototype tool *locutor* built on top of the graph rewriting tool GrGen.NET. Annotation and propagation rules can be specified in the declarative GrGen.NET syntax and used to semantically annotate collections of documents and to analyze the impact of changes on the entire collection. As a running example for this work we consider a wedding planning scenario, where two types of documents occur. The first document represents the guest list and the second one the seating arrangement. The latter depends on the former with the condition of male and female guests being paired. Using this simple example, we explain the complex practice of verification of consistency and identification of ripple effects.

Major results of this research include a set of document models, a set of algorithms that allow authors to evaluate proposed changes, and the prototype tool *locutor* to evaluate the algorithms. The core contribution of this research is the identification of the individual constituent parts of a change management system and their interaction with each other, which enables accurate and effective management of documents describing an organization's business activities.

Acknowledgments

To begin with, I wish to thank my dissertation committee, Bernd Krieg-Brückner, Herbert Jäger, and Michael Kohlhase in general for their endurance and patience with me as well as with the completion of this work.

In particular, I would like to thank my primary advisor Michael Kohlhase. Michael, do you remember how everything started? I do! On our first joint brainstorming meeting in Paris, France, I was pretty frustrated. I still had not found the right subject. Despite all the hustle and bustle there, you took your time to have one of your infamous walks with me — well, I rather call it jogging. During this walk, the MoC idea was born. Thank you so much for this initial support. But even in the next few years, you kept on looking after me with an overwhelming peace and clarity, so that today the here presented work emerged. Thank you sincerely!

Furthermore, I would like to specially stress my co-worker Serge Autexier. Many discussions with him have in particular improved my change impact analysis approach. Even the writing of our first joint paper was a lot of fun and opened up new insights to me. Serge, a heartfelt thanks to you!

The KWARC group has of course contributed their share to this work. May it be on the social level (Cheers!) or at the scientific level (Help!). All have been with me all the time helping me out any way they could. Thank you so much Christoph, Florian, Fulya, Slava and of course my future wife, Christine!

But not only the KWARC group stood by me, but also the developers of GrGen.NET. Together, we have removed some crucial bugs, or in fact I have discovered the bugs and Edgar Jakumeit, Sebastian Buchwald, and Rubino Geiss have removed them with an enormous speed. I thank you for this support!

Last but not least, a warm thanks to the SVNKit team, in particular, Alexander Kitaev, Alexander Sinyushkin, and Semen Vadishev. Also you guys have contributed significantly to the *locutor* system that today is there where it is. SVNKit (team) rocks!

Contents

Part I

Introduction & Preliminaries

Chapter 1

Introduction

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— Bertrand Russell

The present decade is — not without reason — often called the "information age". Information, or rather, the spread of information, has soared with the advent of new technologies and new means of communication, from the invention of the telephone in 1875 through the expansion of radio and later television broadcasting until the breakthrough of computers in the 1970s. Last, but certainly not least, the World Wide Web was opened to the public in 1993. Since then the number of users has exploded, so that today billions of people around the globe are connected to one another and have access to huge amounts of data.

The potential benefits of this connectedness and availability of data are manifold. Computers and the Internet have penetrated every aspect of our everyday lives. Privately, we stay in contact with friends and family regardless of physical whereabouts via e-mails and chat programs, keep up with the latest news thanks to countless online papers and radio broadcasts, and read up product reviews by other customers before buying a new DVD recorder. Along the same lines, our workplace has adapted to this digital environment. We start a typical working day with reading and answering e-mails, go on to check market prices in Asia and the Americas online, have a telephone-conference with business partners located in the Asian headquarters, and continue working on a collaborative project paper, to which colleagues from different parts of the world contribute. New storage technologies, higher computation speeds, and system integration lifted many of the traditional restrictions on the conversion of paper to electronic documents.

All this leads to the assumption that the "electronic office" is now finally a reality. The current situation shows, however, that paper currently is still in the offices and will be for the foreseeable future, so will not change quickly. At this point it is comparable with the situation in the automobile industry more than forty years ago: complex processes, minimal automation, and many people. Today, in the automobile industry processes are perfectly organized and synchronized, the production is largely automated and deserted.

Office communication is now on a similar way, such that, while the "electronic office" may remain an illusion, the "paperless office" is reachable with today's technology. As already described, a strong shift to digital content has taken place and many transactions are electronically displayed. Day-to-day office applications sitting on our computers allow us to read, create and edit a range of different types of content (texts, drawings, spreadsheets etc.) and store them onto our hard drives as different types of office documents (for example, a word processed text file, a spreadsheet of figures or a presentation). Due to the diversity of such applications but the claims of interoperability among these, the utilized document exchange formats and their handling became a high profile issue.

1.1 Semi-structured Documents

Data is a precious thing and will last longer than the systems themselves.

— Tim Berners-Lee

In the early days of personal computers there were many word processing and other office related applications available. These applications usually made use of binary format files, i.e. the human readable content (data) was encoded into a machine-readable representation of the data in binary form [52]. The exact details of the representation or encoding were often a proprietary standard and undocumented and thus difficult for software from other vendors to read or process. This means that content has become deeply coupled with the software that was used to create and handle it. The problem with this was that because there were so many different software packages, which were invariably unable to read another vendor's format users found it very difficult to exchange documents with each other.

Since the 1960s computer scientists have worried about the lack of interoperability and exchangeability of documents between different software applications and there has been an ongoing move towards developing a common document format. Debates about commonality also took place in parallel to discussions about abstracting — the ability to abstract the meaning of information in a document and separate this from its rendition (i.e. presentation) [52]. These discussions led to the development of the Standard Generalized Markup Language (SGML) in the late 1970s and early 1980s.

During the 1990s, the World Wide Web, however, changed the digital information rules. The extreme simplicity of the hypertext mark-up language (HTML) and the universality of the hypertext transfer protocol (HTTP) decreased the cost of authoring and exchanging information. We were suddenly exposed to a huge volume of information; this kind of information was, of course, not new, but the volume was unlike anything seen before. The impact in our daily lives was also tremendous emerging in the subsequent dilemma: it became clear that this rich information could not be stored in relational databases or queried and processed using traditional techniques. We had reached the limits of what we could handle using the traditional rules and needed new technologies. In addition to the pure (unstructured) HTML data on the Web, more data was available in a form that did not fit the purely structured relational model, yet the information had a definite structure — it was not "just text".

This gray area of information is called *semi-structured data*, the term database theorists use to denote data that exhibits any of the following characteristics: (1) numerous repeating fields and structures in a naive hierarchical representation of the data, which lead to large numbers of tables in a second-normal or third-normal form representation[1] (2) wide variation in structure, and (3) sparse tables. Examples of semi-structured data are SGML (or HTML) files or more generally data exchange formats (e.g. ASN.l), hypertexts and programs. Semi-structured information also typically arises when integrating several (possibly structured) heterogeneous sources.

[1]We refer the interested reader to [75].

From here on we consider this gray area of information in the same context database theorists do. We call semi-structured data this data that is (from a particular viewpoint) neither raw data nor strictly typed, i.e., not table-oriented as in a relational model or a sorted-graph as in object databases.

A lot of research has been devoted to this gray area of information, such that, as part of its work in the 1990s, the World Wide Web Consortium (W3C) developed a subset of SGML that would retain SGML's major virtues but also "embrace the Web ethic of minimalist simplicity" [52]. This new language was the Extensible Markup Language (XML). The benefits of XML — exchangeability, reusability, human readability and representation of semi-structured data — are widely seen and taken up by a large variety of different information management and software communities. They lead XML to find its way into our day-to-day business. XML has become the *lingua franca*, for the interchange of data between software, computer systems, documents, databases etc. and as a format for semi-structured data storage.

On that note, we define a **semi-structured document** as a snapshot of some valid XML information set that (1) incorporates many complex information types, (2) exists in multiple places across a network, (3) depends on other documents for information, (4) changes on the fly (as subordinated documents are updated), (5) has an intricate structure and (6) can be accessed and modified by many people simultaneously.

With our assumption that *any* document is representable in XML, semi-structured documents which fit the definition exist in many forms. Examples range from agreements in contracts, complex instruction manuals, business forms, memos, books, software specifications, source code, mathematical knowledge through to websites. Henceforth the term "document" comprises any kind of such semi-structured documents.

The prevalence of XML bears some challenges. In everyday business several different documents reflecting entire system life cycles are passed, commented, corrected, and returned provoking costly, tedious, and error-prone factors in *managing* document life cycles. To avoid inefficiencies, conflicts, and delays, as well as to emphasize the importance of common information spaces in decentralized working environments the integration of an XML-aware system support is indispensable.

1.2 Document Management

We cannot solve our problems with the same thinking we used when we created them.

— Albert Einstein

In the information age the sheer amount of documents literally available at our fingertips is overwhelming. Recent indications by the Association for Information and Image Management (AIIM), the international authority on enterprise content management, declare that our globalized information society produces, maintains, and publishes about 5 petabyte (i.e. 2^{50} bytes, 1024 terabytes, a million gigabytes, or ca. 3 trillion pages) of documents a year. This immense amount of documents has created the problem of finding the necessary information in a timely manner. All these documents are very often saved to different locations, are incoherent, not enterprise-widely accessible, and hence not immediately available. Many companies do a better job managing and securing their office supplies than their business-critical documents. However, no matter what industry or size, gaining control of all documents is critical to every organization. Ever-expanding government regulations require effective and auditable control systems for all documents and communications. Competitive pressures require that organizations become more efficient and responsive in order to survive and thrive.

Generally, however, documents are left unmanaged or even stored in filing cabinets around the office. This sets up organizations for compliance risks, service delays, cost overruns, and a host of other challenges. The documents that are the very lifeblood of the modern business are all too often taken for granted. Very few businesses take the time to consider the expenses that they incur on a daily basis because of (1) time and effort wasted in locating documents, (2) redundant effort necessitated because it is often easier to recreate something than it would be to try to find it, (3) time and effort involved in figuring out who has the latest version of a document and in recovering when various revisions overwrite each other, and (4) unnecessary usage of network storage devices and network bandwidth because the documents are dispersed everywhere across the enterprise, rather than centralized.

Likewise, few businesses take the time to consider the risks that they expose themselves to on a daily basis because (1) security is applied haphazardly at best, which exposes important information to scrutiny by potentially inappropriate people, (2) critical documents are stored – often exclusively – on laptop computers that could be lost, stolen, or damaged at any time, and (3) documents stored centrally on Windows network drives, once deleted, do not go into a recycle bin as commonly believed. They simply disappear, and must be restored slowly from tape backups (if you are lucky enough to have those). Finally, (4) no record exists of precisely who has viewed and/or edited a document. It is therefore impossible to audit a business process to uncover mistakes or inefficiencies.

It is possible, though, to develop an application around a database that adds appropriate features to support these requirements and fortunately this has already been done, and packaged as document management systems by a number of software vendors. However, there is considerable confusion in the marketplace regarding the respective definitions. The scope and role of specific document management systems is particularly blurry, in part caused by the lack of consensus between vendors. With the aim of lessening this confusion, the subsequent paragraphs provide an at-a-glance definition of terms for a range of information management systems.

Content management system. Content management systems (CMS) support the creation, management, distribution, publishing, and discovery of corporate information. Also known as "web content management" (WCM), these systems typically focus on online content targeted at either a corporate website or intranet.

Enterprise content management system. An enterprise content management system (ECMS) consists of a core web content management system, with additional capabilities to manage a broader range of organizational information. This often consists of document management, records management, digital asset management or collaboration features.

Records management system. A record management system (RMS) is defined by the Australian Standard on Records Management (AS 4390) as record keeping systems to be "information systems which capture, maintain and provide access to records over time". This includes managing both physical (paper) records and electronic documents.

Document management system. Document management systems (DMS) are designed to assist organizations to manage the creation and flow of documents through the provision of a centralized repository and workflows that encapsulate business rules and metadata. The focus of a DMS is primarily on the storage and retrieval of self-contained electronic resources, in their native (original) format. Standard features include *locking*, the ability to prevent any of the following actions to be applied to a document, *check-out*, the ability to assign a document to an author, while locking it to prevent other authors from accessing it (a 'check-in' operation places updates back into the system, and unlocks the document), and *versioning*, the ability to preserve the original document after an updated version is checked-in.

All these systems are different, however, in terms of where semi-structured data can be used, they are similar enough to justify lumping them all together under one label. Recalling the fact that we consider the term "document" to comprise any kind of semi-structured documents as defined in Sec. 1.1, this work defines **document management** to be the automated control of (electronic) documents – page images, spreadsheets, word processing documents, and complex, compound documents — including all the variations that are appropriate to enable the creation, capture, organization, storage, retrieval, manipulation, and controlled circulation of documents.

Traditionally, there have been two classes of document management: (1) management of fixed images of pages (the class that seems to be most familiar to librarians) and (2) management of editable documents, such as word processing files and spreadsheets. These two classes differ largely in the fact that images are static, while editable documents are dynamic and changing. The functions associated with the two classes differ as well. Systems supporting images focus on access, with input, indexing, and retrieval as important functions, while systems supporting editable documents focus on creation, with joint authoring, workflow, and revision control at the centre.

The barriers between these two classes of systems, however, are breaking down. Vendors are moving away from specializing exclusively in one class or the other, with a trend toward creating larger, integrated document management systems that incorporate a full range of document management functions. Such systems control the creation and use of documents throughout their life span — across platforms, applications, and company organizational units.

It is important to note that, following our definition, document management is not yet a single technology, but several. The major challenge at this time is the integration of various software packages — those for image storage and retrieval, workflow management, compound document management, and document presentation — into a single integrated system. Therefore the elements of such document management systems have to cover software components to perform all these functions necessary to manage documents across an organization from cradle to grave. Each element is described below [129].

Underlying infrastructure. While not part of an application *per se*, an appropriate underlying infrastructure is a prerequisite for supporting an integrated DMS. The infrastructure is the set of desktop computers, workstations, and servers that are interconnected by LANs and/or WANs. It must have characteristics such as network operating system independence, file format independence, location independence, long file names, and link tracking.

Authoring. Authoring tools support document creation. Some more sophisticated tools support structured or guided authoring, where authors are constrained by the system to enter data in specified ways. Typically, they are interfaced with a DMS in order to capture document metadata at the time of creation and revision.

Storage. The core of a DMS is a database and search engines supporting storage and retrieval of documents. Traditionally, relational DMS are moving toward object-oriented databases. However, most vendors are now using mixed databases with relational databases used to point to information objects. Such databases are called Object-Relational Database Management Systems.

Presentation/Distribution services. The presentation and distribution concerns the form and manner in which users are provided with information. Document management systems should allow “multi-purposing” where information can be distributed in different formats, such as viewed on a network (e.g., the Web), distributed on CD-ROM, or printed on paper. Businesses can reuse information, putting it into a format determined by the target market or business function. On-demand printing, where a document is printed when it is needed from a document database, is growing in popularity and importance.

Workflow. A workflow is defined as the coordination of tasks, data, and people to make a business process more efficient, effective, and adaptable to change. It is the control of information throughout all phases of a process. The path of a particular document is determined by the document type (e.g., press releases, manuals, policy papers, memos), the processes governing a document, and organizational roles (i.e., who has the authority to see what?). It supports functions such as authoring, revising, routing, commentary, approval, conditional branching, and the establishment of deadlines and milestones.

Library services. Not to be confused with what librarians consider to be library services, this is a term used specifically by the document management community to refer to document control mechanisms such as check-in, check-out, audit trail, protection/security, and version control.

The trend toward semi-structured documents and large, integrated document management systems, however, raises serious issues on several of these elements. A document management system been implemented, requires almost inevitably further steps. Topics such as the creation process and automatically controlled workflows on *modifications* are the immediate entailment. A document may be created by one person, by a team of people or an entire department. A budget may be made available. There must be a workflow plan in place to help facilitate the documents distribution from the creator to managers, to the executives in charge of finances and to track changes along the path. The current status of the traditional processes for the creation of documents, the exchange, the revising, and the subsequent updating is time-consuming and very inefficient. In fact, managing such document circulations and revisions has become impossible for humans without computer assistance. Computer-supported systems for *management of change* (*aka.* change management) are thus indispensable for the creation and maintenance of documents. If two or more persons have to work concurrently on a document, the problem clearly emerges: Who has the very latest version of the document? What changes were made by whom, when? What effects do the changes entail after the draw, i.e. which documents are affected by the changes?

Even if we assume that only 3 percent of the overall document corpus are mission-critical documents that will be maintained over time, we are still faced with a huge management problem, which is aggravated by the fact that documents are inter-related (we speak of *document collections*) and that changes in one, e.g., a company’s mission statement, will make changes in others necessary, e.g., planning documents or the company’s website. This is an iterative, incremental challenge. With the failure rate of document management projects estimated at around 50 percent [1], the incorporation of a change management component is an extremely important question to answer before embarking on any document management project. Therefore we recognize the workflow and library services components as central aspect of document management we want to elaborate on to the extend of *change management*. Incorporated into large integrated DMS, these major components allow organizations to get control of and to increase the efficiency of the flow and maintenance of documents that support their business.

1.3 Change Management

Obviously, a man's judgment cannot be better than the information on which he has based it.
— Arthur Hays Sulzberger

Already Heraclitus knew "Nothing is more constant than change" at about 500 years BC. Transformation and change are not a new phenomena, only the speed today's organizations are forced to change, has increased.

The term "change", in general, refers to any modification in equipment, manufacturing materials, facilities, utilities, design, formulations, processes, packaging/labeling, computer systems, and all associated documentation (e.g. standard operation procedures (SOP), quality manuals, etc.). In its most basic interpretation this means that change is the *physical modification, insertion or deletion of a string in a document*. Depending on the type of the document and the exact location of the change its semantics vary greatly. A change may be a simple adjustment brought on by a new customer specification, or an updated document, or a part replacement, or other production needs. It may be caused by a deviation from an approved regulatory filing or written procedures. A change may be temporary or permanent, routine or emergency, innocuous or serious enough to shut down production. The fact that change is inevitable makes management and control a critical factor.

Unfortunately, similar to document management systems (cf. Sec. 1.2), there exits a controversy about the concepts of Change Management and Change Control, because the terms are not clearly delineated. A useful perspective on the distinction between change control and change management — note, we use the terms change management and management of change interchangeably — comes from the definitions of these terms used by the Information Technology Infrastructure Library (ITIL) Service Support processes [70]:

Change Control. The procedures to ensure that all changes are controlled, including the submission, recording, decision making, and approval of the change.

Change Management. The Service Management process responsible for (1) controlling and analyzing changes, (2) managing the effects of changes on the IT Infrastructure, or any aspect of IT services, and (3) to promote business benefit while minimizing the risk of disruption to services.

One can infer from these definitions that change control is a process that is largely internal to the IT department and focused on prioritizing and approving infrastructure changes exclusively based on their technical merit and technical impact. IT technicians often see formalized change control as an administrative burden restricting their ability to react quickly to the rapid pace of change in an increasingly complex business/technology environment.

For the great majority of changes — routine changes with no great impact on users or the business (e.g., moving an application from one server to another because of a conflict, etc.) — the change control protocol seems a pointless formality that stands in the way of getting the actual work done. After all, everyone in IT probably already knows through informal communications what has to be done and why and in what timeframe, so the approval process functions as little more than a bureaucratic exercise and is often ignored or only receives minimal compliance.

The problem is that changes to the infrastructure are not always routine and light in terms of impact to the infrastructure and to the organization both inside and outside of IT. It is in these cases that simple change control does not have the appropriate depth of process or organizational reach to handle the complex change events it is charged with regulating. On top of this, the weak compliance that the process receives for low impact changes is often little better for changes with a significant organizational impact. The results are poor decisions are made to go forward or deny changes, business impacts to areas of the business are not considered, changes are badly prioritized, and implementations of changes are disruptive. In essence, many organizations with IT-focused change control process regimens have too much process for many simple changes and not enough process for the major changes that really matter.

With the renewed attention on "best practices" and Service Management, many IT organizations are recognizing that the complex impact of infrastructure changes extend across many enterprise groups. With this recognition comes the realization that infrastructure changes need be managed and not simply controlled. In fact, change control is an important but subordinate procedural component of a robust change management process. The additional significant components, the "secret sauce" of change management that makes it effective are procedures that emphasize assessing the business impact and ramifications of a change and the communication and coordination activities involved in evaluating, approving and implementing a change.

The broader goal of change management is to ensure that the service risk and business impact of each change is communicated to all impacted and implicated parties, and to coordinate the implementation of approved changes in accordance with current organizational best practices. The result of effective change management is that changes are handled quickly in a uniform way and have the lowest possible impact on service quality.

The British Standards Institute in its Code of Practice for IT Service Management (BS 15000) defines the scope of change management to include the following process steps:

Recording Changes. In practice, the basic details of a change request from the business are recorded to initiate the change process, including references to documents that describe and authorize the change. Well-run change management programs use a uniform means to communicate the request for change, and work to ensure that all constituents are trained and empowered to request changes to the infrastructure.

Assessing the Impact, Cost, Benefits, and Risks of Changes. The business owner of a configuration item (i.e., a CI in the Configuration Management Database which records the exact state of the IT infrastructure) to be changed (e.g., IT for infrastructure, Finance for a Billing application, etc.) and all affected groups (e.g., users, management, IT, etc.) are identified and asked to contribute to an assessment of the risk and impact of a requested change. Through this means, the process is extended well beyond the IT department and draws on input from throughout the organization.

Developing the Business Justification and Obtaining Approval. Formal approval should be obtained for each change from the "change authority". The change authority may be a person or a group. The levels of approval for each change should be judged by the size and risk of the change. For example, changes in a large enterprise that affect several distributed groups may need to be approved by a higher-level change authority than a low risk routine change event. In this way, the process is speeded for the routine kinds of changes IT departments deal with every day.

Implementing the Changes. A change should normally be made by a change owner within the group responsible for the components being changed. A release or implementation plan should be provided for all but the simplest of changes and it should document how to back-out or reverse the change should it fail. On completion of the change the results should be reported back for assessment to those responsible for managing changes, and then presented as a completed change for customer agreement.

Monitoring and Reporting on the Implementation. The change owner monitors the progress of the change and actual implementation. The people implementing the change update the configuration management database proactively and record or report each milestone of change. Key elements of IT management information can be produced as a result of change management, such as regular reports on the status of changes. Reports should be communicated to all relevant parties.

Closing and Reviewing the Change Requests. The change request and configuration management database should be updated, so that the person who initiated the change is aware of its status. Actual resources used and the costs incurred are recorded as part of the record. A post-implementation review should be done to check that the completed change has met its objectives, that customers are happy with the results; and that there have been no unexpected side-effects. Lessons learned are fed back into future changes as an element of continuous process improvement.

As the above process makes clear, true change management differs from change control in the depth of its overall process scope and in the range of inputs it uses. Where change control ensures that changes are recorded and approved, change management considers overall business impact and justification, and focuses not only on the decision to make or not make a given change, but on the implementation of the change as well as the impact of that implementation. Where change control chiefly takes into account the IT perspective on changes, change management draws information from and relays information to constituents throughout the organization at every step of the change process.

We consider document collections to be themselves major constituents. We base our view on the fact that *business processes are inscribed/described within document collections.* Business processes can be seen as *document-centric* processes as in each step preparation, manipulation or verification of documents is involved. Typical examples of such "business processes" are requirements specifications, design documents, source codes and documentations, all of which may use different underlying document formats but as a whole such document collections require intra- and inter-relation maintenance and management. Therefore, we apply change management on documents rather than *only* change control. From here on, when ever we refer to "business processes" we mean document-centric processes.

1.3.1 Document Change Management

In the information age, consumers worldwide have become more aware about product quality. In response, manufacturers are changing their business philosophy. Customer satisfaction and continuous improvement of product quality have become the objectives not only of regulatory bodies, but manufacturers themselves. At the operational level, the focus is moving from detection to prevention. Companies recognize that it is their primary responsibility to determine if a proposed change could significantly affect safety or effectiveness of a product — and not for the FDA inspector or ISO auditor to spot. Manufacturers increasingly realize that effective change management is integral to continuous quality improvement, which can ultimately help them increase customer satisfaction and prevent product recalls, product liability actions, and regulatory violations.

Maintenance of document life cycles, however, is still poorly understood and loosely managed worldwide, although it is generally recognized to consume the majority of resources in many organizations. Standard document management systems have major difficulties in assuring consistency after modifications. This issue is rooted in the insufficient recognition of the benefits to be derived from formal methods. A formal procedure for document maintenance is essential for two reasons: it provides a common communication channel between maintenance staff, users, project managers and operations staff, and it provides a directory of changes to the system, for status reporting, project management, auditing and quality control. It supplies managers with more accurate information and more useful guidelines to aid them in improving the decision-making process, planning and scheduling maintenance activities, foreseeing bottlenecks, allocating resources, and optimizing the implementation of change requests by releases.

At present time human reviewers are needed for managing changes, which is perfectly suitable for documents of a certain, relatively small size. However, a typical document size exceeds the limit for human reviewing. As a result changes are not always properly controlled. Experience shows that making changes without understanding their effects can lead to poor effort estimates, delays in release schedules, degraded design, unreliable products, and the premature retirement of an entire system. Human reviewing is bound to become a costly, tedious, and error-prone factor in document life cycles that is often neglected in cost reduction and likely to lead to sub-optimal and often disastrous results.

A famous disaster that can be traced back to deficient document management is certainly the unsuccessful launch of the European rocket Ariane 5, which exploded on June 4, 1996, on its maiden flight due to a software malfunctioning. With software specifications from the successful parent, Ariane 4, Ariane 5's reused code did no longer fit its improved flight path, thus causing the automated self-destruction system to go off. The Ariane disaster resulted in a loss of about 500 million US-dollar plus considerable delays in the development of the Ariane launch systems.

Consequently, document management requires information not only to be organized in storage systems, but also useful integrated into electronically controlled change processes. The elaboration of a formal procedure making such control processes possible is the core of this work. It facilitates the maintenance and preservation of consistency and completeness of a development during its evolution and therefore should be used at all stages of a document life cycle support. Clearly, we hope that with an integration of our formal procedure for *change management on semi-structured documents* reduction in effort can be achieved by minimizing the time between a proposed change, its implementation, and its delivery, while at the same time maintaining quality.

Recalling the former disambiguation of the concepts Change Management and Change Control, our objects of interests are document collections in sense of being themselves major constituents to be managed throughout an organization at every step of a change process. Therefore we define **document change management** as the process to ensure that all changes on a document collection are *controlled* in terms of recording and analyzing as well as *managed* in terms of promoting impacts of each change to all impacted and implicated parities, i.e. all inter-related (intra-related) documents (fragments).

Our document change management allows maintainers to assess the consequences of a particular change within a document on the respective document collection and can be used as a measure for the effort of a change. For instance, the more a change causes other changes to be made, the higher the cost. Carrying out this analysis before a change is accomplished allows us to assess the cost of the change and allows management to make a trade-off between alternative changes. Henceforth the term "change management" and "management of change", respectively, denotes document change management.

1.3.2 Elements of a Change Management System

Change management helps us think through the full impacts of a proposed change, i.e. the consequences of a change and to adjust affected items. As such, it is an essential part of the evaluation process for major decisions. More than this, it gives us the ability to spot problems before they arise, so that we can develop contingency plans to handle issues smoothly. This can make the difference between well-controlled and seemingly-effortless project management, and an implementation that is seen by your boss, team, clients and peers as a shambles.

In order to accomplish this, we first, however, need to understand the *process of change* and to define and validate a *methodology* both taking the specific characteristics of document maintenance into account. According to Madhaji [83] we define the steps of a **process of change** (PoC) as follows:

(PoC 1) Identify the need to make a change to an item in the environment.

(PoC 2) Acquire adequate change related knowledge about the item.

(PoC 3) Assess the impact of a change on other items in the environment.

(PoC 4) Select or construct a method for the process of change.

(PoC 5) Make changes to all the affected items and make their inter-dependencies resolved satisfactorily.

(PoC 6) Record the details of the changes for future reference, and release the changed item back to the environment.

The key problem in accommodating changes in an environment is to know all the factors that impact a given change, the consequences of this change and how to execute respective adjustments. The two most expensive activities in accomplishing this challenge are the understanding of problems or other expressed needs for change, in relation with the understanding of the maintained document. Second, the mastering of all the *ripple effects* of a proposed change, i.e. the effect caused by making a small change to one document which impacts many other parts. A seemingly small change can ripple throughout a document collection to have major unintended impacts elsewhere. As a result, authors need mechanisms to understand how a change to a document will impact the rest of the collection.

The purpose of our change management is to determine the scope of change requests as a basis for accurate resource planning and scheduling, and to confirm the cost/benefit justification. Think of when things change in your organization, do you ever wish that someone would think things through a little better to avoid the confusion and disruption that often follows? Or have you ever been involved in a project where, with hindsight, a great deal of pain could have been avoided with a little more up-front preparation and planning? Well, hindsight is a wonderful thing — but so is management of change (MoC).

Clearly, the goal of MoC is to see what would happen if a change occurs, before the change really takes place, and to ease adaption afterwards. This information can then be used to help in making a decision on the necessity of a change. Our management of change methodology implicates these functions by providing analyzing and adjustment mechanisms for building up a *management of change system*, i.e. a document management system utilizing management of change methods. In Fig. 1.1 we illustrate our proposed conceptualization of a change management system.

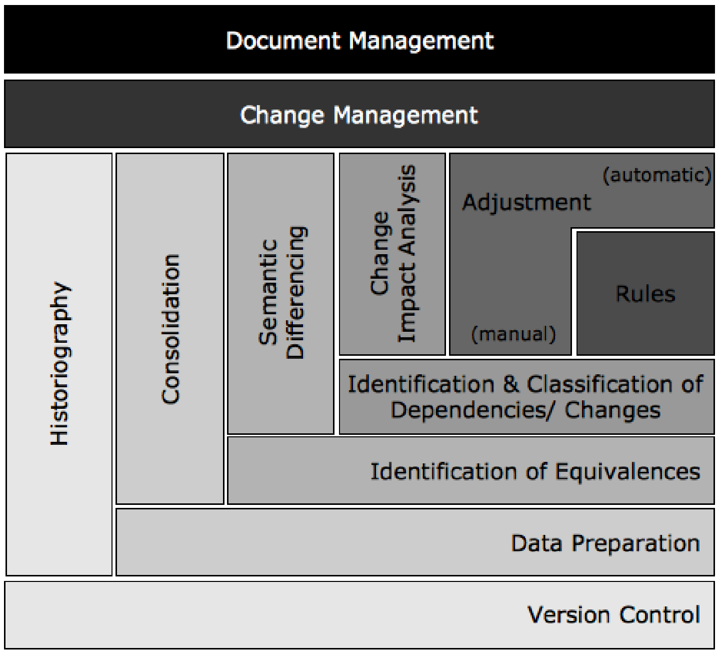


Figure 1.1: A Change Management System.

In contrast to [92], we subsume change impact analysis by change management. We consider change impact analysis as the technique to preview effects of changes, whereas the entire machinery of MoC serves as the instrument to (semi-) automatically adapt affected items and to facilitate inconsistency-correction, respectively. The icing on our MoC cake is the explicit requirement of dependency and change classification. Here we emphasize that maintenance processes are only feasible if precise and unambiguous information on the potential ripple effects of a change is available. In the following we give an overview on each piece of our MoC cake (but the DMS component discussed in Sec. 1.2). In detail descriptions are given in Part II.

Historiography

Version control systems (VCS) serve as a basis for our management of change methodology (cf. Chap. 3). A VCS can be characterized as a system which tracks the history of file and directory changes over time. All version controlled files and directories reside as a tree structure in a system distinct repository and both can have version controlled metadata. Changes to such a tree are transactional resulting in a new snapshot (*aka.* revision) of the whole tree including all recent changes made in that commit operation as well as all previous unchanged data.

Common version control systems (cf. Sec. 3.1.2) try to modify a version controlled file system tree as safely as possible. Before changing the current tree, the to be committed modifications are written to a log file (*aka.* journal). Architecturally, this is similar to a journaled file system. If an operation is interrupted (e.g., if the process is killed or if the machine crashes), the log files remain on disk. By re-executing the log files, the system can complete the previously started operation, and the file system tree can get itself back into a consistent state.

Consequently by ensuring transactional recording of details of changes for future reference and releasing changed items back to the corpora, version control systems cover PoC 6.

Consolidation

Within a document collection modifying and then re-validating one document is complicated: analysis and consistency checking are required for each dependent document and the relations among them. The problem is further compounded because the maintainers are sometimes not the authors and may lack contextual understanding. Finally, cross-project dependencies on document collections mix up the control process even worse. As a document collection ages and evolves, the task of maintenance becomes more complex and more expensive.

To ease maintenance of complex corpora the second slice of our layered MoC cake servers for identification of the *coarse-grained* constituents (cf. Chap 4). These are the version controlled file system trees interrelated to each other. Here a data preparation provides a global view of heterogeneous and distributed document collections by conflating relevant metadata of the identified data sources to a coherent data base (one may think of a Data Warehouse or a Metadata Registry). In this sense we consider consolidation as the process of centralizing and sharing of resources. To keep consistency, we utilize the journaling functionality of the underlying version control system.

Integration of metadata from various corpora resulting in a coherent data pool enables overall evaluations and thus lays the foundation for acquiring adequate (cross-project) change related knowledge about the item subjected to change. In this way we incorporate PoC 2.

Semantic Differencing

Before we can analyze impacts of changes, we have to identify them — just because something is different, does not mean anything has changed! Change detection is the process of automatically detect and pinpoint the occurrence and localization of a change within a document with as high precision as possible.

Most previous work in change detection has focused on computing differences between text-based (*aka.* flat) files. The UNIX `diff` utility is probably the most famous one. This algorithm uses the LCS algorithm [106] to compare two plain text files. Version control systems, like CVS [48] and SUBVERSION [28], use *diff* to detect differences between two versions. Changes are localized by line to line comparison and indicate the location of strings modified, deleted or inserted by providing its exact starting and ending line and column numbers. In the scope of this work, however, it is assumed that documents are semi-structured. Thus the localization of changes has to be performed by indicating the respective position in the tree structure.

Chawathe et al. [23] already pointed out, though, that LCS techniques cannot be generalized to handle semi-structured data because they do not understand the hierarchical structure information contained in such data sets. Typical hierarchically semi-structured data, e.g. XML, place tags on each data segment to indicate context. Standard plain-text change-detection tools have problems matching data segments between two versions of data. Clearly, it is not sufficient to compare XML elements as strings: syntactically different elements may be semantically equivalent.

Hence, the intermediate layer of our MoC cake provides us with the capability of handling tree-structured documents by introducing a semantic notion of similarity between individual segments. These equivalence systems give us a stronger notion of equality leading to more compact, less intrusive edit scripts (cf. Chap 5). These extend the information acquired by the data preparation and in turn serve as the base for the subsequent slices.

Change Impact Analysis

Teams involved in collaborative processes often follow a divide and conquer approach. They try to split the project into tasks a single person can work on. However, there are times when team members must work on the same task. One person may work on one artifact for a while, and then pass it on to the next person perhaps to continue the work or to revise it as required. Iterative development can then occur between these two people in a back and forth manner, or it could even include other team members. Furthermore, it is a common practice one team member working on one task is implicitly modifying dependent ones commissioned to other team members. The challenge is in bringing the deliverables together cohesively, coherently, and consistently. A major problem here is how team members can recognize and/or track changes made in a document.

Change impact analysis is a formal process to provide users with the ability to identify changes and to "identify the potential consequences of a change, or to estimate what needs to be modified to accomplish a change" [14]. The essential part is to predict the system-wide — in sense of within and between document collections — impact of a change request before actually carrying out modifications to the system, so that appropriate decisions related to the change request can be made, such as planning, scheduling and resourcing.

We base our formalism for change impact analysis on graph rewriting initiated by differences between two document versions. We communicate via changes rather than orally or by notes within the document. In order to ripple effects of a change throughout a document collection we represent documents in a *semantic document impact graph*. Such a SDI graph consists of the documents, the semantic information computed from the documents and the impact information containing explicit information on how the document semantics is affected. The latter denotes the implicitly modified artifacts and is computed by application of graph rewrite rules on the entire graph initiated by the explicitly modified information units.

As any recursive analysis like this, however, can produce large numbers of results, which can often be meaningless to the user due to the sheer volume of information, we scope our change impact analysis by classifying changes as well as dependencies according to various types. A change is only propagated along a dependency relation if its type correlates to the type of the dependency. This information filtering technique gives control over the context of the change impact analysis. Additionally it provides user-specific views on the differences between document versions by restricting the analysis to specific relationships to be followed. Without filtering, users may become bombarded with volumes of changes that they may or may not always be interested in or which are out of their responsibility. The facility to traverse a filtered SDI graph empowers users to individually assess the impact of a change on other items in the environment as a basis for accurate change tracking.

We thus consider this layer of our MoC cake to cover PoC 3. Consequently, eating the cake up to this slice, engineers/authors can answer questions such as:

- Have any changes occurred since I last visited this document?
- How many changes have occurred?
- Where have these changes happened?
- How have particular parts of the entire collection changed?
- Who did these changes?
- Why did they perform these changes?

These questions were derived from similar questions raised by Gutwin et al.[58], who was studying how people would track what others were doing when working together in real time over a visual work surface. Gutwin et al. was interested in what he called Workspace Awareness. While related, here we focus on awareness of changes in an asynchronous environment designed for authoring processes rather than a real-time system. The immediate benefits of our approach are improved accuracy of resourcing estimates, hence, better scheduling, a reduction in the amount of corrective maintenance, because of fewer introduced errors, and improved system quality.

Adjustment

The next step following change impact analysis is to actually carry out the identified modifications – adjustment (*aka.* change propagation). Clearly, change impact analysis refers to the *identification* of all system modifications due to a change request and change propagation refers to the *execution* of those. In terms of management, making the initial modifications is straightforward. The challenges come from making the consequent modifications to re-establish system consistency. Therefore, our basic tool support involves advising the user the artifacts to be modified and the types and order of the modifications. In certain well-defined cases, the change process can be further assisted by codifying rules for change propagation in the environment and letting the environment to carry out some of the modifications automatically. In general, adjustment is realized through a combination of rule-based automatic change propagation and (in most cases) tool-guided user intervention. We call this process *semi-automated adjustment*.

Similar to our formalism for change impact analysis, we back up change propagation with graph rewriting techniques. In contrast to differences being the catalyzer, here impact graphs serve as input. The thus identified dependent information units are informed about changes of their environment by activating their *triggers*, i.e. graph rewriting rules specific to the adjustment of changes. Depending on the nature of the item, triggers activate the adjustment of the corresponding fragment or simply signal the user that he has to update them manually. Triggers are especially useful if the content of a document fragment can be automatically recomputed by inspecting its environment.

The facilities for change propagation form the top layer of our MoC cake and cover PoC 5.

Note 1.1. PoC 1 is left to the user and PoC 4 is covered by our MoC cake in general or rather our graph rewriting application in particular. Hence, both a change impact analysis and an adjustment are a combination of guided user intervention and automatic processing based on codified change patterns and propagation rules.

1.4 Objectives and Research Results in a Nutshell

An overwhelming amount of documents is produced and changed every day in nearly all areas of our every day life, such as, for instance, in business, in education, in research or in administration. A non-exhaustive list of examples are filled and signed forms in administration, research reports, test documents, or lecture notes, slides and exercise sheets in education, as well as requirements, documentations and software artifacts in development processes.

The documents, however, are seldom isolated artifacts but are intentionally related and intertwined with other documents altogether framing a collection of documents. Hence, changing a document requires possible adaptations to others within the collection. The fact that such document collections are heterogeneous, i.e. comprising documents of different types, draws the management even more complicating. To solve these issues, we pursue the following objectives.

Objectives

The goal of this research is to build a management of change methodology that

- embraces existing document types and
- allows for the declarative specification of annotation and propagation rules along classified interrelations inside and across documents of different types

in order to improve the maintainability of authoring processes, to optimize the release planning activity and thus to reduce the maintenance effort.

Thesis Statement

> *The research described in this thesis address the objectives by applying algorithmic dependency and change type analysis techniques on semi-structured documents to maintain structural and semantic relationships within and between documents as well as to propagate changes along these relations.*

Solution Strategy

We adapt and extend change management techniques from formal methods to the informal setting. Instead of a formal semantics we assume that these documents adopt syntactical and semantic structuring mechanisms formalized in a document model (consisting of an equivalence system, a semantic model, and an annotation model). This provides a notion of consistency and invariants that allows one to propagate effects of individual changes to entire document collections. Conversely, the document model provides means to localize effects of changes by introducing a notion for semantic dependencies between document parts. The detailed steps are:

- Analyze classes of semi-structured documents and explicate relationships between classified components in respective document models.
- Compose sets of graph rewrite rules to calculate on a semantic document impact graph the ripple effects of a proposed change depending on the types of the involved components as well as the types of the inter- and intra-relations.
- Propose a management of change model to describe the properties of a change impact analysis and adjustment process on semi-structured documents in general.
- Build a proof-of-concept tool to validate the algorithms.

This strategy not only permits evaluation of the consequences of planned changes but also allows trade-offs between suggested change approaches to be considered. Moreover, as we consider the term “document” to be on a very abstract level, our solution is also applicable in software engineering, a field of research change management has been investigated the most. We, however, consider a software system not only in terms of its source code. It consists of many other related items such as specification and design documentation. Our tool can accept information from design, specification documents or toolkits, as long as the information is detailed enough to provide the inputs needed by our algorithms.

Questions to be answered

- What are the impacts a set of proposed changes can bring to a document collection?
- How big is the closure of impact? If several alternative maintenance solutions are proposed to a system, which one is the "best" in terms of cost and efficiency?
- How will the different types of relationships in the system impact change propagation?
- How to set up an change impact analysis model to describe the problem and solution characteristics?

Research Results

The most common use of change management is to determine the ripple effect of a change before it has been made. Major results of this research include the understanding that the problem of change management depends on the ability to

- Create models of relationships among system objects.
- Capture these relationships in software and associated representations.
- Translate a specific change into the impacted objects and relationships.
- Trace relationships and reasonably bound the search for impacts.
- Retranslate the estimated impacted objects back into system objects.

We addressed the problem by analyzing in depth the relationships and types among the information units of semi-structured documents and applied graph rewriting techniques to compute impact graphs according to document-type specific dependency and change types among these components. Further, we developed the prototype tool *locutor* [99] to evaluate the algorithms and proposed a semi-automated adjustment policy to support users in implementing the identified modifications. In particular, our approach has the following characteristics:

- The original representation of the system (artifacts and dependencies) as developed and maintained in the environment is directly used for change management support so that there is no re-formulation required.
- In addition to artifact representations and dependencies, artifact properties and change patterns are also used for change impact analysis, which has greatly increased the flexibility and expressiveness of the approach.
- The use of the original system representation allows automated direct assistance to change propagation (i.e., actually carrying out the modifications) based on the impact analysis results and additional change propagation rules.
- Both change impact analysis and change propagation are a combination of automatic processing (based on change patterns or propagation rules) and guided user intervention.

These are in clear contrast to most existing approaches, where the impact analysis activities are usually performed based on an extracted system representation involving only artifact structures and dependencies (without their contents and types), and as such the actual change propagation process is not directly supported.

1.5 Organization of this Dissertation

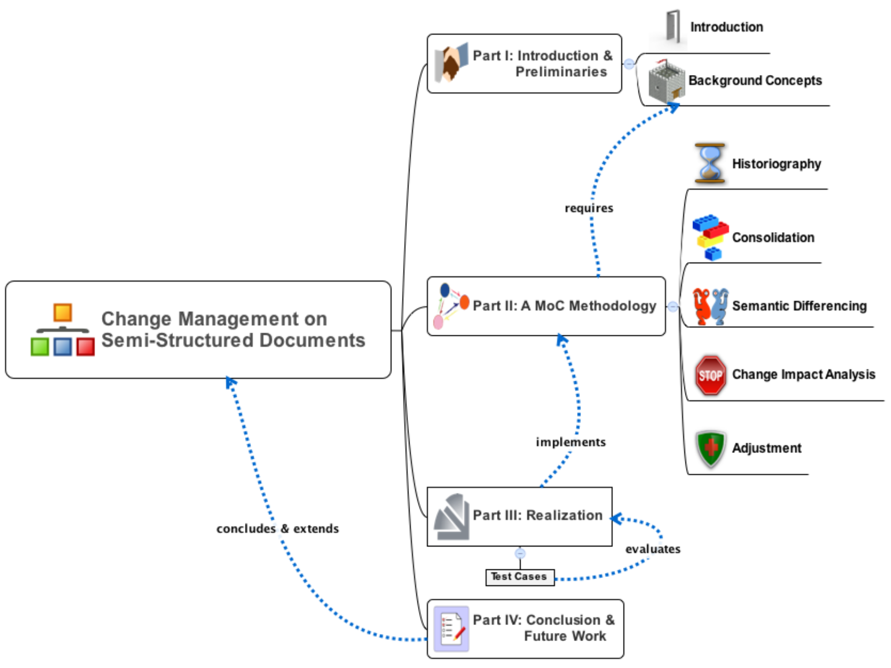


Figure 1.2: Relations between Chapters.

The introductory part (cf. Part I) sets the foundation for the parts reporting the details of this research. The first chapter (cf. Chap. 1) provides information on document maintenance, impact analysis and discusses the difficulties in impact analysis on collection of documents. It last defines the research scope and described briefly our research results. The preliminary chapter (cf. Chap. 2) introduces the background concepts necessary for full understanding of this dissertation. There are fundamental graph concepts introduced, file systems discussed and basic XML concepts addressed.

The second part (cf. Part II) of this work is on our management of change methodology. First, the concepts of version control are discussed and our fundamental data structure is introduced (cf. Chap. 3). In the following chapter a designated store for metadata harvesting is founded (cf. Chap. 4). Subsequently, our semantic difference analysis is discussed (cf. Chap. 5). The second part of this work is completed with the treatment of a change impact analysis (cf. Chap. 6) and adjustment (cf. Chap. 7) on heterogenous collections of documents.

The third part (cf. Part III) deals with the implementation. First, the *locutor* system cf. Chap. 8 is discussed followed by our evaluation description (cf. Chap. 9). The first section of the *locutor* system describes the system architecture (cf. Sec. 8.1). Second, the *locutor* command line client and its usage is discussed (cf. Sec. 8.2). Subsequently, we have a deeper look on the interfaces to our semantic difference analysis (cf. Sec. 8.3). Finally, the *locutor* core library is explained (cf. Sec. 8.4).

The last and fourth part of this work (cf. Part IV) summarizes the results and suggests avenues for further research.

Chapter 2

Preliminaries

This section describes the preliminaries and our utilized basic data structure, graphs, in particular, which are necessary for a full understanding of this dissertation.

Graphs are mathematical structures consisting of a set of nodes and a set of edges that connect these nodes to each other. Because graphs have an intuitive graphical representation, where nodes are represented by boxes and edges as lines between these nodes, they can be used to model virtually any possible structure.

Nowadays graphs are used in many applications, including computer networks, wireless / mobile networks, and car navigation systems. In this thesis, however, we need to understand document collections across file systems as the objects of change management and as both are tree-structured we represent them in a common foundation: *typed graphs* (cf. Sec. 2.1).

The theory of types originates from the early 1900s when Bertrand Russell wrote his Principles of Mathematics, containing what is currently well known as *Russel's paradox*. This paradox states that a set that contains all sets that do not contain themselves, should and should not contain itself at the same time and therefore can never exist. A solution to this problem constituted the first theory on types. Many definition for type systems have been introduced throughout history. Russell, for instance, defined a type as "*a range of significance for certain propositional functions*" [123], and Constable et al. as "*a collection of objects having similar structure*" [31]. For an unification of file systems and semi-structured documents, we follow in this work the view of Constable et al.

A further aspect of type systems we make our advantage of, is the important mechanism of *inheritance*, or *subtyping*, present in most current type systems. The basic idea of inheritance is that new types can be defined using already existing types. This new type is then a subtype of the already existing type and this existing type is a supertype of the new one. A subtype is a specialisation of its supertype and is considered more concrete. Inherent to inheritance is what Pierce calls the *principle of safe substitution*. According to this principle, any object of type S can be used safely in contexts where objects of type T are expected if S is a subtype of T [114]. In this work we avail ourselves of this principle by defining our change managment algorithms on the most generalized types, namely files (cf. Sec. 2.2), in order to seemlessly work across the file / file system boarder, i.e. to quasi work on semi-structured documents being specialized subtypes of ordinary files (cf. Sec. 2.3).

Continous changes on these static structures are described using graph manipulations. In this thesis, we define such manipulations on graphs using graph transformations, which provide a formal yet intuitive way to specify the manipulation of graphs based on rules.

2.1 Basic Concepts

Let $(\underline{\mathbb{T}}, <:)$ be a partially ordered set of types and $\underline{\mathbb{V}} \subseteq \underline{\mathbb{T}}$ be a set of **node types** and $\underline{\mathbb{E}} \subseteq \underline{\mathbb{T}}$ a set of **edge types**.

A $(\underline{\mathbb{V}}, \underline{\mathbb{E}})$**-typed pre-graph** is a tuple $\langle \mathbb{V}, \mathbb{E}, \prec \rangle$ where $\mathbb{V} = \biguplus_{\underline{\mathbb{v}} \in \underline{\mathbb{V}}} \mathbb{V}_{\underline{\mathbb{v}}}$ is the disjoint union of sets of nodes of type $\underline{\mathbb{v}}$, $\mathbb{E} = \biguplus_{\underline{\mathbb{e}} \in \underline{\mathbb{E}}} \mathbb{E}_{\underline{\mathbb{e}}}$ is the disjoint union of sets of edges of type $\underline{\mathbb{e}}$ and $\prec$ is a partial order on $\mathbb{E}$. We denote empty typed pre-graphs by $\bullet$.

Let $\mathcal{G} = \langle \mathbb{V}, \mathbb{E}, \prec \rangle$ be a pre-graph. The **size** of $\mathcal{G}$ is denoted by $|\mathcal{G}|$, is $|\mathbb{V}|$.

An edge $v \rightarrowtail v' \in \mathbb{E}$ is said to **connect** nodes v and v'. We also say that $v \rightarrowtail v'$ **leaves** v and **enters** v'. An edge $v \rightarrowtail v'$ is **incoming** for v' and **outgoing** for v. For a node $v \in \mathbb{V}$, we denote the set of its incoming edges by $\eta^+(v)$ and the set of its outgoing edges by $\eta^-(v)$. We will always assume that $\mathbb{E}$ has no edges of the form $u \rightarrowtail u$, i.e. the pre-graph $\mathcal{G}$ is **simple**.

A **walk** in $\mathcal{G}$ is a sequence $\langle v_1, \ldots, v_k \rangle$ of nodes in $\mathbb{V}$, such that $\mathbb{E}$ contains edges $v_i \rightarrowtail v_{i+1}$ for all $i = 1, \ldots, k-1$. The **length** of the walk $\langle v_1, \ldots, v_k \rangle$ is k. A walk is called a **path** if all $v_1, \ldots, v_k$ are distinct. It is called a **cycle** if $v_1 = v_k$. A node $v' \in \mathbb{V}$ is called **reachable** from a node $v \in \mathbb{V}$ if there is a path from v to v'.

A typed pre-graph $\mathcal{G}' = (\mathbb{V}', \mathbb{E}', \prec)$ is called a **subgraph** of $\mathcal{G} = (\mathbb{V}, \mathbb{E}, \prec)$ if $\mathbb{V}' \subseteq \mathbb{V}$ and $\mathbb{E}' \subseteq \mathbb{E}$, short $\mathcal{G}' \sqsubseteq \mathcal{G}$.

A typed pre-graph $\mathcal{G} = \langle \mathbb{V}, \mathbb{E}, \prec \rangle$ is a **typed graph** if for each $v \rightarrowtail v' \in \mathbb{E}$ it holds $v, v' \in \mathbb{V}$. The class of all $(\underline{\mathbb{V}}, \underline{\mathbb{E}})$-typed pre-graphs (resp. graphs) is denoted by $\mathbb{G}^{\underline{\mathbb{E}}}_{\underline{\mathbb{V}}}$ (resp. $\ddot{\mathbb{G}}^{\underline{\mathbb{E}}}_{\underline{\mathbb{V}}}$) or simply $\mathbb{G}$ (resp. $\ddot{\mathbb{G}}$) if $\underline{\mathbb{V}}$ and $\underline{\mathbb{E}}$ clear from context. For a node v we denote its type by *type*(v) and if *type*$(v) = \underline{\mathbb{v}}$ for some $\underline{\mathbb{v}} \in \underline{\mathbb{V}}$ we write $v : \underline{\mathbb{v}}$. Similarly for edges.

Let $\mathcal{G} = (\mathbb{V}, \mathbb{E}, \prec)$ be a typed graph. If $\mathbb{V}' \subseteq \mathbb{V}$, then the typed graph $\mathcal{G}' = (\mathbb{V}', \mathbb{E}', \prec)$ with $\mathbb{E}' = \{v \rightarrowtail v' \in \mathbb{E} \mid v \in \mathbb{V}', v' \in \mathbb{V}'\}$ is called a **subgraph induced by** $\mathbb{V}'$. If $\underline{\mathbb{V}}' \subseteq \underline{\mathbb{V}}$ and $\underline{\mathbb{E}}' \subseteq \underline{\mathbb{E}}$, then the **restriction** of $\mathcal{G}$ on $\underline{\mathbb{V}}'$ and $\underline{\mathbb{E}}'$ is defined by

$$\mathcal{G}_{|\underline{\mathbb{V}}', \underline{\mathbb{E}}'} = (\mathbb{V}, \mathbb{E}, \prec)_{|\underline{\mathbb{V}}', \underline{\mathbb{E}}'} := (\biguplus_{\underline{\mathbb{v}} \in \underline{\mathbb{V}}'} \mathbb{V}_{\underline{\mathbb{v}}}, \biguplus_{\underline{\mathbb{e}} \in \underline{\mathbb{E}}'} \mathbb{E}_{\underline{\mathbb{e}}}, \prec_{|\biguplus_{\underline{\mathbb{e}} \in \underline{\mathbb{E}}'} \mathbb{E}_{\underline{\mathbb{e}}}})$$

where $\prec_{|\mathbb{L}} := \{l \prec l' \mid l, l' \in \mathbb{L}\}$, for every set of edges $\mathbb{L}$.

A typed graph $(\mathbb{V}, \mathbb{E}, \prec)$ is a **typed tree** if there exists a unique node r such that for every node v different from r there exists exactly one path from r to v and $\prec_{|\{v \rightarrowtail v' \in \mathbb{E}\}}$ is a total order for every node $v \in \mathbb{V}$. We denote $\Uparrow\mathcal{G}$ as the **root node** of the typed tree $\mathcal{G}$ and in the following we use the terms "tree" and "typed tree" interchangeably.

Let $\mathcal{G} = (\mathbb{V}, \mathbb{E}, \prec)$ be a typed tree. If $v \rightarrowtail v' \in \mathbb{E}$, then v is called a **parent** of v' and v' is a **child** of v, respectively. Nodes which do not have children are called **leaves**. For a node v, we denote its parent by *parent*(v), the sequence of its children by *children*(v). Via the total order $\prec$ on $\eta^-(v)$ for all $v \in \mathbb{V}$ the children of v can be represented as a sequence $\langle v_1, \ldots, v_k \rangle$ where v_1 is the $\prec$-minimal (left-most) child and v_k is the $\prec$-maximal (right-most) child.

Nodes $v, v' \in \mathbb{V}$ are called **siblings** if they share a parent. If $\langle v_1, \ldots, v_k \rangle$ is the sequence of children of a node v, we term $\langle v_1, \ldots, v_{i-1} \rangle$ the left siblings of v_i and $\langle v_{i+1}, \ldots, v_k \rangle$ the right siblings, $i = 1, \ldots, k$. The node v_{i-1} is called the direct left sibling of v_i, while v_i is the direct right sibling for v_{i-1}, $i = 2, \ldots, k$.

A node $v \in \mathbb{V}$ is called an **ancestor** of a node $v' \in \mathbb{V}$, if $v = v$ or there is a path from v to v'. In this case, v' is called a **descendant** of v. Note that every node is both an ancestor and a descendant of itself and $\Uparrow\mathcal{G}$ is an ancestor for all nodes in $\mathcal{G}$.

The set of all descendants of a node $v \in \mathbb{V}$ (including v) induces also a typed tree with root v. This tree is called a **subtree** of $\mathcal{G}$ rooted at v and denoted by $\mathcal{G}|_v$. In addition we use the notation $\mathcal{G}||_v$ to denote the set of trees induced by the children of v.

A typed graph $\mathcal{G}$ is a **collection of typed trees** if there exists a partitioning of $\mathcal{G}$ into non-overlapping typed graphs $\mathcal{G}_1,\ldots,\mathcal{G}_n$ (i.e. $\mathcal{G}=\mathcal{G}_1 \uplus \ldots \uplus \mathcal{G}_n$) and each $\mathcal{G}_i$ is a typed tree.

2.2 File Systems

In computing, a file system is a method of storing data in an organized structure and to retrieve it when requested. File system types can be classified into disk file systems, network file systems and special purpose file systems. Among the various disk file systems, we employ the Unix file system (UFS) used by many Unix and Unix-like operating systems. Everything in UFS is a file in sense of a collection of data. The UFS contains several different types of files[1]:

Ordinary Files are used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with. These file types are always located within/under a directory file and do not contain other files.

Directories build the branching points in the hierarchical tree of a UFS. These are used to organize groups of files, may contain ordinary files, special files or other directories but never contain "real" information which you would work with (such as text). Basically, these file types are just used for organizing files. All files are descendants of the root directory, located at the top of the tree.

Links are a system to make a file or directory visible in multiple parts of the system's file tree.

Each "file" has at least a location within the file system, a name and other attributes. This information is actually stored in an index node (inode), the basic file system entry. One of the most important characteristics of UFS for us here is its hierarchical structure. All of the files in a Unix file system are organized into a multi-leveled hierarchy called a directory tree. At the very top of the file system is a single directory called `root` which is represented by a / (slash). All other files are descendants of root. At the first glance, a directory tree is a tree but a symbolic link may turn the logical structure of UFS into a cyclic graph. We will see, however, that with our fundamental data structure we are able to model Unix file systems in a tree structure without loss of data but gain in simplicity.

Therefore we introduce **solid node types** and **solid edge types** $(\underline{\mathbb{V}}_{\text{sld}}, \underline{\mathbb{E}}_{\text{sld}})$ where $\underline{\mathbb{V}}_{\text{sld}}$ comprises the type (1) $\underline{\mathbb{f}}$ for ordinary files, (2) $\underline{\mathbb{s}} <: \underline{\mathbb{f}}$ for links, and (3) $\underline{\mathbb{d}} <: \underline{\mathbb{f}}$ for directories . The solid edge types $\underline{\mathbb{E}}_{\text{sld}}$ contains the type $\underline{\Supset}$ for edges leaving a node of type $\underline{\mathbb{d}}$ and entering a node type $\underline{\mathbb{f}}$, i.e. $u : \underline{\mathbb{d}} \rightarrowtail v : \underline{\mathbb{f}}$.

In the following when we refer to file systems we abstract on all UFS compliant systems and convertible ones, respectively. In case of MS Windows systems, for example, one constructs an additional imaginary root node.

2.3 XML

The Extensible Markup Language (XML) [19] is a simple, very flexible text format for documents containing structured information. It is derived from the Standard Generalized Markup Language (SGML) [51].

[1]At this point we neglect both processes, executable programs identified by unique process identifier, and special files used to represent physical devices.

XML enables developers to define XML instances by providing a grammar for their special purposes. Such a grammar definition is basically known as a *document type definition* (DTD)[2]. A DTD defines the legal building blocks of an XML document, i.e. a list of legal *element* names (*aka.* label) and *attribute* names. Each element has an optional associated set of attributes. An attribute has a name and a string-value. Character data is grouped into specific *text* elements.

XML allows developers to mix elements from different document type definitions. In such combined documents it is likely to find the same element name used for two (or even more) different things. *Namespaces* disambiguate these instances by associating a Uniform Resource Identifier (URI) with each element set, and by attaching a prefix (just a string) to each element to indicate which set it belongs to. According to [25], an element name, comprising an optional namespace prefix and a local name, is called a *qualified name* (QName). A QName can be converted into an *expanded* QName by resolving its namespace prefix to the respective namespace URI. For further information on XML terminology we refer to [19].

Document Characteristics. There are three general correctness criteria for XML documents. Based on these criteria, XML documents can be well-formed, valid, or broken. As described in [52], a **well-formed** XML document must meet the following requirements:

(R1) The document must contain a single root element.

(R2) Every element must be correctly nested.

(R3) Each attribute can have only one value.

(R4) All attribute values must be enclosed in single/double quotation marks.

(R5) Elements must have begin and end tags, unless they are empty elements.

(R6) Empty elements are denoted by a single tag ending with a slash (`/`).

(R7) Isolated markup characters are not allowed in content. The special characters <, `&`, and > are represented as `&gt`, `&amp`, `&lt` in content sections.

(R8) A single (double) quotation mark is represented as `&apos` (`&quot`) in content sections.

(R9) The sequence `<[[` and `]]>` cannot be used.

(R10) If a document does not have a DTD, the values for all attributes must be of type CDATA by default.

A **valid** XML document is a well-formed XML document, which also conforms to associated grammar rules, like in a DTD. A **broken** XML document violates either well-formedness or validness.

XML documents are ordered [25]. Therefore, in combination with R1 and R2 we consider an XML document to be a tree.

Note an XML *root node* is the node representing the document itself, whereas an XML *document element* is the element node representing the element that has all other elements within the document as children. The separation of concepts is essential, because the root node can have other children besides the document element, e.g. comments and processing instructions.

[2]There are several alternatives to DTD. The most popular ones are: (1) XML Schema [142] and (2) RelaxNG [140] .

For XML tree construction we follow the XPath data model [25]. Regarding typing we introduce **syntactic node types** and **syntactic edge types** ($\underline{\mathbb{V}}_{\text{syn}}, \underline{\mathbb{E}}_{\text{syn}}$). The latter comprises the type $\underline{ↄ} <: \underline{\ni}$. With respect to node typing we consider labels of XML reference statements, like XInclude [87], as subtypes of $\underline{\mathbb{s}}$, labels of empty elements as subtypes of $\underline{\mathbb{f}}$, and XML text nodes, comments and processing instructions are of type $\underline{\mathbb{t}} <: \underline{\mathbb{f}}$, $\underline{\mathbb{c}} <: \underline{\mathbb{f}}$, and $\underline{\mathbb{p}} <: \underline{\mathbb{f}}$, respectively. Otherwise element types are subtypes of $\underline{\mathbb{d}}$. To attribute nodes a special treatment is given in our fundamental data structure. We call an XML tree **simple**, if it exclusively contains nodes of type element and text (no attributes).

Further in this work we restrict our consideration to valid XML documents only and use the terms “XML document” and “XML tree” interchangeably.

2.4 Notation Overview

Kind	Symbols	Description
Sets	$\mathbb{\Sigma}$	An alphabet, a set of symbols or letters, e.g. characters or digits (cf. page 37).
	$\mathbb{\Delta}$	A parametrized set of transformation functions δ_v for each node $v \in \mathbb{V}(\mathscr{T})$. (cf. page 43)
	Δ	An edit script (cf. page 91).
	$\mathbb{B}$	The set of booleans (cf. page 85).
	$\mathbb{D}$	A data set comprising all of $\mathbb{\Sigma}^*$, $\mathbb{N}$ and closed under n-tuple, set construction, and function definition (cf. page 37).
	$\mathbb{E}$	A set of edges (cf. page 22).
	$\underline{\mathbb{E}}$	A set of edge types (cf. page 22).
	$\underline{\mathbb{E}}_{\text{sld}}$	The set of solid edge types (cf. page 23).
	$\underline{\mathbb{E}}_{\text{syn}}$	The set of syntactic edge types (cf. page 25).
	$\mathbb{FS}$	The set of all *fs*-trees (cf. page 37).
	$\mathbb{FS}^+$	The set of all *fsp*-trees (cf page 45).
	$\mathbb{G}_{\underline{\mathbb{V}}}^{\underline{\mathbb{E}}}$	The class of all ($\underline{\mathbb{V}}, \underline{\mathbb{E}}$)-typed pre-graphs (cf. page 22).
	$\ddot{\mathbb{G}}_{\underline{\mathbb{V}}}^{\underline{\mathbb{E}}}$	The class of all ($\underline{\mathbb{V}}, \underline{\mathbb{E}}$)-typed graphs (cf. page 22).
	$\mathbb{N}$	The set of natural numbers (cf. page 37).
	$\mathbb{FS}_{\mathscr{D}}^+$	A set of documents conforming to some $\mathscr{D}$ (cf. page 81).
	$\mathbb{P}$	A set of property names (cf. page 45).
	$\mathbb{Q}$	The set of equivalence systems (cf. page 81).
	$\mathbb{R}$	The set of repositories (cf. page 42).
	$\mathbb{REG}_{\mathscr{T}}$	A *locutor* metadata registry on some *fsp*-tree $\mathscr{T}$ (cf. page 64).
	$\mathbb{V}$	A set of nodes (cf. page 22).
	$\underline{\mathbb{V}}$	A set of node types (cf. page 22).
	$\underline{\mathbb{V}}_{\text{imp}}$	A set of impact node types (cf. page 99).
	$\underline{\mathbb{V}}_{\text{sem}}$	A set of semantic node types (cf. page 99).
	$\underline{\mathbb{V}}_{\text{sld}}$	The set of solid node types (cf. page 23).
	$\underline{\mathbb{V}}_{\text{syn}}$	A set of syntactic node types (cf. page 25).

Continued on next page

Notation Overview – continued from previous page

Kind	Symbols	Description
	$\mathbb{W}_{\mathscr{T}}$	The set of all working copies in some *fs*-tree $\mathscr{T}$ (cf. page 43).
	$\mathbb{X}_{\mathscr{T}}$	The set of all externals in some *fs*-tree $\mathscr{T}$ (cf. page 46).
Structures	$\bullet$	An empty graph (cf. page 22).
	$\mathscr{A}_{\mathscr{S}}$	An annotation model for some semantic model $\mathscr{S}_{\mathscr{D}}$ (cf. page 103).
	$\mathscr{D}$	A document type specification (cf. page 81).
	$\overleftarrow{\mathcal{D}}$	An SDI graph (cf. page 99).
	$\mathscr{E}_{\mathscr{D}}$	An equivalence system on some $\mathbb{FS}^{+}_{\mathscr{D}}$ (cf. page 81).
	$\mathscr{I}$	An interaction model between some document models $\mathscr{M}_1,\ldots,\mathscr{M}_n$ (cf. page 109).
	$\mathcal{I}$	The impacts sub-graph of some SDI graph $\overleftarrow{\mathcal{D}} = \langle \mathcal{O}, \mathcal{S}, \mathcal{I} \rangle$ (cf. page 100).
	$\mathscr{L}, \mathscr{M}, \mathscr{R}, \mathscr{T}$	*fs*-trees and *fsp*-trees, respectively.
	$\mathscr{M}_{\mathscr{D}}$	A document model for some $\mathscr{D}$ (cf. page 104).
	$\mathcal{O}$	The document sub-graph of some SDI graph $\overleftarrow{\mathcal{D}} = \langle \mathcal{O}, \mathcal{S}, \mathcal{I} \rangle$ (cf. page 100).
	$\mathscr{P}_{\mathscr{A}_{\mathscr{S}_{\mathscr{D}}}}$	An adjustment model for some annotation model $\mathscr{A}_{\mathscr{S}_{\mathscr{D}}}$ (cf. page 114).
	$\mathfrak{R}$	A *locutor* metadata registry (cf. page 65).
	$\mathcal{S}$	The semantics sub-graph of some SDI graph $\overleftarrow{\mathcal{D}} = \langle \mathcal{O}, \mathcal{S}, \mathcal{I} \rangle$ (cf. page 100).
	$\mathscr{S}_{\mathscr{D}}$	A semantic model for some $\mathscr{D}$ (cf. page 101).
	w, m, n a, b, c, d, e, f	Working copies.
	$w\vert_e$	A working copy induced by an external e (cf. page 46).
Operators	$\prec$	A partial order on a set of edges $\mathbb{E}$ (cf. page 22).
	$\rightarrowtail$	An edge within a set of edges $\mathbb{E}$ (cf. page 22).
	η^{+}	A set of incoming edges of some node (cf. page 22).
	η^{-}	A set of outgoing edges of some node (cf. page 22).
	$\sqsubseteq$	A partial order on a set of graphs (cf. page 22).
	$\gg$	A redundancy relation on a set of working copies (cf. page 51).
	$\xleftarrow{-}$	A dependency relation on a set of working copies (cf. page 62).
	$\Uparrow\mathcal{G}$	The root node of a typed tree $\mathcal{G}$ (cf. page 22).
	$\mathcal{G}\vert_u$	The subgraph of some graph $\mathcal{G}$ rooted at node u (cf. page 22).
	$\mathcal{G}\Vert_u$	The set of subgraphs induced by the children of u (cf. page 22).
	$\mathcal{G}_{\vert\underline{\mathbb{V}},\underline{\mathbb{E}}}$	A restriction of $\mathcal{G}$ on $\underline{\mathbb{V}}$ and $\underline{\mathbb{E}}$ (cf. page 22).
	l	A *fs*-tree edge labeling (cf. page 37).

Continued on next page

Notation Overview – continued from previous page

Kind	Symbols	Description
	$\#$	A strict *fs*-tree node decoding (cf. page 37).
	▪	A *fs*-tree lookup function (cf. page 41).
	$\varepsilon, \pi, \pi_1, \ldots$	*fs*-tree paths.
	γ	A *fs*-tree value function (cf. page 41).
	ϱ	A repository funciton (cf. page 42).
	ω	A repositroy mapping (cf. page 42).
	μ^{ω}	A correspondence function (cf. page 42).
	δ	A transformation (patch) function (cf. page 43).
	β	A *fsp*-tree property funciton (cf. page 45).
	ξ	An externals definition (cf. page 46).
	$\equiv$	A congruence on $\mathbb{FS}^{+}_{\mathscr{D}}$ (cf. page 81).
	o	A predicate on $\mathbb{FS}^{+}_{\mathscr{D}}$ (cf. page 81).
	μ	A transformation of treeish/ concise edit scripts (cf. page 91).
	$\widetilde{\mu}$	A semantically minimization of ordinary edit scripts (cf. page 91).
	α	An abstraction graph transformation (cf. page 103).
	σ	A propagation graph transformation (cf. page 103).
	ι	A projection graph transformatoin (cf. page 103).
	ψ	An adjustment graph transformation (cf. page 114).
	$\hookleftarrow$	A propagation and projection graph transformation (cf. page 129).
	parent	The parent of some node (cf. page 22).
	children	The sequence of children of some node (cf. page 22).
	type	The type of some node (cf. page 22).
	resolve	A redundancy resolution (cf. page 52).
Instances	$\underline{\mathbb{f}}$	The solid node type for ordinary files (cf. page 23).
	$\underline{\mathbb{s}}$	The solid node type for links (cf. page 23).
	$\underline{\mathbb{d}}$	The solid node type for directories (cf. page 23).
	$\underline{\ni}$	The solid edge type for edges leaving a node of type $\underline{\mathbb{d}}$ (cf. page 23).
	$\underline{\text{ɔ}}$	The syntactic edge type for edges representing a XML *child-of* relation (cf. page 25).

Part II

A Management of Change Methodology

Chapter 3

Historiography

We must use time as a tool, not as a couch.

— John F. Kennedy

An important element of change management is version control. It means tracking the history, i.e., the different versions of documents and it plays a major role in ensuring the quality of delivered systems. Version control also ensures that documents are not degraded by uncontrolled or unapproved changes, and provides an essential audit facility.

3.1 Introduction

Writing technical papers for journals as well as for the end-user is one of the most important tasks in day-to-day business. Many articles are co-authored by multiple authors, frequently also by authors that are not even member of the same institution. When multiple people are working on a project together, it becomes increasingly difficult to keep up to date information about the project and apply the changes of the individuals in a central version. Starting at two or three people we are more often employed in reconciling the various files and changes, as to do what we really want to do: work.

Typically, a paper goes through numerous states of editing during the writing process, requiring that changes to the document are shared between authors. Many publications, at least in the field of engineering and computer science, are written using the LaTeX typesetting system, but some authors prefer conventional What You See Is What You Get (WYSIWYG) word processors. Apart from the text, technical papers typically include illustrations, tables, and bibliographic information. These illustrations and the respective raw-data should also be stored together with the data of the document. An important observation is also that most authors love to work with their own, optimized workflow and IT infrastructure. They are generally reluctant to change this workflow, unless given a very convincing reason. Hence, assuming that everybody is agreeing on the same operating system, text editor or graphics program is unrealistic.

In summary, a system for collaborative editing should satisfy the following requirements [20]: (1) provide *access* to documents for all authors, (2) allow for *collaboration* with (in particular external) co-authors, (3) provide *versioning* of documents and support *sharing changes*, (4) store additional data (figures, tables, bibliographic information), and (5) leave authors maximum freedom to chose their optimal workflow (operating system, text editor, graphics editor), in short, be *minimally invasive* .

Surprisingly, no ready-made solutions have emerged that are specifically tailored to this task. Readers familiar with version control systems will notice, though, that using a version control system can solve many of the requirements presented above. However, the extremely important point for collaboration, the support of sharing changes and the rippling of changes is only poorly or not at all provided. Therefore we herein described our development through to a sophisticated change management system closing these gaps. In fact, our solution is based on SUBVERSION, but extends the support of sharing changes by change management within the meaning of change impact analysis (cf. Chap. 6) and adjustment (cf. Chap. 7). But before we get to that, let us first examine our foundation, version control systems, a bit further.

3.1.1 Why use Version Control?

There are a number of reasons why to use an automated version control system for a project. It will track the history and evolution of a project, so you do not have to. For every change, there will be a log of *who* made it; *why* it was made — depending on the level of detail of the log message; *when* it was made; and *what* the change was. Version control software makes it easier to collaborate. For example, when people more or less simultaneously make potentially incompatible changes, the software helps in identification and manually resolving those conflicts. It can help to recover from mistakes. If one project member makes a change that later turns out to be in error, an earlier version of one or more files can be reverted. Version control systems, therefore, help to efficiently figure out exactly when a problem was introduced and help to work simultaneously on, and manage the drift between, multiple versions of a project.

It is also important to note, that storing projects in a version control system is essential when multiple authors are editing papers concurrently. But putting projects under version control makes sense, even if you are the sole author. In the author's experience, the general benefits of version control, such as archiving and restoring of different versions for all files, documentation of changes and painless synchronization of data between multiple computers, pay off quickly, even in a single user environment. Thus our MoC cake is not only useful in a multi-author collaborative editing process, but can be used equally well for a single author.

3.1.2 A Short History of Version Control

Version control is a diverse field, so much so that it is referred to by many names and acronyms. Here are a few of the more common variations you will encounter in literature: (1) revision control, (2) software configuration management or configuration management (3) source code management, and (4) source code control or source control. Some people claim that these terms actually have different meanings, but in practice they overlap so much that there is no agreed or even useful way to tease them apart. Therefore, here we use the term "version control" and "version control system", respectively, as an abstraction of all these variations.

The process of version control is the process of managing multiple versions of a piece of information. In its simplest form, this is something that many people do by hand: every time you modify a file, save it under a new name that contains a number, each one higher than the number of the preceding version.

Manually managing multiple versions of even a single file is an error-prone task, though, software tools to help automate this process have long been available. The earliest automated vision control tools were intended to help a single user to manage versions of a single file. Over the past few decades, the scope of vision control tools has expanded greatly: they now manage multiple files, and help multiple people to work together. The best modern version control tools have no problem coping with thousands of people working together on projects that consist of hundreds of thousands of files.

The best known of the old-time version control tools is the Source Code Control System (SCCS) [119], which Marc Rochkind wrote at Bell Labs, in the early 1970s. SCCS operated on individual files and required every person working on a project to have access to a shared workspace on a single system. Only one person could modify a file at any time; arbitration for access to files was via locks. It was common for people to lock files, and later forget to unlock them, preventing anyone else from modifying those files without the help of an administrator.

Walter Tichy developed a free alternative to SCCS in the early 1980s; he called his program Revision Control System (RCS) [138]. Like SCCS, RCS required developers to work in a single shared workspace, and to lock files to prevent multiple people from modifying them simultaneously.

Later in the 1980s, Dick Grune used RCS as a building block for a set of shell scripts he initially called `cmt` but then renamed to Concurrent Versions System (CVS). The big innovation of CVS was that it let developers work simultaneously and somewhat independently in their own personal workspaces. The personal workspaces prevented developers from stepping on each other's toes all the time, as it was common with SCCS and RCS. Each developer had a copy of every project file and could modify their copies independently. They had to merge their edits prior to committing changes to the central repository.

Brian Berliner took Grune's original scripts and rewrote them in C, releasing in 1989 the code that has since developed into the modern version of CVS [48]. CVS subsequently acquired the ability to operate over a network connection, giving it a client/server architecture. CVS's architecture is centralized; only the server has a copy of the history of the project. Client workspaces just contain copies of recent versions of the project's files and a little metadata to tell them where the server is. CVS has been enormously successful; it is probably the world's most widely used version control system.

In the early 1990s, Sun Microsystems developed an early distributed version control system, called TeamWare. A TeamWare workspace contains a complete copy of the project's history. TeamWare has no notion of a central repository (CVS relied upon RCS for its history storage; TeamWare used SCCS).

In the mid-late 1990s, IBM developed the Configuration Management Version Control (CMVC) system, a software package that serves as an object repository, and performs software version control, configuration management, and change management functions. It integrated source code control, bug tracking, and build coordination in a distributed environment. CMVC sales and support terminated some time after IBM acquired Rational Software, its functions being superseded by products in the Rational product line. However, some customer installations of CMVC remain in use as of 2008 and it is still widely used within IBM.

As the 1990s progressed, awareness grew of a number of problems with CVS. It records simultaneous changes to multiple files individually, instead of grouping them together as a single logically atomic operation. It does not manage its file hierarchy well; it is easy to make a mess of a repository by renaming files and directories. Even worse, its source code is difficult to read and to maintain, which made the "pain level" of fixing these architectural problems prohibitive.

In 2001, Jim Blandy and Karl Fogel, two developers who had worked on CVS, started a project to replace it with a tool that would have a better architecture and cleaner code. The result, Subversion, does not stray from CVS's centralized client/server model but it adds multi-file atomic commits, better namespace management, and a number of other features that make it a generally better tool than CVS. Since its initial release, it has rapidly grown in popularity.

More or less at the same time, Graydon Hoare began working on an ambitious distributed version control system that he named MONOTONE [91]. While MONOTONE addresses many of CVS's design flaws and has a peer-to-peer architecture, it goes beyond earlier (and subsequent) revision control tools in a number of innovative ways. It uses cryptographic hashes as identifiers and has an integral notion of "trust" for code from different sources.

In April 2005, MONOTONE became the subject of increased interest in the Free/Libre/Open Source Software community (FLOSS) as a possible replacement for BitKeeper (followup of TEAMWARE) in the Linux development process. Instead of adopting MONOTONE, Linus Torvalds wrote his own SCM system, GIT. GIT falls in the category of distributed version control management systems. It's design uses some ideas from MONOTONE, but the two projects do not share any core source code. GIT is a popular version control system designed to handle very large projects with speed and efficiency. It is used mainly for various open source projects, most notably the Linux kernel [50]. It is currently maintained by Junio C. Hamano.

SVK is a decentralized version control system built with the robust SUBVERSION file system. It was originally developed by Kao Chia-Liang since his sabbatical year in 2003. The system has one special characteristic: SVK is able to mirror file system trees of different version control systems, including SUBVERSION, and define working copies on these. So one can mirror, for example, a SUBVERSION repository of any project, create an individual local branch, and work on this branch obtaining full local version control management. Once finished the branch, one may send a commit to the origin repository or inquire a *diff*-script and send this to the respective developer. Similar to GIT, every SVK working directory is a full-fledged repository with full revision tracking capabilities, not dependent on network access or a central server [130].

In April 2003 David Roundy developed DARCS a distributed revision control system, along the lines of CVS. That means that it keeps track of various revisions and branches of a project, allows for changes to propagate from one branch to another. DARCS is intended to be an "advanced" revision control system. DARCS has two particularly distinctive features which differ from other revision control systems: (1) each copy of the source is a fully functional branch and (2) underlying DARCS is a consistent and powerful theory of patches [33].

MERCURIAL began life in 2005. While a few aspects of its design are influenced by MONOTONE, MERCURIAL focuses on ease of use, high performance, and scalability to very large projects. MERCURIAL is a fast, lightweight source control management system designed for easy and efficient handling of very large distributed projects. The differences to a "traditional" centralized VCS (like CVS or SUBVERSION) include the fact that users get a self-contained repository that they can commit changes to, even when being offline and changes can be pushed back to the parent repository or even to other repositories. MERCURIAL is also called `hg` for short.

This survey shows the initial manifestation that version control systems can solve many of the requirements presented at the end of Sec. 3.1, but all here listed systems largely neglect relations between and within documents as well as between working copies and repositories, respectively. These are mandatory information for change management, though! Thus, not surprisingly, these systems do not have notions of change impact analysis nor adjustment. In addition they do not solve the nested working copy problem per se (cf. Sec. 3.2.4), another important point regarding consistency and maintenance in multi-author multi-projects environments. Working copies and repositories are regarded as self-contained units, although metadata describing various interrelations are present altogether. For example, SUBVERSION as well as GIT store information regarding versioned file system entries in designated directories like `.svn` and `.git`. However, tools like SVNX [134] sparsely utilize these metadata, but to a non satisfying extend, i.e., the information is exclusively used on single working copies. Metadata on relations in between are again neglected.

These weaknesses hinder collaborative authoring process immensely, for example, think of sharing changes between dependent working copies. In what form we use such metadata to realize our change management solution, will be discussed further in Chap. 4. It should be noted, however, that our approach is not limited to the use of SUBVERSION but, quite the contrary, takes a general validity for the mentioned version control systems.

This claim to generality requires a deeper understanding of the underlying data structures both on the coarse-grained level, e.g., relations between working copies/repositories, as well as on the fine-grained level, e.g., relations between directories/files/documents. With our abstraction on these underlying data structures we are able to extend the usage of metadata across working copy-/repository and file system/file borders. Our fundamental data structure is the basis for a innovative document change management, i.e., for a sophisticated intra- and inter-relation management on documents.

3.2 The Fundamental Data Structure

A version control system can be characterized as a system which tracks the history of file and directory changes over time. All version controlled files and directories reside as a tree structure in a system distinct repository and both can have version controlled metadata. Using the aforementioned journaling functionality, changes to such a tree are transactional resulting in a new snapshot (*aka.* revision) of the whole tree including all recent changes made in that commit operation as well as all previous unchanged data.

In case of our reference version control system, SUBVERSION, the repository does not actually keep the whole contents in every revision. For every n-th revision it uses smart mechanisms to store only the differences that were made in that n-th revision itself (i.e. data differences between revisions $n-1$ and n). So, revisions allow us to retrieve any version of a file or directory or even of the whole tree (since a single revision is created for the whole repository tree) at any time, no changes will be lost. Each revision is a permanent snapshot of a tree, i.e. if an item is added to a SUBVERSION repository it can not be removed from the repository entirely because it can be always found in those revisions where it was added and changed.

For this reason version control systems constitute the corner stone of our change management, the base of our MoC cake: a VCS ensures transactional recording of details of the changes for future reference and releases the changed item(s) back to the corpora, i.e. the collection of documents under version control.

The following targets the terms such as tree structure, repository, and working copies more precisely through to formalization, in order to get a better grasp of these for our further development.

3.2.1 *fs*-trees

A common scenario of using a SUBVERSION repository is working with a copy of any sub-tree of a repository tree on a local computer and publishing results of this work into the repository. In other words, a client somehow changes files and directories taken from a repository and then commits his changes into the repository. A local version controlled data tree is called *working copy* (WC).

The core of most version control systems is a central *repository*, which stores information in form of a file system tree. The question for these systems is how to allow users to share information, but prevent them from accidentally stepping on each other's feet? The most widely used solution is the *copy-modify-merge* model [28], where each user's client creates a private working copy, i.e. a local reflection of the repository's files and directories. Users can then work simultaneously and independently, modifying their working copies. The private copies are periodically committed to the repository, allowing other users to merge changes into their working copy to keep it synchronized.

We are interested in version control for large and complex projects where it is often useful to construct a working copy that is made out of a number of different projects. For example, one may want different sub-directories to come from different locations in a repository, or from different repositories altogether. Most version control systems provide support for this via *externals definitions*.

However, this model only partially applies, though. Version control systems (as well as XML databases) neglect both ways in which documents are organized: *logic-internally*, i.e., by their (highly) hierarchical internal structure and *logic-externally*, i.e., by their decomposition into various files. Thus, authors are facing the constantly recurring decisions whether to compose their artifacts internally or externally. For example, one way of organizing documents is by considering the hierarchical structure as a book and splitting it in sections and paragraphs, being included and/or having links to other "books". The organization, however, can also be logic-internal, i.e. depending on the content. We consider the external/internal distinction to be a matter of taste. By making the change management algorithms independent of this we want to give users the freedom to choose.

But, as we consider version control systems at the heart of our management of change methodology, we first have to understand the underlying formalism. Here we develop a mathematical data structure (the *fs*-tree model) that generalizes version controlled file systems and the semi-structured documents alike. This allows us to specify and implement version control algorithms that seamlessly work across the file/file system border (cf. Fig. 3.1). For instance, we are going to provide sophisticated equality theories on top of *fs*-trees (cf. Chap. 5), which are essential to provide notions of consistency and invariants facilitating the identification and propagation of effects of local changes to entire document collections. The beginnings of this have already been published in [98, 104, 105]. Our *fs*-tree model in itself also already induces a non-trivial equality theory over file systems and XML files enhancing version control to arrive at less intrusive edit-scripts.

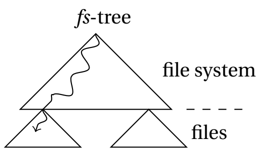


Figure 3.1: Seamless Transition between file/file system boarder.

But why have we opted for a tree data structure? The support for externals definitions rather characterizes a graph data structure. To answer this question, let us consider the purposes for which one might want to use a *graph*, and those for which one might want to use a *tree*. In a tree there is exactly one path from the root to each node in the tree, and a tree has the minimum number of pointers sufficient to connect all the nodes. This makes it a simple and efficient structure. Trees are useful for when efficiency with minimal complexity is what is desired, and there is no need to reach a node by more than one route. However, the latter constraint prevents node sharing. A partial solution are acyclic graphs allowing links to leaves only. The most general topology are cyclic graphs, but in turn are the most difficult to implement, e.g. indefinite looping in recursive algorithms. We solve this issue by utilizing a simple tree structure for the *storage layer*, and model node sharing at a *semantic layer* via explicit links, i.e. links explicitly encoded into *fs*-tree nodes.

Definition 3.1 (*fs*-tree). Let Σ be an alphabet, $\mathbb{D}$ a data set, $\underline{\mathbb{V}}$ a set of node types and $\underline{\mathbb{E}}$ a set of edge types. We call a tuple $\mathscr{T} = \langle \mathbb{V}, \mathbb{E}, \prec, l, \# \rangle$ a ***fs*-tree** if and only if

(1) $\mathcal{G} = (\mathbb{V}, \mathbb{E}, \prec)$ is a $(\underline{\mathbb{V}} \uplus \underline{\mathbb{V}}_{\text{sld}}, \underline{\mathbb{E}} \uplus \underline{\mathbb{E}}_{\text{sld}})$-typed tree with $\forall v \in \mathbb{V}.\ v : \underline{\mathbb{v}} \Rightarrow \underline{\mathbb{v}} <: \underline{\mathbb{f}}$ and $\forall e \in \mathbb{E}.\ e : \underline{\mathbb{v}} \Rightarrow \underline{\mathbb{v}} <: \underline{\ni}$.

(2) $l : \mathbb{E} \rightarrow \Sigma$ is an edge labeling with $\forall v, w \in \mathbb{V}.\ w \neq w' \Rightarrow l(v \rightarrowtail w) \neq l(v \rightarrowtail w')$.

(3) $\# : \mathbb{V} \rightarrow \Sigma^* + \wp(\mathbb{E}) + \mathbb{D}$ is a strict decoding such that

$$\#(v : \underline{\mathbb{v}}) \in \begin{cases} \Sigma^* & \textit{if} \ \ \underline{\mathbb{v}} <: \underline{\mathbb{s}} \\ \wp(\mathbb{E}) & \textit{if} \ \ \underline{\mathbb{v}} <: \underline{\mathbb{d}} \\ \mathbb{D} & \textit{if} \ \ \underline{\mathbb{v}} = \underline{\mathbb{f}} \end{cases}$$

A *fs*-tree is a typed tree where the nodes are ascribed to be of subtype $\underline{\mathbb{f}}$ and the edges to be of subtype $\underline{\ni}$. The edge labeling function servers for node denomination and the strict node decoding[1] function for translating the encoded information within a *fs*-tree back into the file system. We denote the set of all *fs*-trees with $\mathbb{FS}$ and without loss of generality we assume that $\mathbb{D}$ contains all of Σ^*, $\mathbb{N}$ and is closed under n-tuple, set construction and function definition.

Note 3.1. fs-trees are an extension of typed trees (cf. Sec. 2.1). Consequently the definitions on typed trees get a natural extension on *fs*-trees, e.g. $\Uparrow\mathscr{T} = \Uparrow\mathcal{G}(\mathscr{T})$ or $\mathscr{T} \in \ddot{\mathbb{G}}^{\underline{\mathbb{E}}}_{\underline{\mathbb{V}}} = \mathcal{G}(\mathscr{T}) \in \ddot{\mathbb{G}}^{\underline{\mathbb{E}}}_{\underline{\mathbb{V}}}$.

With *fs*-trees we can now abstract on the file/file system border. The following statement precisely designs our intuition regarding file systems.

Lemma 3.1. *Any file system tree can be represented as a fs-tree.*

Proof. Recall that we have restricted ourselves onto UFS compliant systems only (cf. Sec. 2.2). The statement follows from the hierarchical structure property and the supported file types. Ordinary files and directories are encoded to $\underline{\mathbb{f}}$-nodes and $\underline{\mathbb{d}}$-nodes, respectively. Symbolic links, potentially turning a file system into a cyclic graph, are resolved via explicitly encoded $\underline{\mathbb{s}}$-nodes. There is no need for an artificial order construction but using the lexicographic order on the file names. Due to the specification of $\mathbb{D}$ the encoding/decoding follows straightforward. □

To fortify our intuition about file systems and *fs*-trees we are going to investigate one illustrative application.

Example 3.1 (File system). Let us assume the UNIX-like file system presented on the left side of Fig. 3.2. The directory structure up to the two `vc` directories merely serves the purpose to convey the feeling of a typical UNIX directory structure.

The root directory (`/`) contains two sub-directories (`bin` and `home`). The later contains the home directories `cmueller` and `nmueller`. In Christine Müller's home directory (`cmueller`) there are two sub-directories, `tmp` and `vc`, whereas the latter is constituted by the sub-directory `m2.de`. The `m2.de` directory, in turn, contains the sub-directories `adm` and `org`. The former holds one text file, `booking.txt` and the latter a symbolic link `m2@` pointing to the `wedding` directory within Normen Müller's home directory (`nmueller`). The `nmueller` directory is structured quite similar to `cmueller`. It contains a symbolic link `users@` pointing to the root directory (`/`) and

[1]We use the terms "encoding" and "decoding" in sense of encoding a file system tree to a *fs*-tree and decoding a *fs*-tree to a file system.

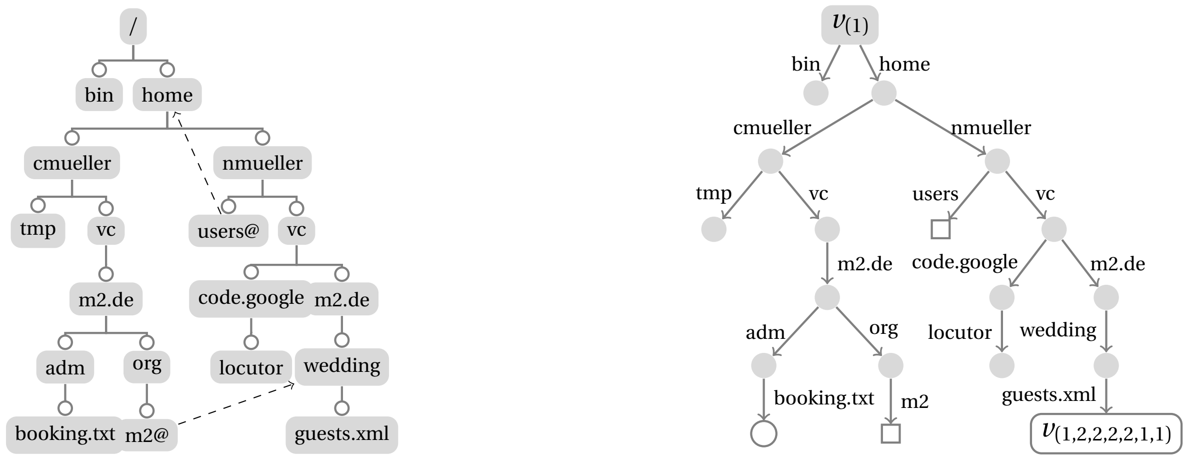


Figure 3.2: A *fs*-tree over a UNIX-like File System.

one sub-directory `vc`. This one comprises two sub-directories `code.google` and `m2.de`. The former contains a `locutor` sub-directory and the later a `wedding` directory containing the semi-structured document `guests.xml`. Note that, the symbolic link `m2@` turns the file system tree into an acyclic graph, the symbolic link `users@`, however, turns the underlying structure into a cyclic graph.

Utilizing our fundamental data structure, we are nevertheless able to represent the information contained in a tree data structure. The corresponding *fs*-tree $\mathcal{T}$ is depicted on the right side of Fig. 3.2. According to the respective file types, directories are represented by ● symbols, ordinary files by ○ symbols, and symbolic links by □ symbols. Two nodes $v, u \in \mathbb{V}(\mathcal{T})$ are connected if u is in the list of content of v. Assuming an adequate alphabet Σ and data set $\mathbb{D}$, respectively, file nodes are encoded by their corresponding content, symbolic link nodes by the target path, and directories are encoded by the set of outgoing edges representing the files they are pointing to. Consequently, *fs*-trees allow us interpreting file system graphs as trees without any information loss. Note, for a better future addressing, we partially applied a linear tree extension [127] on the nodes of $\mathcal{T}$. ♦

Note 3.2. As to UNIX-like file systems, top-level directories (`/`) are special, i.e. the location of $v_{(1)}$ is encoded into a mount point. This can be modeled by adding an artificial root node, similar to an XML root element (cf. Sec. 2.3).

Now, to remove the border between file systems and files completely, we consider simple, semi-structured documents as regards to *fs*-trees.

Lemma 3.2. *Any simple semi-structured document can be represented as a fs-tree.*

Proof. Recall that we have restricted ourselves onto valid XML documents only (cf. Sec. 2.3). The statement follows from the well-formedness requirements. The "One Root Element" rule (cf. Sec. 2.3#R1) assure that we have at most one root node, i.e., the node without a parent, while the "Proper Nesting" rule (cf. Sec. 2.3#R2) guarantees that all other nodes have a unique type. The document order is preserved by applying the inductive construction method for linear tree extensions. Analog to the proof of Lem. 3.1, the encoding/decoding follows directly. □

Note 3.3. Encoding of XML attributes in *fs*-trees is modeled in Sec. 3.2.3.

```
<guests>
  <header>
    <title>Guest List</title>
    <hosts>
      <host>Christine Müller</host>
      <host>Normen Müller</host>
    </hosts>
  </header>
  <body>
    <include>𝒰ℛℒ</include>
  </body>
</guests>
```

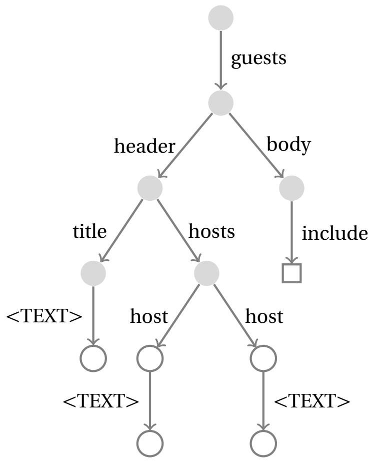


Figure 3.3: A *fs*-tree over a Simple XML Document.

In the following example we demonstrate how to build *fs*-trees over simple, semi-structured documents. The basic idea is that an XML document is a typed tree and that redundancy can be avoided by sharing fragments through include operators.

*Example 3.2 (*XML *documents).* The left side of Fig. 3.3 presents a structure of a guest list in an imagined XML format. The `header` element specifies the cover page. Constituent parts are the title and the hosts. The first one is represented by a `title` element, the second one by a `hosts` element, where the individual hosts are marked up by an explicit `host` element. The actual text is specified by a `body` element and included via an `include` directive. One may think of an XInclude [87] statement, a standardized XML vocabulary for specifying document inclusions. However, as we are at this point restricted to simple semi-structured documents we chose this imaginary XML referencing syntax.

The corresponding *fs*-tree is depicted on the right side of Fig. 3.3 whereas the root node represents the XML root element, here only[2] pointing to the document element `<guests>`. The node typing results in the respective element labels as defined in Sec. 2.3 whereas XML text nodes[3] are treated like file contents and URLs within XML reference statements like symbolic links on file systems. The ordering is preserved by applying the inductive construction of linear tree extensions [127]. This always enables us to insert new tree nodes[4]. ♦

As illustrated in Ex. 3.1 and Ex. 3.2, we use a straightforward analogy between XML files and UNIX file systems. In particular, a version control system based on *fs*-trees would blow up a file system with XML files into one big tree where such files themselves are expanded according to their structure and each unstructured textual content can be versioned individually.

Theorem 3.1. *Any file system tree with simple, semi-structured documents can be represented as one fs-tree.*

Proof. The statement directly follows from Lem. 3.1 and Lem. 3.2. □

[2]Processing instructions or DTD statements are other possible child elements.

[3]We mark edges entering XML text nodes, comment nodes, or processing instruction nodes by distinguished `<TEXT>`, `<COMMENT>` and `<PROCINST>` labels, respectively.

[4]We consider trees to be purely applicative data structures.

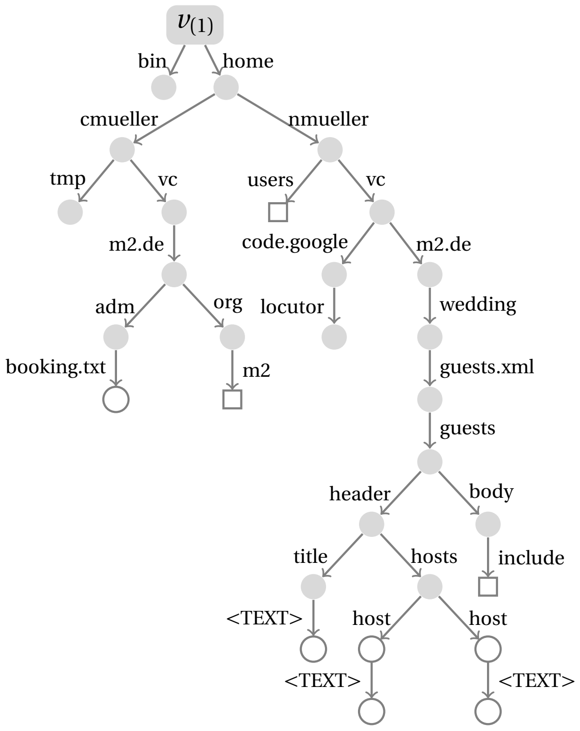


Figure 3.4: A *fs*-tree over a UNIX-like File System with Semi-structured Files.

Applying Thm. 3.1 on the file system represented on the left side of Fig. 3.2 extends the therein pictured *fs*-tree at note $v_{(1,2,2,2,2,1,1)}$ with the *fs*-tree of `guests.xml` depicted in Fig. 3.3 resulting in the *fs*-tree illustrated in Fig. 3.4 where $\underline{\mathbb{V}} = \underline{\mathbb{V}} \uplus \underline{\mathbb{V}}_{\text{syn}}$ is the disjoint union of solid and syntactic node types and $\underline{\mathbb{E}} = \underline{\mathbb{E}} \uplus \underline{\mathbb{E}}_{\text{syn}}$ is the disjoint union of solid and syntactic edge types.[5]

Note 3.4. For round-tripping we have to break the seamless transition between the file/file system border, i.e. we need to identify the transition from the encoded file system to a semi-structured document. In Fig. 3.4, for example, we need to identify the transition from the `wedding` directory into the XML document `guest.xml`. Therefore we suggest to extend $\underline{\mathbb{V}}_{\text{sld}}$ by $\underline{\mathbb{m}} <: \underline{\mathbb{f}}$ where $\underline{\mathbb{m}}$ denotes the respective Internet media type, originally called a multipurpose internet mail extensions (MIME).

Consequently, with *fs*-trees we have the freedom to structurally decompose semi-structured files over file systems, but holding up the dependency graph. This generalization of XML documents promotes version control as well as authoring processes. Regarding authoring process, in sense of making changes, self-contained XML documents are appreciated, for instance, to realize consistent text replacements. But regarding version control, in sense of history tracking, small decomposed XML documents are preferred to manage small-sized revision chunks.

Note 3.5. Some of our ideas have already been discussed in the context of file systems, e.g. REISER4 [118], and file system tools, e.g. XSH [154]. The former proposes to blur the traditional distinction between files and directories by introducing the concept of a "resource". For example, a resource named `ilovemaus.mp3` can be accessed as `./ilovemaus.mp3` to obtain the

[5]Solid node and edge types ($\underline{\mathbb{V}}_{\text{sld}}, \underline{\mathbb{E}}_{\text{sld}}$) are preluded in any set of *fs*-tree node and edge types (cf. Def. 3.1).

compressed audio and as `./ilovemaus.mp3/` for a "directory" of meta files. Our approach can model this behavior: we would interpret the resource `ilovemaus.mp3` as a structured XML file that contains both the compressed audio in a CDATA section and the metadata in custom XML markup. All aspects of the resource can be addressed by standard XPath queries, e.g. fine-grained access to document fragments, for example the title of the third song can be obtained with the query `./ilovemaus.mp3/metadata/song[3]/@title`. The XSH system is also related to our *fs*-tree model. It is a powerful command-line tool for querying, processing and editing XML documents. It only concerns XML files, though, rather than entire file systems.

The next step to work with *fs*-trees is the ability for traversing and encoding. To traverse a *fs*-tree we define a lookup function seeking a target node relative to a source node via a *tree path*, i.e. a sequence of edge labels, not to be confused with a *graph path* being a sequence of node labels (cf. Sec. 2.1).

Definition 3.2 (*fs*-tree lookup). Let $\mathcal{T} = \langle \mathbb{V}, \mathbb{E}, \prec, l, \# \rangle$ be a *fs*-tree. We define a **lookup function** $\centerdot : \mathbb{V} \times \Sigma^* \rightharpoonup \mathbb{V}$ such that

$$v \centerdot \varepsilon = v \quad \textit{where } \varepsilon \textit{ denotes the empty fs-tree path}$$

$$v \centerdot \pi \centerdot a = \begin{cases} w & \textit{if} \quad \exists z \in \mathbb{V}.\ v \centerdot \pi = z \land z \rightarrowtail w \in \mathbb{E} \land l(z \rightarrowtail w) = a \\ \bot & \textit{otherwise} \end{cases}$$

for all $v \in \mathbb{V}, \pi \in \Sigma^*$ and $a \in \Sigma$. We naturally extend the lookup function on *fs*-trees, such that $\mathcal{T} \centerdot \pi = r \centerdot \pi$ *if* $r = {\Uparrow}\mathcal{T}$ and call π a ***fs*-lookup path**, or *fs*-path, in $\mathcal{T}$ if $\mathcal{T} \centerdot \pi$ is defined.

Note 3.6. The *fs*-tree lookup function $\centerdot$ is well-defined: if a node $w \in \mathbb{V}(\mathcal{T})$ is a descendant of a node $v \in \mathbb{V}(\mathcal{T})$ then because $\mathcal{G}(\mathcal{T})$ is a tree, there is a *fs*-path π such that $v \centerdot \pi = w$. The uniqueness is directly entailed by Def. 3.1.2.

A *fs*-path references a location within a *fs*-tree, and in order to obtain the value at the location a *fs*-path refers to — *aka. dereferencing* — we define a *fs*-tree value function.[6]

Definition 3.3 (Value function). Let $\mathcal{T} = \langle \mathbb{V}, \mathbb{E}, \prec, l, \# \rangle$ be a *fs*-tree. A **value function** $\gamma_{\mathcal{T}} : \mathbb{V} \rightarrow \Sigma^* + \wp(\Sigma) + \mathbb{D}$ over $\mathcal{T}$ is a strict function such that

$$\gamma_{\mathcal{T}}(v : \underline{\mathbb{v}}) = \begin{cases} \mathcal{T} \centerdot \pi & \textit{if} \quad \underline{\mathbb{v}} <: \underline{\mathbb{s}} \textit{ and } \pi = \#(v) \textit{ is a fs-path in } \mathcal{T} \\ \{ l(v \rightarrowtail w) \mid v \rightarrowtail w \in \#(v) \} & \textit{if} \quad \underline{\mathbb{v}} <: \underline{\mathbb{d}} \\ \#(v) & \textit{if} \quad \underline{\mathbb{v}} = \underline{\mathbb{f}} \\ \bot & \textit{otherwise} \end{cases}$$

We usually omit the index of γ unless context requires.

Dereferencing encoded file system links results in the *fs*-tree node representing the target of the symbolic file system link. Nodes representing file system directories are dereferenced to the set of their file system content, i.e. therein contained files subsuming ordinary ones as well as file system links and subdirectories. The value of nodes representing ordinary files is just their file content.

We conclude this section with the observation that up to this point we have neglected the versioning itself. In the following we model the core notions of version control systems, in order to map versioning workflows to *fs*-trees.

[6] In term of programming languages with pointer arithmetic, a *fs*-path may be understood as a memory pointer, a *fs*-tree lookup function as a adress operator (usually denoted by "`&`"), and a *fs*-tree value function as a dereference operator (usually denoted by "`*`").

3.2.2 Versioned *fs*-trees

To begin, we want to recall the basic features of version control system. In general, a version control system is a special file server, designed for concurrent editing and storing history information. A normal file server (e.g. NFS) can provide file sharing, but would keep only one version of each file (the most recent one). The core of a version control system, a *repository*, is a centralized store for data. It stores data in form of a file system tree, provides read/write access to the stored data[7], and remembers any modification made to it. A *working copy* is made up of two parts. A local copy of the directory tree of a project and an administrative directory (e.g. `.svn`) in each directory, storing version control information. Users edit their working copy locally and commit their changes to the repository. After a commit, all other users can access the changes by updating their working copies to the latest revision. For each file the respective administrative directory stores the working revision of the file and a time stamp of the last update. Based on this information, version control systems can determine the state of the file. The state of the repository after each commit is called a *revision.* To each revision, an identifier is assigned which identifies the revision uniquely. Without loss of generality, we follow the SUBVERSION model here, which uses natural numbers as identifiers and assigns revisions to the whole tree. A certain file can be left unchanged through different revisions, i.e. files in a working copy might have different revisions, but files in the repository always have the same revision. In terms of finte-state automation, a repository can be understood as a state-transition function where here the status is represented by the respective *fs*-tree.

Definition 3.4 (Repository). A **repository** is a function $\varrho\colon \mathbb{N} \to \mathbb{FS}$ mapping revisions represent by natural numbers to the set of *fs*-trees. We denote the set of all repositories with $\mathbb{R}$.

To define a working copy, we need to capture the intuition that every (versioned) *fs*-tree node comes from a repository.

Definition 3.5 (Repository mapping). Let $\mathcal{T}$ be a *fs*-tree. A **repository mapping** is a function $\omega\colon \mathbb{V}(\mathcal{T}) \rightharpoonup \mathbb{R} \times \Sigma^* \times \mathbb{N}$ and we define the operation

$$\mu^\omega(v) := \begin{cases} \varrho(r)\mathbin{\centerdot}\pi & \textit{if } r \in \operatorname{dom}(\varrho) \textit{ and } \omega(v) = \langle \varrho, \pi, r\rangle \\ \bot & \textit{otherwise} \end{cases}$$

that computes the **corresponding** repository node for $v \in \mathbb{V}(\mathcal{T})$. Note that $\mu^\omega \in \mathbb{V}(\omega_1(v)(\omega_3(v)))$. For convenience we write $\omega(v) = \varrho/\pi@r$ instead of $\omega(v) = \langle \varrho, \pi, r\rangle$.

A repository mapping represents a pointer augmented to a *fs*-tree node referencing the corresponding *fs*-tree node within the respective repository. The correspondence operator dereferences such pointers, i.e. evaluates the referred repository node (cf. Ex. 3.3, p. 44).

Note 3.7. We left out the signature $\mu^\omega\colon \mathbb{V}(\mathcal{T}) \rightharpoonup \mathbb{V}(\varrho(r))$ because both ϱ and r are derived from ω and in this way the signature is kind of awkward. Instead we define μ^ω as an operation, which allows us to retrieve ϱ and r via a simple pattern matching on v over $\omega(v)$. The situation that π might be invalid is captured by the lookup function.

A repository mapping ω is a necessary condition for being under version control, i.e., if a node is not ω-labeled it is not under version control. But if a node is ω-labeled does not imply that is versioned. Therefore we define μ^ω. The operation μ^ω represents a sufficient condition to determine on versioning: for every node $v \in \mathbb{V}(\mathcal{T})$, there has to be a repository $\omega_1(v)$ with a *fs*-tree at revision $\omega_3(v)$ such that $\omega_2(v)$ is a valid *fs*-path in $\omega_1(v)(\omega_3(v))$ to the corresponding node of v.

[7]We neglect modeling access control lists (ACL) for convenience.

In the following, we can now define precisely what it means to be under version control or, in short, to be versioned.

Definition 3.6 (Version controlled). Let $\mathscr{T}$ be a *fs*-tree. We call a node $v \in \mathbb{V}(\mathscr{T})$ **version controlled**, or versioned, with respect to ω *iff* $v \in \operatorname{dom}(\mu^{\omega})$ and we call a *fs*-tree versioned *iff* all nodes $v \in \mathbb{V}(\mathscr{T})$ are versioned.

For version controlled directories we also want to enforce that all the contents of the directories are actually checked out. This ensures that all inode pointers in a directory file are valid.

Definition 3.7 (Repository complete). Let $\mathscr{T}$ be a *fs*-tree. We call a versioned node $v \in \mathbb{V}_{\underline{\mathbb{d}}}(\mathscr{T})$ with $\omega(v) = \varrho/\pi@r$ **repository complete** *iff* for all $a \in \Sigma$ with $e' := \mu^{\omega}(v) \rightarrowtail w' \in \mathbb{E}(\varrho(r))$ and $l(e') = a$ there exists a $w \in \mathbb{V}(\mathscr{T})$ such that $\mu^{\omega}(w) = w'$, $e := v \rightarrowtail w \in \mathbb{E}(\mathscr{T})$ and $l(e) = a$.

With the previous terminology, we can now formalizes the described notion of a working copy as a local reflection of the respective repository directory tree.

Definition 3.8 (Working copy). Let $\mathscr{T} = \langle \mathbb{V}, \mathbb{E}, \prec, l, \# \rangle$ be a *fs*-tree. We call a tuple $\langle \mathcal{H}, \omega, \mathbb{A} \rangle$ a **working copy** *iff*

(1) $\mathcal{H} = \mathcal{G}(\mathscr{T})|_{v}$ for some $v \in \mathbb{V}(\mathscr{T})$

(2) $\mathbb{A}\colon \mathbb{V}_{\underline{\mathbb{f}}} \times \mathbb{D} \to \mathbb{D}$, $\mathbb{A}\colon \mathbb{V}_{\underline{\mathbb{s}}} \times \Sigma^{*} \to \Sigma^{*}$, and $\mathbb{A}\colon \mathbb{V}_{\underline{\mathbb{d}}} \times \wp(\Sigma) \to \wp(\Sigma)$ with

 (2.1) if $v \in \mathbb{V}_{\underline{\mathbb{f}},\underline{\mathbb{s}}}(\mathcal{H})$ then $\#(v) = \mathbb{A}(v, \#(\mu^{\omega}(v)))$

 (2.2) if $v \in \mathbb{V}_{\underline{\mathbb{d}}}(\mathcal{H})$ then v is repository complete and for all $v \rightarrowtail u \in \#_{\mathcal{H}}(v)$ we have $v \rightarrowtail u \in \#_{\mathcal{H}'}(\mu^{\omega}(v))$ and $\#_{\mathcal{H}}(u) = \mathbb{A}(u, \#_{\mathcal{H}'}(\mu^{\omega}(u)))$ where $\mathcal{H}' = \omega_1(v)(\omega_3(v))$.

(3) for all $v \in \mathbb{V}(\mathcal{H}) \cap \operatorname{dom}(\mu^{\omega})$ we have $\mu^{\omega}(v) = \omega_1(v)(\omega_3(v)) \centerdot \omega_2(v)$.

We denote the set of all working copies in $\mathscr{T}$ with $\mathbb{W}_{\mathscr{T}}$ where

$$\mathbb{W}_{\mathscr{T}} = \left\{ w \;\middle|\; \begin{array}{l} w = \langle \mathscr{T}|_{v}, \omega, \mathbb{A} \rangle \textit{ working copy in } \mathscr{T} \textit{ and} \\ \mu^{\omega}(\textit{parent}(v)) \rightarrowtail \mu^{\omega}(v) \notin \mathbb{E}(\varrho(r)) \end{array} \right\}$$

We usually omit the index of $\mathbb{W}$ unless context requires.

Postulate (3.8.1) specifies a locality condition for working copies. That is, working copies are “local” in sense of accessible within the current *fs*-tree, e.g. the current file system. Postulate (3.8.2) ensures for each node the value is equal to the value of the corresponding node modulo any local modifications. The *difference function* $\mathbb{A}$ may be considered as a parametrized collection of transformation functions δ_v for each node $v \in \mathbb{V}(\mathscr{T})$, such that $\mathbb{A}(v, \#(\mu^{\omega}(v))) = \delta_v(\mu^{\omega}(v)) = \#(v)$. For example, think of SUBVERSION: after a fresh working copy synchronization we have $\delta_v = id$ for all $v \in \mathbb{V}(\mathscr{T})$. Postulate (3.8.3) ensures that every versioned working copy node corresponds to the correct repository node.

Note 3.8. What if $\mu^{\omega}(v) = \bot$? In other words, what is the value of $\#(\bot)$? Well, as $\#$ is a strict decoding we have $\#(\bot) = \bot$. The same holds for $\mathbb{A}$.

Lemma 3.3. *Let $\mathscr{T}$ be a fs-tree. If $\langle \mathcal{H}, \omega, \mathbb{A} \rangle$ is a working copy in $\mathscr{T}$, then $\langle \mathcal{H}|_{\pi}, \omega, \mathbb{A} \rangle$ is a working copy in $\mathscr{T}$ for all fs-paths π in $\mathcal{H}$.*[8]

[8]As we have not explicitly introduced subtree identification via *fs*-paths, $\mathcal{H}|_{\pi}$ should be understood as a shorthand for $\mathcal{H}|_{\mathcal{H}\centerdot\pi}$.

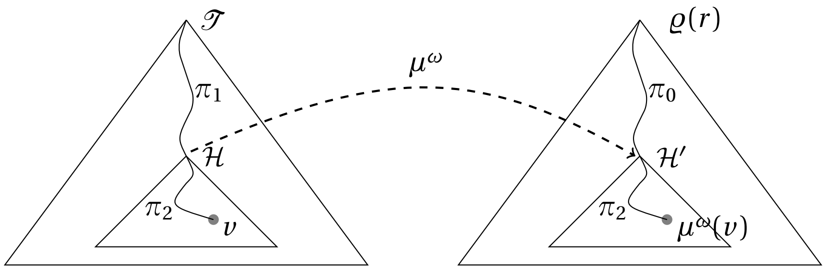


Figure 3.5: A Working Copy for $\varrho/\pi_0@r$ in $\mathscr{T}$ at π_1.

Proof. Given that $\mathcal{H}|_\pi \sqsubseteq \mathcal{H} \sqsubseteq \mathcal{G}(\mathscr{T})$ and (3.8.2) – (3.8.3) are universal, all conditions are inherited by $\mathcal{H}|_\pi$, in particular, repository completeness. Consequently $\langle \mathcal{H}|_\pi, \omega, \mathbb{A}\rangle$ is a working copy in $\mathscr{T}$. □

In the following we visualize on the basis of an abstract example the concepts introduced herein and the notion of a working copy, in particular.

Example 3.3 (Working Copy). Let us consider the working copy structure depicted in Fig. 3.5. The *fs*-tree $\mathscr{T}$ constitutes a local file system tree and the subtree $\mathcal{H} \sqsubseteq \mathcal{G}(\mathscr{T})$ a working copy $\langle \mathcal{H}, \omega, \mathbb{A}\rangle$ in $\mathscr{T}$ at π_1. Here we have $\omega_1(v) = \varrho$, $\omega_2(v) = \pi_0/\pi_2$ where $\pi_2 \in \Sigma^*$ such that $\mathcal{H}\centerdot\pi_2 = v$ and $\mu^{\omega}(v) = \varrho(r)\centerdot\pi_0\centerdot\pi_2$, and $\omega_3(v) = r$ for all $v \in \mathbb{V}(\mathcal{H})$. Hence $\langle \mathcal{H}, \omega, \mathbb{A}\rangle$ is a working copy. ♦

We are now in a position to understand the how to map common versioning commands like `add`, `update`, `remove`, and `commit` to *fs*-trees: in case of SUBVERSION and `add` command on a file system entry f adds repository mapping information for the entry to the respective administrative directory. An `update` command on a versioned file f retrieves from the repository the difference δ_f^k between the working copy revision of f and the specified revision k and merges it into f. A `remove` command deletes the repository mapping information from the administrative directory. Finally, a `commit` command communicates the differences between the repository revision of f (cached in the administrative directory) and the potentially modified working copy entry to the repository, where it is incorporated into the data store and updates the metadata in the administrative directory (for details about these commands see [28]).

Definition 3.9 (Version commands on *fs*-trees). Let $\langle \mathcal{H}, \omega, \mathbb{A}\rangle$ be a working copy and $v \in \mathbb{V}(\mathcal{H})$. The version control command » `add v` « extends the repository mapping ω by a new pair $(v, \omega_1(\mathit{parent}(v))/\omega_2(\mathit{parent}(v))\centerdot a@\omega_3(\mathit{parent}(v)))$, where a is the name of v. An » `update -r k v` « changes $\omega_3(v)$ to k, and merges δ_f^k into $\mathbb{A}(v, \#(\mu^{\omega}(v)))$ so that (3.8.3) is maintained. A » `remove v` « command deletes the pair $(v, \_)$ from ω. A » `commit v` « command communicates $\mathbb{A}(v, \#(\mu^{\omega}(v)))$ to the repository $\omega_1(v)$ which extends the repository $\varrho' := \omega_1(v)$ with a new pair $(k, \mathcal{H}')$, where $k = \max(\mathrm{dom}(\varrho)) + 1$ and $\mathcal{H}'$ is derived from $\varrho(k)$ by replacing $w := \varrho(k)\centerdot\omega_3(v)$ with $\mathbb{A}(v, w)$.

With the here introduced basis in terms of repositories and working copies, we are in the following going to extend these concepts to the outstanding concept of externals definitions.

3.2.3 Version Control with Properties and Externals

As mentioned at the beginning of Sec. 3.2.1, sometimes it is useful to construct working copies made of a number of different sub-directories to come from different locations in a repository, or from different repositories altogether. One could set up such a structure by hand, but if this structure is important for everyone else using the respective repository, every other user would need to perform the same manual setup.

To support those structures of working copies without the need for manual reconstruction, we are going to formalize the well-known construct of *externals definitions*. Basically, an externals definition is a versioned *property* defined on a working copy, mapping a directory to an uniform resource locator (URL) and a particular revision of another versioned directory.

How this works is best explained with an example. Suppose we check out our code from a repository:

```
svn co https://svn.your.host/project/trunk project
```

Inside our project, we need code from a different location in the repository or maybe even from some other repository. This could be a software module, test data, a generic part of your company's standard build system or other things shared between projects. Many people do this manually:

```
cd project
svn co https://svn.your.host/whatever/tags/release−1.3 whatever
```

That is tedious and error prone. A better approach is to use an externals definition that tells SUBVERSION to check out that project automatically for us. It works by adding some magic metadata to the `project` directory

```
cd project
svn propset svn:externals 'whatever https://svn.your.host/whatever/tags/release−1.3' .
```

The `svn:externals` property accepts a set of declarations, each consisting of a directory name, an URL, and an optional revision. So, every time we check out (or update) the parent project, the `whatever` project will be checked out, too.

This ability to mix various working copies with respect to corresponding repositories and/or revisions gives us the notion of **nested working copies**, i.e. a working copy n inside another working copy w. Hence, externals definitions themselves lead to nested working copies, but with the decided advantage the structural information being versioned themselves: once *defined* on a working copy, all other users benefit by updating their working copies to the latest revision. Note, however, the fact that the respective *fs*-tree edge, connecting the root node u of n to a leaf node v of w, is not versioned in the corresponding repository $\varrho(r)$ of w, i.e. $\mu^{\omega}(v) \rightarrowtail \mu^{\omega}(u) \notin \mathbb{E}(\varrho(r))$, but the respective property. From here on, when ever we refer to a "nested working copy" we mean nested working copies defined by externals definitions.

To capture the notion of nested working copies, we first extend *fs*-trees by *properties* to eventually adapt the definition of a repository and working copy, respectively, through to externals definitions.

Definition 3.10 (*fsp*-tree). Let $\mathbb{P}$ be an arbitrary set. A ***fsp*-tree** is a *fs*-tree $\mathscr{T}$ together with a property function $\beta\colon \mathbb{V}(\mathscr{T}) \times \mathbb{P} \rightharpoonup \mathbb{D}$. We call elements of $\mathbb{P}$ **property names** and denote the set of all *fsp*-trees with $\mathbb{FS}^{+}$.

Property functions, or properties for short, are custom metadata attached to a *fs*-tree. However, we cannot map *fsp*-trees directly to file system trees. Version control systems solve this issue by administrative directories, though, we assume an equivalent file system structure. Regarding XML documents, the mapping is straightforward: properties become XML attributes in the obvious way, for example cf. Fig. 3.7.

Note 3.9. Note that the definition of *fsp*-trees allows us now to naturally extend Thm. 3.1 on general semi-structured documents rather than simple ones only. In turn, we can encode XML reference statements, like XInclude, to *fsp*-tree nodes of type $\underline{\mathbb{s}}$. For instance, let us revisit Ex. 3.2 but using the legal XInclude statement `<xi:include href="`$\mathcal{URL}$`"/>` instead of our artificial `<include>`$\mathcal{URL}$`</include>` statement. Rather than only using the element label for node typing, here the property `href` enables us to differentiate XML reference statements. The handling of URLs, however, remains the same.

Most version control systems provide a distinguished property for externals definitions, we will assume the existence of a property name `ext` $\in \mathbb{P}$ with a specific signature: a symbol is associated with a repository entry at a specific (remote) path and revision.

Definition 3.11 (Externals Definition). Let $\mathscr{T}$ be an *fsp*-tree, and $v \in \mathbb{V}(\mathscr{T})$. An **externals definition** is a property function $\xi(v) := \beta(v, \mathtt{ext})$, such that $\xi(v)\colon \Sigma^+ \rightharpoonup \mathbb{R} \times \Sigma^* \times \mathbb{N}$. We call elements of $\xi(v)$ **externals** and denote the set of all externals in $\mathscr{T}$ with $\mathbb{X}_{\mathscr{T}} = (\xi(v))_{v \in \mathbb{I}}$ with $\mathbb{I} = \{v \mid v \in \mathbb{V}(\mathscr{T})\ \mathit{and}\ v \in \mathrm{dom}(\xi)\}$. To simplify matters, we usually omit the index of $\mathbb{X}$ unless context requires.

We consider an external $\langle \pi_0 \centerdot d, \varrho/\pi_1 @ r \rangle \in \xi(v)$ to be **well-defined** *iff* $\varrho(r) \centerdot \pi_1$ is defined, otherwise **dangling**.

An external $\langle \pi_0 \centerdot d, \varrho/\pi_1 @ r \rangle \in \xi(v)$ is **admissible** *iff* well-defined and there is a $u \in \mathbb{V}(\mathscr{T})$ such that $u = v \centerdot \pi_0$ and $u \centerdot d = \perp$. We call $\xi(v)$ admissible *iff* all $e \in \xi(v)$ are admissible.

The well-definedness criterion ensures that the repository referring to actually exists and the admissibility specifies $\pi_0 \centerdot d$ is relative to v and the symbol d has not been versioned yet.

External definitions are not local to a working copy, but global in sense of all working copies of the respective repository obtain these properties. Thus external definitions have to be stored in repositories. Note, in case of SUBVERSION's externals definitions, they extend the working copy they are defined on. That is, the underlying *fsp*-tree of a working copy is extended by the underlying *fs*-tree of the externally defined working copy. In the following we naturally extend repositories and working copies to externals.

Definition 3.12 (Repository with Externals). A **repository with externals** is a repository whose externals are admissible.

Definition 3.13 (Working Copy with Externals). Let $\mathscr{T}$ be a *fsp*-tree. We call a working copy $\langle \mathcal{H}, \omega, \mathbb{\Delta} \rangle$ in $\mathscr{T}$ a **working copy with externals** if and only if

(1) $\beta(v, \mathrm{k}) = \mathbb{\Delta}(v, \beta(\mu^{\omega}(v)), \mathrm{k})$ for all $v \in \mathbb{V}(\mathcal{H})$ and $\mathrm{k} \in \mathbb{P}$.

(2) for all $v = \mathcal{H} \centerdot \pi_0$ with $\langle \pi_1, \Theta \rangle \in \xi(v)$ we have $\omega(\mathcal{H} \centerdot \pi_0 \centerdot \pi_1) = \Theta$.

Similar to postulate (3.8.2), postulate (3.13.1) ensures for each versioned node the property value is equal to the property value of the corresponding node modulo any local modifications with respect to property names. This restriction does not refer solely to externals definitions, but to properties in general. Postulate (3.13.2) ensures that all externally defined working copies hold the correct repository mapping with respect to the corresponding externals definition. It therefore follows directly that $\mathbb{W}_{\mathscr{T}}$ is implicitly extended by $\mathbb{X}_{\mathscr{T}}$.

Corollary 3.1. *Let $\mathscr{T}$ be an fsp-tree and $w = \langle \mathcal{H}, \omega, \mathbb{\Delta} \rangle \in \mathbb{W}_{\mathscr{T}}$. For all $\xi(v) \in \mathbb{X}_{\mathcal{H}}$ and for all admissible $e \in \xi(v)$ with $e = \langle \pi, \Theta \rangle$, we have $w|_e = \langle \mathcal{H}|_{v \centerdot \pi}, \omega, \mathbb{\Delta} \rangle \in \mathbb{W}_{\mathscr{T}}$ where $\omega(v \centerdot \pi) = \Theta$.*

From here on we will always assume that repositories and working copies, respectively, have externals without further mention. In the following we revise our abstract working copy Ex. 3.3 with the addition of externals definitions.

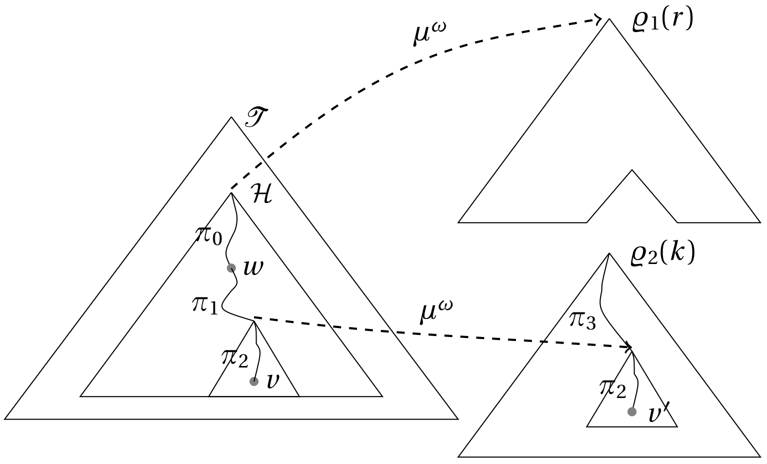


Figure 3.6: A Working Copy for $\varrho_1/\varepsilon@r$ in $\mathscr{T}$ with an External for $\varrho_2/\pi_3@k$.

Example 3.4 (Working Copy with External). Let us consider the working copy with externals depicted in Fig. 3.6. The *fsp*-tree $\mathscr{T}$ constitutes a local file system tree and the subtree $\mathcal{H} \sqsubseteq \mathcal{G}(\mathscr{T})$ a working copy $\langle \mathcal{H}, \omega, \mathbb{A} \rangle$. The node $w = \mathcal{H}.\pi_0$ has an externals definition with $\xi(w) = \langle \pi_1, \varrho_2/\pi_3@k \rangle$. Here we have $\omega_1(v) = \varrho_2$, $\omega_2(v) = \pi_3/\pi_2$, and $\omega_3(v) = k$ where $\varrho_2(k).\pi_3.\pi_2 = \mu^\omega(v)$ and $v = \mathcal{H}.\pi_0.\pi_1.\pi_2$. Hence $\langle \mathcal{H}, \omega, \mathbb{A} \rangle$ is a working copy with externals. ⧫

Thanks to version control systems, externals definitions are stored in administrative directories as well and therefore we do not have to modify the previously introduced mapping between file systems (with externals) and *fsp*-trees.

In case of XML documents, however, we have to perform some adaptations: for each XML element induced by a *fsp*-tree node, we have to identify the corresponding node in the respective repository to guarantee a fine-granular version control for XML fragments. To encode the information about the components of ω we propose to extend the elements in the respective XML format by following three attributes[9]:

- the `locutor:rep` attribute specifies the repository root URL
- the `locutor:path` attribute represents the respective remote path, and
- the `locutor:rev` attribute represents the revision

The `locutor:path` attribute has to be annotated to the XML document element and is implicitly defined for all descendants (relative to the XML document element). The XML attributes `locutor:rep` and `locutor:rev` are inherited by XML child elements. All attributes may be individually overwritten. This enables authors to mix XML fragments within one XML document from various repositories and revisions. In this case XML fragments act like nested working copies.

[9]As most XML applications allow foreign namespaces, we place the new attributes in our own name space `locutor`, which we map to `http://code.google.com/p/locutor`.

Regarding externals definitions, we cannot encode all of the required information in attributes unless we would pack additional semantics in the attribute values. Due to common folklore, however, this is seen not as a good schema design. Therefore we represent XML elements defined by externals definitions with empty `locutor:external` elements. A `locutor:external` element carries a `rev`, `rep`, `path`, and `name` attribute. Similar to the encoding of ω, the value of a `rev` attribute specifies the external revision, the value of a `rep` attribute specifies the external repository root URL, the value of a `path` attribute specifies the external remote path, and the value of a `name`[10] attribute identifies the externally defined XML fragments via an XPath expression. After synchronization, `locutor:external` elements are replaced with the referenced XML fragments. Each externally retrieved XML fragment is extended by respective `locutor:rev`, `locutor:rep`, `locutor:path`, and `locutor:name` attributes. We assume, without loss of generality, XML elements externally linked do not corrupt XML documents with respect to XML validity.

```
<guests>
  <header>...</header>
  <body>
    <locutor:external rev="512" rep="https://m2.de/r/adm" path="friends.xml" name="/*[1]//"/>
 </body>
</guests>
```

Figure 3.7: An XML Document with an Externals Definition.

Example 3.5 (A versioned XML document). To gain a better understanding of the just introduced XML attributes and elements for version control, let us put the XML document presented in Fig. 3.3 under version control. Figure 3.7 depicts our example XML document enriched with versioning metadata. For illustration, let us assume to define an externals definition on the `body` element in order to retrieve $\mathcal{URL}$ rather than to handle the `include` element as a versioned symbolic link. Therefore the `body` element is extended by an externals definition (`locutor:external`) referring to the node set identified by the XPath expression of the `name` attribute evaluated on the remote file `https:://m2.de/r/adm/friends/xml` at revision 512.

After synchronization the resulting XML document is depicted in Fig. 3.8. In the XML root element `guests` the repository root URL (`locutor:rep`), the remote path (`locutor:path`), and

```
<guests locutor:rep="https://m2de/r/wedding" locutor:path="guests.xml" locutor:rev="@HEAD">
  <header>...</header>
  <body>
    <person locutor:rev="512" locutor:rep="https://m2.de/r/adm"
         locutor:path="/friends.xml#/*[1]/*[1]">...</person>
    <person locutor:rev="512" locutor:rep="https://m2.de/r/adm"
         locutor:path="/friends.xml#/*[1]/*[2]">...</person>
   [...]
  </body>
</guests>
```

Figure 3.8: An XML Document with Externals.

[10]We chose the term “name” to mimic SUBVERSION terminology, rather than using “ref”, for example.

the revision (`locutor:rev`) are presented. Furthermore, following the example of XInclude and the handling of externals definition by SUBVERSION, the externally defined node set is included into the `guests.xml` document.

The explicit markup of externals definitions within XML documents mimics SUBVERSION's procedural method: The repository only stores the *externals definitions* as such but respective working copies retrieve the thereby defined *externals*. Analog an XML document within a working copy is flattened with respect to node sets identified by externals definitions. ♦

We are now in a position to implement the version control commands from Definition 3.9 on XML files to achieve the seamless operation of version control across the file/file system border.

Disclaimer 3.1. At this point the author likes to explicitly point out that the mapping of version control commands on *fs*-trees and *fsp*-trees, respectively, i.e. with and without properties (cf. Def. 3.14 and Def. 3.9), reflects the current state of this research. However, the development process is not yet complete! In our implementation, at the moment, we use our fundamental data structure "only" for redundancy resolution (cf. Sec. 3.2.4), consolidation (cf. Chap. 4), semantic differencing (cf. Chap. 5), and change impact analysis (cf. Chap. 6). This was a design decision to eventually be once able to fully bake our MoC cake with the foresight that a few ingredients are missing through to perfection. The drawing on the full integration into a version control system and therefore required fine-granular versioning of semi-structured documents is ongoing work (cf. Sec. 5.2.3). The following definition provides the current status of this work regarding version control on XML files.

Definition 3.14 (Version Control on XML files). Let X be a versioned XML file that admits the `locutor` attributes and e an element in X. The version control command » `add e` « annotates e by `locutor` attribute `locutor:rev="k"`, where k is current revision. Note that the `locutor:rep` and `locutor:path` attributes are inherited from the parent. Dually, a » `remove e` « command just removes the `locutor` attributes. A » `commit e` « command computes the XML-differences δ_e (cf. Sec. 5.2.2) between e and the cached repository copy e' and communicates them to the repository in `locutor:rep`, which creates a new repository revision by adjusting the `locutor` attributes in X' and its original in the repository. An » `update -r k e` « command changes `locutor:rev` to k, and merges δ_f^k into e. Versioning commands for properties like `propset`, `propdel` are modeled by the obvious `locutor` attribute movements.

In case of a `commit` command, we assume that the underlying version control system stores a cached repository copy X' (*aka.*base version) of X in the administrative directory. From this, we can take the element e' that is the cached repository copy for e.

In case of an `update` command, note that we are not (yet) specifying how the merge actually works and appeal to the intuition of the reader. In fact, we are currently working on an extension of XML merge algorithms to include format equality information and *fsp*-tree property attributes. Note that given suitable merging algorithms, we can execute the merge on the client, since SUBVERSION (and thus *locutor*) caches base revisions in a private part of the working copy (cf. Sec. 5.2.3).

Obviously the real interest in the seamless integration of file systems and XML trees comes with general and model-based *diff*/*patch*/*merge* algorithms, and indeed the work reported here and the need for formalizing a unified data structure comes out of the author's ongoing development of exactly these algorithms in the SCAUP/SCALAXX projects [103, 102].

However, as mentioned in Disclaimer 3.1, we use *fsp*-trees not only for the seamless integration of file systems and XML trees, but also as a basis for our consolidation process (cf. Chap. 4) as well as for resolving redundancies. The latter we consider in the following.

3.2.4 Redundancy Resolution on *fsp*-trees

VCS clients allow us to browse repositories, check for changes, commit changes, update local working copies and examine the revision history. A version control client manages files and directories that change over time and are stored in a central repository. The version control repository is much like an ordinary file server, except that it remembers every change ever made to files and directories. This allows us to access older versions of files and examine the history of how and when data changed. Unfortunately, all these features are restricted to single working copies at a time, so to keep several (related) working copies in sync with central repositories, one has to update each one by its own.

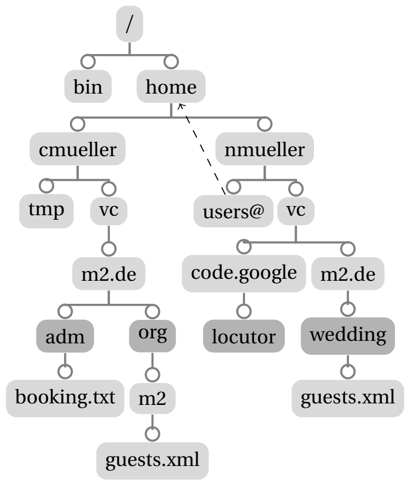


Figure 3.9: A Structure of Working Copies.

```
#!/bin/bash
# === update–local.sh ===
VCHOMES="/home/cmueller/vc\␣
      /home/nmueller/vc"
for i in $VCHOMES ; do
   cd $i ; VCDOMS=‘ls $i‘;
    for j in $VCDOMS ; do
       cd $j ; WC=‘ls $j‘;
        for k in $WC ; do
           cd $i/$j/$k; svn up $1
        done
    done
done
```

Figure 3.10: A SUBVERSION Update Script.

Example 3.6 (Redundant Working Copies). Let us take up again the first example illustrating *fs*-trees (cf. Ex. 3.1). In this case, however, we consider the directories `adm`, `org`, `locutor` and `wedding` as working copies, where $\omega(\texttt{adm}) = \varrho_1/\pi_1@r_1$, $\omega(\texttt{org}) = \varrho_1/\pi_2@r_2$, $\omega(\texttt{locutor}) = \varrho_2/\pi_3@r_3$, and $\omega(\texttt{wedding}) = \varrho_1/\pi_4@r_4$. In addition, we understand `m2` to be an external with the externals definition on `org` such that $\xi(\texttt{org}) = \langle \texttt{m2}, \varrho_1/\pi_4@r_4\rangle$. Figure 3.9 depicts the respective file system graph. Note that for readability sake we left out the corresponding *fs*-tree representation but handle the file system graph as such (cf. Thm. 3.1). In order to keep `wedding` as well as `m2` in sync with the central repository we have to update both by invoking the respective version control `update` command on each directory.

A simple approach to solve this problem is a shell script as listed in Fig. 3.10. The variable `VCHOMES` specifies the local directories of used version control domains (`VCDOMS`) and the innermost loop updates all working copies within the respective directory (`WC`). Consequently, all entries within all working copies are updated. Redundant working copies, however, get updated more than once, which is time consuming and a sheer waste of disk space and leads to inconsistencies until the next synchronization process when only one working copy is changed. In our case `m2` and `wedding` are updated and stored twice. But besides for synchronizing all working copies with the central repositories, we also have to run the script to propagate changes from one working copy (e.g. `wedding`) to all remaining ones (e.g. `org`) having this one defined as an externals definition. Otherwise, for example, changes to `wedding` are not immediately available in `m2`. This is once more a very time consuming task that requieres strict discipline. Furthermore, the identification of nested working copies is entirely left to the user. This means that, in order to find out which working copies are related to each other via externals definitions and which of those have already been checked out, the user has to manually analyze each working copy. ♦

All in all, maintaining a structure of working copies related to each other via externals definitions, is — even for such a small structure — an extremely complex task. An automatism handling these dependencies is thus highly desirable and consequently would foster logical separations of multiple repositories via externals definitions. Informally we summarize these exemplified problems as the **nested working copy problem** (NWCP). We have identified three major issues:

(NWCP 1) *Identification* of redundancy. In a structure of related working copies we have to determine by ourselves if a nested working copy has been already checked out to a different location.

(NWCP 2) *Synchronization* of redundant externals definitions. If NWCP 1 exists, we have to care about updating each redundant working copy directory by ourselves.

(NWCP 3) *Propagation* between redundant externals definitions. If NWCP 1 exists, we have to care about committing the modifications on a redundant working copy, before switching to collateral ones.

In order to solve these problems, we first need to better grasp the concept of *redundancy* on working copies. Informally we can say a working copy is redundant, i.e. already locally available, if at least one other working copy exists so that the nodes of both two working copies reference the same repository nodes. The following definition renders this statement more precisely.

Definition 3.15 (Redundancy). Let $\mathscr{T}$ be a *fsp*-tree and $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ a working copy in $\mathscr{T}$. We call $v \in \mathbb{V}(\mathscr{T})$ **redundant** to $\mathcal{H}$ ($v \gg \mathcal{H}$) *iff* there is a $w \in \mathbb{V}(\mathcal{H})$ such that $\omega(w) = \omega(v)$. We call a working copy $m = \langle \mathcal{H}', \omega, \mathbb{A} \rangle$ redundant to w ($m \gg w$) *iff* $v \gg \mathcal{H}$ for all $v \in \mathbb{V}(\mathcal{H}')$.

With this definition we now can automatically identify redundant working copies and solve problem NWCP 1. But note that the concept of redundancy is a relation rather than a function. There may exist a number of redundant working copies — including each working copy to itself. This aspect needs to be and is taken into account in the realization part of this work (cf. Part III).

There is yet another subject regarding redundancies. The last part of Definition 3.15 poses a problem in itself on the basis of SUBVERSION's flexibility. As a general principle, SUBVERSION tries to be as flexible as possible. One special kind of flexibility is the ability to have a working copy containing files and directories with a mix of different working revision numbers. Hence, switching repositories and/or revisions within a versioned *fs*-tree is legal, however, invalid path labelings are restricted due to repository completeness and (3.1.2).

For example, suppose we have a working copy entirely at revision 10. We edit the file `booking.txt` and then perform a `commit` command, which creates revision 15 in the repository. After the commit succeeds, one might expect the working copy to be entirely at revision 15, but that is not the case! Any number of changes might have happened in the repository between revisions 10 and 15. The client knows nothing of those changes in the repository, since we have not yet run an `update` command, and a `commit` command does not pull down new changes — one of the fundamental rules of SUBVERSION [28]. Therefore, the only safe thing the SUBVERSION client can do is mark the one file, `booking.txt`, as being at revision 15. The rest of the working copy remains at revision 10. Only by running an `update` command the latest changes can be downloaded and the whole working copy be marked as revision 15. We call a working copy having this status being *uniform*.

Definition 3.16. A *fs*-tree $\mathscr{T}$ is **uniform** *iff* there exists a $\varrho \in \mathbb{R}$ and a $r \in \mathbb{N}$ such that $\omega_1(v) = \varrho$ and $\omega_3(v) = r$ for all $v \in \mathbb{V}(\mathscr{T})$.

The property of a working copy being uniform directly leads us to the following conclusion.

Corollary 3.2. *Let $\mathscr{T}$ be a fsp-tree and $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ a working copy in $\mathscr{T}$.*

1. *We have $v \gg \mathcal{H}$ for all $v \in \mathbb{V}(\mathcal{H})$.*

2. *If $m = \langle \mathcal{H}', \omega', \mathbb{A} \rangle$ is a working copy, $\mathcal{H}, \mathcal{H}'$ are uniform, and $r = \Uparrow\mathcal{H}'$ with $r \gg \mathcal{H}$ then we have $m \gg w$.*

Proof. Choose $w = v$ for (3.2.1) and (3.2.2) is a consequence of Lem. 3.3.

In order to find the key to the outstanding issues NWCP 2 and NWCP 3, we now substitute each identified redundant working copy m by a fresh *fsp*-tree node u of type $\underline{\mathtt{s}}$ such that $\gamma(u) = r$ where $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ with $r = \Uparrow\mathcal{H}$ and $m \gg w$. Informally, we replace redundant copies working with symbolic file system links.

Definition 3.17 (Redundancy Resolution). Let $\mathscr{T}$ be a *fsp*-tree. A **redundancy resolution** is a function *resolve*: $\mathbb{FS}^+ \times \mathbb{W} \to \mathbb{FS}^+ : \mathscr{T} \times \mathbb{W}_{\mathscr{T}} \mapsto \mathscr{T}[m/u]$ where $m \gg w$ with $w = \langle \mathscr{T}|_{\gamma(u)}, \omega, \mathbb{A} \rangle$, $u : \underline{\mathtt{s}} \notin \mathbb{V}(\mathscr{T})$, and *type*$(\gamma(u)) \not<: \underline{\mathtt{s}}$ for all $m \in \mathbb{W}_{\mathscr{T}}$.

By decoding a resolved *fsp*-tree, the problem of synchronizing as well as propagating modifications between related working copies solves itself via symbolic file system links. That this is already the first concrete MoC step, in sense of "*change once, change everywhere*"! However, as mentioned below of Def. 3.15, the realization part of this work (cf. Part III) has excluded the fact that a working copy is redundant to itself. The restriction *type*$(\gamma(u)) \not<: \underline{\mathtt{s}}$ is mandatory, though. Otherwise entire working copies could disappear in $\underline{\mathtt{s}}$-nodes.

With these formalisms at hand, we now can consider a precise definition of the nested working copy problem.

Definition 3.18 (Nested Working Copy Problem). Let $\mathscr{T}$ be a *fsp*-tree. We call the **nested working copy problem** the difficulty to find the subtree $\mathscr{T}' \sqsubseteq \mathscr{T}$ such that *resolve*$(\mathscr{T}', \mathbb{W}_{\mathscr{T}'}) = \mathscr{T}'$.

Regarding our Ex. 3.6, we solve the nested working copy problem by first identifying the externals definition `m2` on `org` to be redundant to `wedding` due to equivalent repository root URL, remote path and revision. We then substitute `m2` with an symbolic file system link `m2@` pointing to `wedding` resulting in the file system graph depicted at Ex. 3.1. Note that this task is less trivial than it seems at the first glance. For instance, replacing redundant file system entries with symbolic file system links forces to introduce *update scopes* to avoid infinite loops inside version control commands.

Besides that, the complexity of the NWCP highly depends on the identification of $\mathbb{W}_{\mathscr{T}}$ of the particular *fsp*-tree $\mathscr{T}$. In the next chapter we discuss a more efficient approach based on our consolidation component.

3.3 Conclusion

We have presented an abstract theory of collaborative content management and version control for collections of semi-structured documents and, in particular, have defined version control algorithms on our *fs*-tree model. We have laid the foundation for extended version control in arbitrary XML formats that allow foreign-namespace attributes and extend them to seamlessly work across the file/file system border. Furthermore, we have shown how to control redundancy and confusion induced by duplicate externals definitions. Thus, with the introduction of our fundamental data structure we have already improved collaborative authoring process across project boundaries.

The basic idea is that an XML file is an ordered tree and that code redundancy can be avoided thanks to shared code referenced through XML include operators, so that a deep analogy can be carried out between XML files and UNIX-like file systems, which our model handles simultaneously. One can imagine that a version control system based on this model blows up an XML file system into a big tree where files themselves are expanded according to their structure and that each unstructured textual content can be versioned individually.

Acknowledgments. While the specific definitions and the remaining results are ours, the whole section benefited strongly from discussions with and concrete advices by Michael Kohlhase. The author would like to thank him for his initial help on drafting the fundamental data structure in [104].

Chapter 4

Consolidation

Relationships are all there is. Everything in the universe only exists because it is in relationship to everything else. Nothing exists in isolation.

— Margaret J. Wheatley

As digital devices have found their way into nearly all domains of every-day life, the amount of digital content is increasing and becomes more valuable and important. Managing a considerable quantity of documents involves administration efforts and certain strategies for ordering and arrangement to keep track of content and structure of the collection — especially over a long period. The problem is intensified by (1) the complex and partly high-dimensional (temporal, spatial) characteristics of semi-structured data and (2) an increasing information fragmentation [73]. A typical result is a progressively disorientation within heterogeneous document collections regarding origin, context, and inter-relations.

With the help of Semantic Web technologies, which ensure machine processability and interchangeability, we apply semantic knowledge models and paths to organize and describe heterogeneous document collections. A document collection is no longer just an aggregation of separate items, but forms an individual data base. Such a data base provides rich and valuable information for innovative change management, e.g. an aggregated view of the pool of document collections and their relationships to each other.

4.1 Introduction

The support for business processes by the IT site has mainly the modeling, analysis and implementation of processes in focus. The support in terms of new forms of collaboration within and between enterprises, especially in terms of involved document management, has not been addressed, though. The emerging complexity in the wake of globalization of business so requires a uniform, consistent and transparent management of individual projects and their dependencies to each other. Common document management systems, however, primarily focus on the management of documents themselves. Usage of metadata, for example, on cross-project dependencies is rather restricted. Problems and barriers which appear when users deal with management tasks on heterogeneous document collections mainly result from lacking expressiveness and flexibility of the traditional data models to represent individual knowledge.

The aim of our MoC cake and consolidation slice (cf. Fig. 1.1), in particular, is to provide a homogeneous data base on heterogeneous collections of documents prepared to facilitate and to improve the analysis and multidisciplinary management of business processes. With respect to our major assumptions that business processes are inscribed in document collections and these are for reasons of history tracking (cf. Sec. 3.1.1) version controlled, we develop in the following a homogeneous data base for heterogeneous, versioned file system trees. This means that at this point we do not model dependencies induced by relations within and between the documents (fine-grained), but dependencies on the project-level (coarse-grained). In the style of extract, transform and load processes (ETL) in the area of databases and data warehouses, in particular, our consolidation component therefore contains the extraction, transformation and preparation of coarse-grained metadata from diverse working copies to strengthen cross-project collaboration and process integration. Identification and evaluation of fine-grained dependencies as well as the rippling of changes along both dependency-levels is treated in Chap. 6.

But where should we classify our consolidation component? On the conceptual level, this is a set of ordered pairs (cf. Def. 4.2). From a technical side, our consolidation component is something between a regular database and a data warehouse — we call it a *datahouse*. The primary difference between a database and a data warehouse is that while the former is designed (and optimized) to *record*, the latter has to be designed (and optimized) to *respond to analysis questions* that are critical for a business. Obviously we settle consolidation in the middle between the two technologies: information *about* working copies have to be recorded as well as analyzed to enable cross-project change management.

Recall, the aim of our consolidation slice is to bring together coarse-grained information about disparate sources and put the information into a format that is conducive to making cross-project decisions on change processes. For this purpose, it is of needs to aggregate information on the location of the various sources and their dependencies in order to make local and cross-project statements about effects of changes. Regarding the preparation of data, a project hierarchy is the main dimension. The top level, which is connected directly with a working copy root, represents the highest level of aggregation. The lowest level corresponds to externals definitions interpreted as the coarse-grained cross-project dependencies. This requirement for hierarchical distribution lays the foundation for acquiring adequate metadata on change related knowledge across project boundaries. In addition, this allows us to improve our redundancy resolution (cf. Def. 3.17) on working copies by adapting to our *locutor* metadata registry (cf. Sec. 4.2.2).

4.1.1 Why use Metadata?

According to [61], metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. Metadata is often called data about data or information about information. There are three main types of metadata:

Descriptive metadata describes a resource for purposes such as discovery and identification. It can include elements, for example, such as title, abstract, author, and keywords.

Structural metadata indicates how compound objects are put together, for example, how pages are ordered to form chapters.

Administrative metadata provides information to help manage a resource, such as when and how it was created, file type and other technical information, and who can access it. There are several subsets of administrative data; two that sometimes are listed as separate metadata types are *Rights management metadata*, which deals with intellectual property rights and (1) *Preservation metadata*, which contains information needed to archive and preserve a resource.

Metadata can describe resources at any level of aggregation. It can describe a collection, a single resource, or a component part of a larger resource. Metadata can be embedded in a digital object or it can be stored separately. Metadata is often embedded in HTML documents and in the headers of image files. Storing metadata with the object it describes ensures the metadata will not be lost, obviates problems of linking between data and metadata, and helps ensure that the metadata and object will be updated together. However, it is impossible to embed metadata in some types of objects, for example, due to access rights. Also, storing metadata separately can simplify the management of the metadata itself and facilitate search and retrieval. Therefore, we consolidate metadata in a *metadata registry* and link to the objects described.

A metadata registry is a datahouse of data about data. The purpose of the metadata registry is to provide a consistent and reliable means of access to data. The registry itself may be stored in a physical location or may be a virtual database, in which metadata is drawn from separate sources.

Note 4.1 (Repositories vs. Registries). For several years people have been using the terms registry and repository inconstantly, imprecisely and almost interchangeably. The author would like to take the opportunity to weigh in as to how these terms could be used more precisely. First let us take the definition of a repository. Webster defines a repository as "... a place, room, or container where something is deposited or stored ...". Note that here is nothing in this definition about the quality of the things being stored or the process to check to see if new incoming items are duplicates of things already in the repository. On the other hand let us take the word registry. A registry has the connotation of more than just a shared dumping ground. Registries have the additional capability to create workflow processes to check that new metadata is not a duplicate (for a given namespace). One of the definitions from Webster is "an official record book". A repository is similar to a front-porch of a house. No locks prevent new things from landing there. But a registry is a protected back room where human-centric workflow processes are used to ensure that items are non-duplicates, precise, consistent, concise, distinct, approved and unencumbered with business rules that prevent reuse across an enterprise. Registries have the implicit connotation of trust behind them and that is why we are consolidating metadata in a registry rather than a repository.

Note 4.2 (Registries vs. Database Indices). Apart from the distinction between registries and repositories, we can also compare registries with database indices. In computer science, an index is a data structure that facilitates fast and accurate information retrieval. Database indices are typically used to speed up data retrieval operations on a database table at the cost of slower writes and increased storage space. They are implemented by creating specialized database tables, which point to the original database tables and provide rapid random lookups and efficient access of a specific group of items in these database tables. If a database index is removed, the original database table and its items remain. In the scope of our MoC cake, a registry is an index that organizes and directs to information stored in a repository, i.e. to working copies and their external definitions. A registry is represented as a list of pointers that indicate the actual location of those working copies and their externals on the local file system. The location information is used to reduce file system space by replacing redundant directories with symbolic file system links. Consequently, both concepts, database indices and registries, provide a compact view onto an initial data structure — database tables and file system entries, respectively. In the following, we focus on registries since we are working with file system trees rather than database tables. Nevertheless, future work could verify whether the technique of database indexing can be applied to registries, i.e. whether a corresponding indexing of the registry entries based on working copy root paths can improve a registry's lookup and resolution processes (cf. Sec. 4.2.3).

So, to sum up, we use the concept of a registry to consolidate metadata. But what does metadata do for change management? An important reason for creating descriptive metadata is to facilitate discovery of relevant information. In addition to resource discovery, administrative as well as structural metadata can help organize working copies in sense of facilitating management of interoperability and resource integration. Consolidation of metadata serves the same functions in resource discovery as good cataloging does by (1) allowing resources to be found by relevant criteria, (2) identifying resources and their interrelations, (3) bringing similar resources together, (4) distinguishing dissimilar resources, and (5) giving location information. As this information is assumed for catalogers to provide a quick and good service on book requests of their customers, the same holds for a sophisticated change management to support fast and good — in sense of consistent — services with regard to collaborative authoring processes by employees of a company. Therefore, as the number of resources grows exponentially, aggregation is increasingly useful to keep to the overview.

But how do you gather the necessary data? Two commonly used approaches are cross-system search and metadata harvesting. Regarding the first approach, the metadata remains in the source repository, but the local search system accepts queries from remote search systems. The best-known protocol for cross-system search is the international standard Z39.50 [158], which is being modernized for the web environment. We follow the contrasting approach taken by the Open Archives Initiative [108]: all data providers have to translate their native metadata to a common core set of elements and expose this for harvesting. A search/registration service then gathers the metadata into a consistent designated data store to allow cross-project maintenance. We call this designated data store the *locutor metadata registry* (cf. Sec. 4.2.2), a datahouse representing a network of working copies with dependencies on the coarse-grained level.

However, before we tackle the development and profit of our *locutor* metadata registry, we first give a short history of commonly accepted metadata schemes used by data providers to not make any further translations.

4.1.2 A Short History of Metadata Schemes

Metadata schemes (*aka.* schemata) are sets of metadata elements designed for a specific purpose, such as describing a particular type of information resource. The definition or meaning of the elements themselves is known as the semantics of the scheme. The values given to metadata elements are the content. Metadata schemes generally specify names of elements and their semantics. We will now look at the most common and most widely used metadata schemes. This overview also serves as an evaluation regarding the adequacy of the various schemes for our purposes, the maintenance of related working copies.

Dublin Core. The Dublin Core Metadata Element Set arose from discussions at a 1995 workshop sponsored by Online Computer Library Center (OCLC) and the National Center for Supercomputing Applications (NCSA). As the workshop was held in Dublin, Ohio, the element set was named the Dublin Core. The original objective of the Dublin Core was to define a set of elements that could be used by authors to describe their own web resources. However, because of its simplicity, the Dublin Core element set is now used by many outside the library community — researchers, museum curators, and music collectors to name only a few. There are hundreds of projects worldwide that use the Dublin Core either for cataloging or to collect data from the Internet; more than 50 of these have links on the Dublin Core Metadata Initiative (DCMI) website. The subjects range from cultural heritage and art to math and physics. Meanwhile the DCMI has expanded beyond simply maintaining the Dublin Core Metadata Element Set into an organization that describes itself as “dedicated to promoting the widespread adoption of interoperable metadata standards and developing specialized metadata vocabularies for discovery systems” [34].

The Text Encoding Initiative. The Text Encoding Initiative (TEI) is an international project to develop guidelines for marking up electronic texts such as novels, plays, and poetry, primarily to support research in the humanities. However, there are also elements defined to record details about how the text was transcribed and edited, how mark-up was performed, what revisions were made, and other non-bibliographic facts. Libraries tend to use TEI headers when they have collections of SGML-encoded full text. This SGML mark-up becomes part of the electronic resource itself. Since the TEI DTD is rather large and complicated in order to apply to a vast range of texts and uses, a simpler subset of the DTD, known as TEI Lite, is commonly used in libraries.

Metadata Encoding and Transmission Standard. The Metadata Encoding and Transmission Standard (METS) was developed to fill the need for a standard data structure for describing complex digital library objects. METS is an XML Schema for creating XML document instances that express the structure of digital library objects, the associated descriptive and administrative metadata, and the names and locations of the files that comprise the digital object. It provides a document format for encoding the metadata necessary for management of digital library objects within a repository and for exchange between repositories. Further work is in process on extension schemas for audio, video, and websites.

Metadata Object Description Schema. The Metadata Object Description Schema (MODS) is a descriptive metadata schema that is a derivative of MARC21[1] and intended to either carry selected data from existing MARC21 records or enable the creation of original resource description records. It includes a subset of MARC fields and uses language based tags rather than the numeric ones used in MARC21 records. In some cases, it regroups elements from the MARC21 bibliographic format. Like METS, MODS is expressed using the XML schema language. MODS elements are richer than the Dublin Core, its elements are more compatible with library data, and it is simpler to apply than the full MARC21 bibliographic format. With its use of XML Schema language, MODS offers enhancements over MARC21, such as linking at the element level, the ability to specify language, script, and transliteration scheme at the element level, and the ability to embed a rich description of components.

The Encoded Archival Description. The Encoded Archival Description (EAD) was developed as a way of marking up the data contained in finding aids so that they can be searched and displayed online. Finding aids differ from catalog records by being much longer, more narrative and explanatory, and highly structured. The EAD is particularly popular in academic libraries, historical societies, and museums with large special collections. Like the TEI Header, the EAD is defined as an SGML/XML DTD jointly maintained by the Library of Congress and the Society of American Archivists.

Learning Object Metadata. The IEEE Learning Technology Standards Committee (LTSC) developed the Learning Object Metadata (LOM) standard (IEEE 1484.12.1-2002) to enable the use and re-use of technology-supported learning resources such as computer-based training and distance learning. The LOM defines the minimal set of attributes to manage, locate, and evaluate learning

[1]Machine-Readable Cataloging (MARC) defines a data format that provides the mechanism by which computers exchange, use, and interpret bibliographic information. MARC became USMARC in the 1980s and MARC21 in the late 1990s [86].

objects. Within each category is a hierarchy of data elements to which the metadata values are assigned. The IMS Global Learning Consortium has developed a suite of specifications to enable interoperability in a learning environment based on the IEEE LOM scheme with only minor modifications.

To put it briefly, all of theses metadata schemes do not fit our needs regarding maintenance of related working copies. As stated before, on the coarse-grained level we have to describe a network of working copies for cross-project maintenance. The canonical features of versioned *fsp*-trees, i.e. local and remote working copy root locations as well as local and remote locations of externals definitions, are the nodes and edges of such a network. The fine granular level is constituted by the leaves of the respective file system tree and the documents, in particular. The listed standard metadata schemes only deal with documents themselves, though, neglecting inter-relations on the project level. In this regard, let us consider some of the schemes more closely.

Even that [36] declares the Dublin Core Metadata standard for the primary online metadata standard, so this is not suitable for an intuitive — in sense of self-explanatory — description of versioned *fsp*-trees. Arguably, Dublin Core Metadata elements, like `Publisher`, act somewhat artificial for the description of a repository location. And for the next hierarchy levels as working copy and externals definition there are also no intuitive elements present.

Regarding TEI modules we analyzed `tei_math`, `tei_xinclude` and `tei_dictionaries` for our purposes. The `change` element, for example, which summarizes a particular change or correction made to a particular version of an electronic text gave us valuable insights for the representation of changes. But even there are no elements for intuitive mark-up of version controlled *fsp*-trees. Analogous to Dublin Core one could also use the TEI element `Distributor` or `Publisher` to describe a repository location, but again for the next hierarchy level there are none adequate elements.

Under the condition to consider repositories as libraries, METS seems at first glance to be a reasonable candidate for our purposes. In METS, the structures of digital library objects as well as their locations can be described. In this context projects or working copies could be perceived as such. Even the approach packaging together descriptive, administrative, and structural metadata for objects within a digital library sounds appropriate for our needs. However, METS also refers only to the management of fine granular constituents. The structural description of a METS document, for example, describes the hierarchical structure that could be provided to the users of a digital object to navigate *within* the object. This feature we already gain from the characteristics of semi-structured documents and their mapping to *fsp*-trees. The navigation over project boundaries is neglected, though.

At this point, we cancel the detailed delineation of the outstanding metadata schemes and combine our analysis into one statement: all of these metadata schemes are suitable to enrich the structural hierarchy of semi-structured documents with semantic relations, to thereby simplify the management on the fine-grained level. However, at this layer of our MoC cake we are interested in the management of the coarse-grained constituents. Further investigations regarding utilizing these schemes on the fine-grained level will be carried out in Chap. 6.

In the following we discuss our identification of a minimal kernel of required coarse-grained metadata for version controlled *fsp*-trees.

4.2 A Designated Store for Metadata Harvesting

The creation of metadata automatically or by information originators who are not familiar with cataloging or vocabulary control can create quality problems. Mandatory elements may be missing or used incorrectly. Schema syntax may have errors that prevent the metadata from being processed correctly. Metadata content terminology may be inconsistent, making it difficult to locate relevant information.

The Framework of Guidance for Building Good Digital Collections [47] articulates six principles applying to good metadata:

- Good metadata should be appropriate to the materials in the collection, users of the collection, and intended, current and likely use of the digital object.
- Good metadata supports interoperability.
- Good metadata uses standard controlled vocabularies to reflect the what, where, when and who of the content.
- Good metadata includes a clear statement on the conditions and terms of use for the digital object.
- Good metadata records are objects themselves and therefore should have the qualities of archivability, persistence, unique identification, etc. Good metadata should be authoritative and verifiable.
- Good metadata supports the long-term management of objects in collections.

There are a number of ongoing efforts for dealing with the metadata quality challenge:

- Metadata creation tools are being improved with such features as templates, pick lists that limit the selection in a particular field, and improved validation rules.
- Software interoperability programs that can automate the “crosswalk” between different schemas are continuously being developed and refined.
- Content originators are being formally trained in understanding metadata and controlled vocabulary concepts and in the use of metadata-related software tools.
- Existing controlled vocabularies that may have initially been designed for a specific use or a narrow audience are getting broader use and awareness. For example, the Content Types and Subtypes originally defined for MIME email exchange are commonly used as the controlled list for the Dublin Core Format element.
- Communities of users are developing and refining audience-specific metadata schemas, application profiles, controlled vocabularies, and user guidelines. The MODS User Guidelines are a good example of the latter.

In general, however, quality is very difficult to define. The guiding principle that we use in the context of metadata is: *high quality metadata supports the functional requirements of the system it is designed to support*, which can be summarized as *quality is about fitness for purpose*.

Our purpose is change management on semi-structured documents and so, at this point, our mark of quality is to obtain metadata enabling us to make cross-project statements on change processes. Fortunately, we can avail ourselves of the first category, metadata creation tools: we use the *fsp*-tree encoded information on working copies directly provided by version control systems, i.e., we use the metadata made available by such systems — in this case interpreted as metadata creation tools — extract the relevant subset for coarse-grained maintenance, and consolidate these information within our *locutor* metadata registry.

4.2.1 Metadata for Versioned *fsp*-trees

What is a "relevant subset for coarse-grained maintenance"? As initially stated, for the coarse-grained maintenance of version controlled *fsp*-trees we need metadata that enables us to identify resources and their interrelations, to bring similar resources together, and to give location information. In essence, we want to build up a network of working copies which holds this information.

Regarding to the nodes of such a network, represented by working copy root directories, we need unique identifiers. Most metadata schemes usually include elements such as standard numbers to uniquely identify the work or object to which the metadata refers. The location of a digital object may also be given using a file name, URL or some more persistent identifier such as a persistent URL (PURL) or Digital Object Identifier (DOI). Persistent identifiers are preferred because object locations often change, making the standard URL (and therefore the metadata record) invalid. In addition to the actual elements that point to the object, the metadata can be combined to act as a set of identifying data, differentiating one object from another for validation purposes. We therefore utilize the local and (persistent) remote locations of each individual versioned *fsp*-tree. In combination this gives us a unique key for working copies. The local site, retrieved from the local file system and *fsp*-tree decoding (cf. Def. 3.1), respectively, gives us information on where a particular working copy is stored. The local working copy path is the absolute path from the file system root to the working copy root. The remote location, the repository root URL in combination with the remote working copy path, gives us information from where the respective working copy is retrieved, say whatever repository. Such a fully qualified remote working copy path is retrieved from the repository mapping of the respective working copy. In addition, we need information about the dependencies between the respective working copies. A working copy depends on another working copy if there exists an externals definition between these two, i.e. if the one working copy maps a containing directory to the repository path corresponding to the working copy root path of the second one (cf. Ex. 4.1). The following definition clarifies this statement.

Definition 4.1 (Dependency). Let $\mathcal{T}$ be a *fsp*-tree. A working copy $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle \in \mathbb{W}_{\mathcal{T}}$ **depends** on a working copy $m = \langle \mathcal{H}', \omega, \mathbb{A} \rangle \in \mathbb{W}_{\mathcal{T}}$ via an external e ($w \overset{e}{\longleftarrow} m$) *iff* there exists an externals definition $\xi(v) \in \mathbb{X}_{\mathcal{H}}$ such that $w|_e \gg m$ with $e \in \xi(v)$. We call w the **dependant** and m the **provider**.

It should be noted again that we are here interested in coarse-grained relationships between working copies rather than inter-relations and intra-relations between the documents of a working copy. Because of this, we interpret externals definitions as such relations. The set of externals of a working copy $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ is retrieved by $\mathbb{X}_{\mathcal{H}}$ and, again, the fact that $w|_e \gg w|_e$ is restricted in the realization part of this work (cf. Part III).

Example 4.1. To sharpen our intuition on a network of working copies and their dependencies, let us consider the abstract *fsp*-tree $\mathcal{T}$ depicted on the left side of Fig. 4.1. The right side illustrates the corresponding network of working copies. Here we have working copy a depending on b via external e_2. Working copy b and c depend on a via e_1 and e_3, respectively. Working copy e depends via both e_4 and e_5 on d, and f depends on itself via e_6. ♦

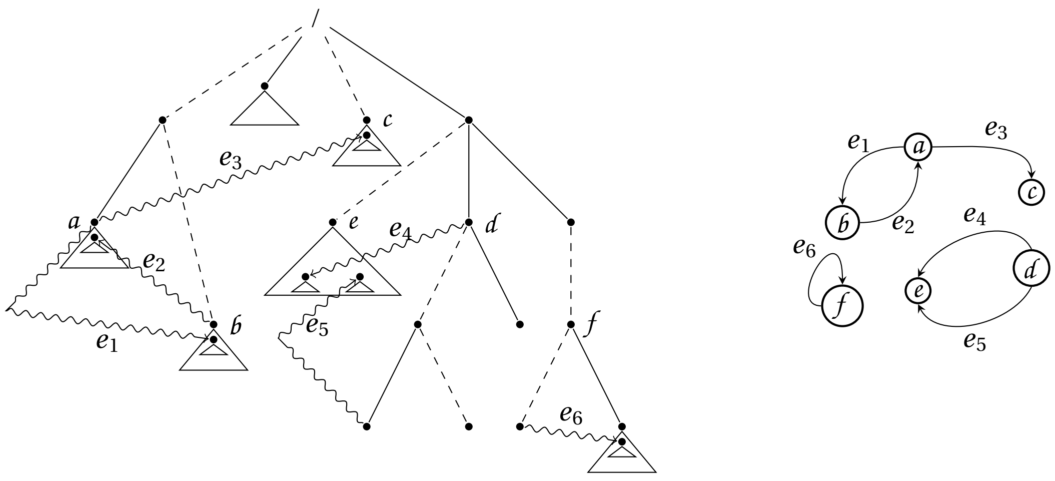


Figure 4.1: A Network of Working Copies.

Obviously, a network of working copies is far more concise than the corresponding *fsp*-tree and thus in order to obtain the same information from a *fsp*-tree as from a network is far more complex. Ergo, a *fsp*-tree as such is not suitable for an *efficient* analysis regarding redundancy resolution and rippling of effects. In addition, equal to redundancy resolution (cf. Def. 3.17), the identification of $\mathbb{W}$ and $\mathbb{X}$ are very time consuming tasks. For both problems, we will provide shortly an adequate and effective solution (cf. Sec. 4.2.2).

Now we know why and what metadata we need in order to build up such a network and we can assure that *fsp*-trees provides us with all the information on a working copy for coarse-grained maintenance: the local and remote working copy location and the externals definitions representing interrelations between working copies. Hence, following the METS approach, in sense of packaging together descriptive, administrative, and structural metadata for objects within a digital library, we identify the following metadata for maintenance of versioned *fsp*-trees:

Repository Root URL retrieved from the repository mapping of the respective working copy. SUBVERSION is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a file system tree. A repository root URL is the root of this file system tree.

Remote Working Copy Path retrieved from the repository mapping of the respective working copy. The remote working copy path is the path of the working copy relative to the repository root.

Local Working Copy Path retrieved from local file system and *fsp*-tree decoding (cf. Def. 3.1), respectively. The local working copy path is the absolute path from the file system root to the working copy root.

Externals Definitions retrieved from the respective set of externals, where each externals is identifies by

Owner denotes the node of the working copy externals definitions are defined on

Revision denotes the state of the repository file system tree

Remote Working Copy `URL` denotes the absolute remote repository path of the referenced working copy.

Name the local sub-directory relative to owner the externals definition is defined on.

With these data on working copies and their coarse-grained relationships to each other, we can now consolidate these information in the space provided by the *locutor* metadata registry. This metadata also meet our quality standards so that they allow us to build up the desired network of working copies.

4.2.2 The *locutor* Metadata Registry

Metadata registries are an important tool for managing metadata. They provide information on the origin, source, and location of data. Registration can apply at many levels, including schemes, usage profiles, metadata elements/content, and code lists for element values. A metadata registry provides an integrating resource for proprietary/legacy data and acts as a lookup tool for documents. Registries can document multiple element sets, particularly within a specific field of interest.

Recalling, from our point of view a metadata registry is a datahouse, a database and a data warehouse at the same time. It stores data about data conducive to respond to analysis questions. In our case our specific field of interest is consolidating data about versioned controlled *fsp*-trees and their interrelations to improve the change management process (cf. Sec. 1.3.2). Hence, our analysis questions are related to redundancy resolution and rippling of effects of changes. For both we need locality information to be able to identify redundancies and to determine if and where potential effects have to be propagated.

On the conceptual level our *locutor* metadata registry is a semantic consolidation aggregating the in Sec. 4.2.1 introduced metadata on version controlled file system trees. On the pragmatic level it is a mapping from working copy root paths to contained externals definitions. The next definition specifies the concept.

Definition 4.2 (*locutor* Metadata Registry). Let $\mathcal{T}$ be a *fsp*-tree. We call the relation

$$\mathbb{REG}_{\mathcal{T}} = \langle \mathbb{W}_{\mathcal{T}} \times \wp(\mathbb{X}_{\mathcal{T}}), \text{Graph}(\mathbb{REG}_{\mathcal{T}})\rangle$$

a ***locutor* metadata registry**, short registry, where

$$\text{Graph}(\mathbb{REG}_{\mathcal{T}}) = \left\{ (w, \mathbb{X}_{\mathcal{H}}) \;\middle|\; \begin{array}{l} w = \langle \mathcal{H}, \omega, \mathbb{A}\rangle \in \mathbb{W}_{\mathcal{T}} \textit{ and} \\ \xi(v) \textit{ admissible for all } \xi(v) \in \mathbb{X}_{\mathcal{H}} \end{array} \right\}$$

We usually omit the index of $\mathbb{REG}$ unless context requires and model operations like *register*, *unregister*, and *is_registered* straightforward by the respective set operations $\cup$, $\setminus$, and $\in$.

Regarding Ex. 4.1, the registry exactly describes the network of working copies depicted on the right side of Fig. 4.1. For instance, let us assume for the *fsp*-tree $\mathcal{T}$ depicted on the left side of Fig. 4.1 the working copy $a = \langle \mathcal{T}|_a, \omega, \mathbb{A}\rangle$ with the external $e_2 = \langle \pi, \Theta\rangle$ and $\omega(\mathcal{T}.a) = \Psi$. For working copy $b = \langle \mathcal{T}|_b, \omega, \mathbb{A}\rangle$ let us assume the external $e_1 = \langle \phi, \Psi\rangle$ and $\omega(\mathcal{T}.b) = \Theta$. The two working copies being registered are described by the pairs $\langle a, \{\{e_2\}_{v_1}\}\rangle \in \mathbb{REG}$ and $\langle b, \{\{e_1\}_{v_2}\}\rangle \in \mathbb{REG}$, respectively, where v_i denotes the respective owner of the set of externals. Thus we can identify the dependencies $a \overset{e_2}{\longleftarrow} b$ and $b \overset{e_1}{\longleftarrow} a$ due to $a|_{e_2} \gg b$ and $b|_{e_1} \gg a$. This also demonstrates the fact that a working copy in one dependency relation can take the role of the dependant and in another the role of the provider. All other dependencies are identified in the same way, so, for example, with similar assumptions to e and d we can identify the dependencies $e \xleftarrow{e_4} d$ and $e \xleftarrow{e_5} d$ for the registered working copies $\langle e, \{\{e_4\}_{v_1}, \{e_5\}_{v_2}\}\rangle \in \mathbb{REG}$ and $\langle d, \emptyset\rangle \in \mathbb{REG}$, respectively.

With the introduction of a registry, it is now possible to store metadata in such a way as the analysis requires and not as the operating system makes it available. Our *locutor* metadata data registry is a semantic consolidation considering metadata in the context of the entire knowledge base, i.e. the file system trees put under version control. From the technical side, a consolidation process is necessary whenever the knowledge base has been changed or extended by any (automated) process. Therefore in order to keep up keep consistency, we utilize the journaling functionality of the underlying version control system.

In fact through to our consolidation process there arises a second pool of data (redundant data!) but now the data is pre-compressed and in accordance with user friendliness as well as with change management. Regarding the latter, this means local identified modifications on a document can not only propagated within the document itself and dependants in the same working copy but can be propagated to depending ones hosted in different working copies, i.e. in case of registered dependants the propagation can directly be accomplished on the client side. Regarding user friendliness, consolidation represents a concise overview of existing working copies and their relationships to each other. For example, in order to retrieve all registers working copies within a registry $\mathfrak{R}$ we define the set $\mathbb{W}(\mathfrak{R}) = \{w \mid w = \text{proj}_1(r) \textit{ for all } r \in \mathfrak{R}\}$. This allows us to replace a script like `update-local.sh` (cf. Fig. 3.10) through integrating the notion of a registry into the appropriate version control commands. In turn, we can restrict these commands to work on registered working copies only and thus speed up both the lookup for working copies within a *fsp*-tree and to be performed redundancy resolutions.

In the following we solve the in this work multiple-addressed problem of identifying the sets $\mathbb{W}$ and $\mathbb{X}$ with regard to redundancy resolution on *fsp*-trees by adapting the concept of redundancy resolution on registries.

4.2.3 Redundancy Resolution on Registered *fsp*-trees

We want to begin this section with the question why a notion of consolidation and redundancy resolution is such an important point? The task of redundancy resolution comprises the detection and resolution of duplicates and therefore prevents inconsistencies from data collections and in turn improves the quality of data. Data quality problems are present in single data collections, such as files and databases, e.g., due to misspellings during data entry, missing information or other invalid data as well as when multiple data sources need to be integrated, e.g., in data warehouses, federated database systems or global web-based information systems. This is because the sources often contain redundant data in different representations. In order to provide access to accurate and consistent data, the need for consolidation of different data representations and the need for elimination of duplicate information increases significantly — especially when integrating heterogeneous data sources. Therefore a datahouse, as we have defined it, requires extensive support for data cleaning. Huge amounts of data are continuously refreshed from a variety of sources so the probability that some of the sources contain "dirty data" is high. Furthermore, datahouses — not only within change management — are used for decision making, so that the correctness of their data is vital to avoid wrong conclusions. For instance, duplicated information will produce incorrect or misleading statistics.

Due to the wide range of possible data inconsistencies and the sheer data volume, we consider consolidation and redundancy resolution to be the fundamental slices of our MoC cake. Hence, in order to provide a consistent and high-performance MoC base, we adapt redundancy resolution on registries.

Definition 4.3 (Redundancy Resolution). A **redundancy resolution** is a function *resolve*: $\mathbb{FS}^+ \times \mathbb{REG} \to \mathbb{FS}^+$: $\mathscr{T} \times \mathfrak{R} \mapsto resolve(\mathscr{T}, \mathbb{W}_{\mathfrak{R}})$ where $\mathbb{W}_{\mathfrak{R}} = \{w|_e, m \mid w \overset{e}{\longleftarrow} m \textit{ for all } w, m \in \mathfrak{R}\}$.

Note the fact that the nested working copy problem (cf. Def. 3.18) is adapted accordingly. However, set-theoretically one might think of the difficulty to find the smallest registry $\mathfrak{R}$ such that there is no pairwise distinct tuple $\langle w, m\rangle \in \mathbb{W}_{\mathfrak{R}}$ such that $m \gg w$.

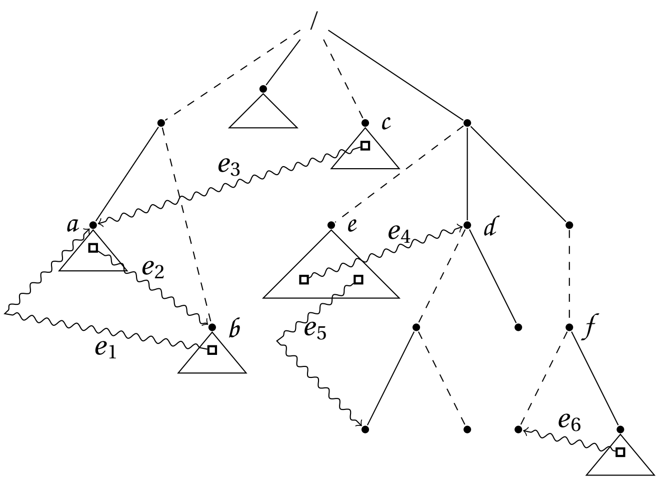


Figure 4.2: A Resolved *fsp*-tree wrt. the Registry Depicted on the right side of Fig. 4.1.

If we now again take up Ex. 4.1, the benefits become obvious. From the chaos of working copies we can not only distill a compact and consistent overview in terms of our *locutor* metadata registry but with the adaption of redundancy resolution on registered *fsp*-trees we can provide an efficient detection and resolution of duplicates: redundancy resolution makes the network of working copies small meshed and in turn the “holes” on the file system coarse meshed.

To sharpen our intuition, we illustrate the resulting *fsp*-tree in Fig. 4.2. Redundant externals have been resolved by s-nodes reverting the dependency relation to illustrate the corresponding symbolic file system links. We call such symbolic file system links *transformed externals* (TransEX).

To emphasize *raison d'être* of transformed externals the author performed the following case study on the *locutor* system implementing redundancy identification as well as resolution on file system level: mirroring all local working copies in two distinct directories a disk usage evaluation on the first mirror » `du -sh ~/svn` « resulted in 7.2GB of disk usage, but a disk usage evaluation on the second mirror » `du -sh ~/locutor` « resulted in only 3.7GB after a redundancy resolution.

4.3 Conclusion

We have presented a framework for maintenance of coarse-grained relationships between version controlled *fsp*-trees. In turn, this gives us a sophisticated foundation for an efficient change impact analysis across project boarders and redundancy resolution. The latter we revised from redundancy resolution on *fsp*-trees through to redundancy resolution on registered *fsp*-trees. Employing version controlled metadata we enabled to register working copies and their interrelations represented by externals definitions. Consolidated in the *locutor* metadata registry enables us to set up a network of working copies reducing the complexity of both redundancy identification as well as maintenance of coarse-grained dependencies necessary for change management on collection on documents hosted in different working copies.

In addition, our consolidation process improves common version control workflows in sense of making `commit` and `update` commands much faster. Working copies freed from redundancy result in less disk usage and in turn less versioned directory tress to be checked for modifications.

Acknowledgments. While the work presented in this chapter is original, it owes much to discussions with Michael Kohlhase. Furthermore, the discussions with Alexander Sinyushkin and Alexander Kitaev were very fruitful. Both are members of TMate Software, developing the SVNKIT library [131] and planing to integrate the concept of a *locutor* metadata registry into their SVNKIT client.

Chapter 5

Semantic Differencing

Just because something is different, doesn't mean anything has changed!

As initially discussed, digital documents have become a major pillar of today's business. Many of these documents are shared by a team of authors, which requires strong means of version control to support collaboration. Otherwise, teamwork is limited to sequential editing, or rigorous coordination efforts must be agreed on and manually maintained. Both is unacceptable for larger teams.

Version control is a major enabling factor for collaboration which has been extensively studied in Chap. 3. The discussed version control systems sufficiently support collaboration on the directory tree structure level but are rather weak on the file content level. In order to enable collaborative editing, a version control system must support document type specific conflict detection and merge operations. Unfortunately, up to now semi-structured documents are handled as simple text documents. This severely hinders collaborative editing because for those document types most version control features such as merging and conflict detection are limited to text lines rather than considering the structure.

5.1 Introduction

Version control systems utilize an internal repository for versioning to ensure data integrity and traceability of all changes on version controlled directory trees. Here, however, we are interested on versioning on the file content level, i.e. *fsp*-tree encoded XML documents rather than on *fsp*-trees in general.

On the file content level, two versions of a file are distinguished by a *delta* (*aka. patch*), which covers the modified parts of a file, i.e. the changed lines. This fact is exploited by state-of-the-art version control systems in order to save disk space. Instead of archiving each version of a file completely, version control systems compute the delta between two versions and store only this one in the repository. As an "anchor" for the delta mechanism the repository requires a complete version of each file. All other versions can be derived from the anchor by applying the delta (*aka. patching*). Depending on whether the very first or the latest version is saved completely, it is called *forward* or *backward* delta.

Regardless of the direction of the delta mechanism, the question remains how differences between two versions are determined. The `diff` tool, one of the oldest UNIX commands (was included around 1976), is commonly used to detect differences between two versions. The algorithm compares the contents of the two files *source* and *target* (modified version) and produces a delta — lines that are changed or absent in either of files[1]. It was written by Hunt and McIlroy and based on the algorithm for file comparison that they defined in [65]. The algorithm considers lines indivisible and computes the so called "longest common subsequence of lines" (LCS). Then anything not in this LCS is declared to belong to the difference set — the minimal set of lines that needs to be changed for the transformation of source to the target.

The disadvantage of this approach is the assumption that in a classic UNIX `diff` both files are text-based (*aka.* flat), i.e. the deltas are determined solely on the basis of lines. Version control systems, like SUBVERSION, utilizing this delta mechanism, however, need to manage *all* types of files in a repository. This also applies to our concern here: semi-structured documents.

5.1.1 Why use Structure-Aware Differencing?

Using the UNIX `diff` tool on XML documents to find out what has been changed between two versions of a document immediately demonstrates the basic problems: (1) changes in the order of XML attributes generate spurious deltas, (2) an XML attribute present in one document with a default value but absent and defaulted from the schema in another document generate spurious deltas, (3) some elements may appear in any order and so a change to order should not be identified within the delta, (4) it is difficult to know where in the XML tree structure a change has happened when the change is represented by line numbers, and (5) changes in white space within elements generate spurious changes.

As another example, none-XML but semi-structured, let us assume a JAVA program containing at least two structures. While editing this document, one could, accidentally or due to some reasons, insert a newline before the first structure. From the point of view of JAVA parsers this operation does not change the document — white spaces between structures are insignificant. In contrast, SUBVERSION will consider the new version different from the first one, since all lines in the second document have been shifted by one. Thus, essentially we have two equivalent structural documents but SUBVERSION will create a new delta to represent the difference — what is fine so far but not if you want information on semantic changes.

This results from the fact that the basic entities utilized to locate changes are not related to any basic semantic entity. The UNIX `diff` uses text lines as *the* structuring mechanism for all documents but while comparing XML documents one wants to compare logical differences in the XML nodes not just differences in text. Ideally, it should be possible to ignore the insignificant differences and identify only "real" changes.

Canonical XML [18] seeks to address some of these issues by specifying a defined way to write an XML file such that files that contain exactly the same information (or as the specification says, "that are logically equivalent within an application context") will be represented by the same sequence of characters. Canonical XML recognizes "that two documents may have differing canonical forms yet still be equivalent in a given context based on application-specific equivalence rules for which no generalized XML specification could account". These issues are important in actual use cases and might include one or more of the following: (1) the use of ID attribute may be important, because these are defined as unique within a document, and are often used as internal

[1]For completeness, it should be mentioned that the UNIX `diff` tool can not only compare two files, it can walk entire directory trees, recursively checking differences between subdirectories and files that occur at comparable points in each tree: `diff -rq dirA dirB`.

pointers. Therefore if two documents are the same in terms of their structure, attributes and elements, except that the ID values are different (but consistent as internal pointers) then it may be reasonable to consider that the two documents are still the same, (2) it is often the case that in a document some information, for example metadata, may be presented in any order, again with no semantic difference, (3) differences in comments and processing instructions again do not constitute changes to the actual XML documents, and may or may not be important as a difference to a user, and (4) a specific XML format may allow different representations for the same information.

In summary, canonical XML provides a way to ensure two XML documents are the same, but provides almost no help when they are different[2]. Consequently, differencing tools that work on flat documents are no longer appropriate. New tools are needed to identify changes on semi-structured documents and new methods are needed to represent these changes.

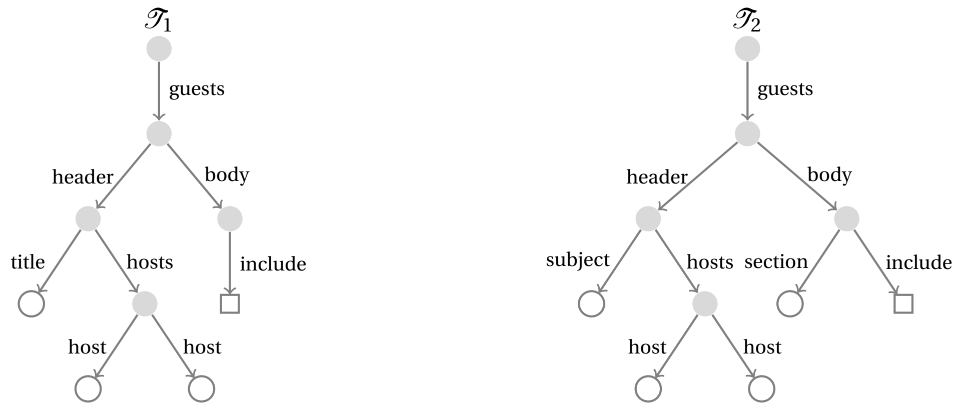


Figure 5.1: Two Versions of one XML Document.

The problem of finding the changes between two XML documents can be seen as the *tree-to-tree correction problem* [10] for ordered labeled trees. Consider the two trees in Fig. 5.1. We wish to apply a sequence of operations on $\mathcal{T}_1$ to create $\mathcal{T}_2$. The most basic operations we can apply are: (1) change the label of a node, (2) delete a leaf node, and (3) insert a leaf node. We will call a sequence of such operations an *edit script*: An edit script S between $\mathcal{T}_1$ and $\mathcal{T}_2$ is a sequence of edit operations turning $\mathcal{T}_1$ into $\mathcal{T}_2$. Assuming that we are given a cost function defined on each edit operation, the cost of S is the sum of the costs of the operations in S. An *optimal edit script* between $\mathcal{T}_1$ and $\mathcal{T}_2$ is an edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ of minimum cost and this cost is the *tree edit distance*. The *tree edit distance problem* (*aka.*tree-to-tree correction problem) introduced by Tai [135] as a generalization of the well-known *string edit distance problem* [144] is to compute the edit distance and a corresponding edit script. The set of edit scripts which transform $\mathcal{T}_1$ into $\mathcal{T}_2$ is infinite; we could continuously add and delete nodes. However we want to find a minimal edit script which transforms $\mathcal{T}_1$ into $\mathcal{T}_2$. An example edit script to change the tree on the left side of Fig. 5.1 into the tree on the right side could be: (1) add `section` as the first child of node `body`, and (2) relabel the first child of node `header` from `title` to `subject`.

To be of more value to users, we need to be able to show the edit script in a more intuitive, visual and immediately discernible format. An obvious and flexible method of solving this problem is to change the edit script into a format which is valid XML and can be used by other programs. This format can then be modified, e.g. by an XSLT transformation into a format which displays the changes in a form which is easy to read by users.

[2]We investigate the benefits of canonical XML further in Sec. 5.2

There are many possible applications for XML differencing tools, some conceivable uses are:

Versioning As just illustrated, Version Control systems covering XML would greatly benefit from an XML *diff* tool. Although the UNIX `diff` utility will produce valid output for XML files, the output will be suboptimal in comparison to a *diff* utility cognizant of the hierarchical structure of the data. A hierarchical delta would also capture the "essence" of any changes much better than the line-oriented *diff*, i.e. it would express what the users intentions were when modifying the file.

Document Comparison and Updating XML documents written by an author or co-author can be checked to find the changes between versions. Patches can be distributed containing the changes made by an author, and others can choose whether or not they wish to apply the changes to their copy of the document.

Databases XML is increasingly used for storing data in databases. Detecting changes to data is important for many database applications. The XYDIFF [155] program was developed specially for the XYLEME [156] data warehousing project. For example if the database returns XML documents for query results, we can identify the nature of any changes to a standing query e.g. detect when a new name is added to a mailing list.

Web Caching Currently web caches must request complete documents if they do not hold a current version of the requested page. Using a differencing utility, they need only request a delta between the cached page and the new page. This could create a large reduction in the amount of web traffic, and result in improved transfer times for users. Such a system could cache dynamic as well as static web objects. See [69] for more information.

Transaction Data If a user has a common query against an application it would be possible to send only a delta of any changes to the previous query result rather than send the complete document again. For example a sports ticker application could send information on only the current event (e.g. a goal being scored, a yellow card given), rather than send the full account of the match to date. This can result in significant bandwidth savings.

In the following we give an overview of existing XML *diff* tools and their various output formats. We analyze the usability in sense of the minimality of the calculated edit scripts and their representations. Based on this analysis, we create our list of requirements XML *diff* tools must comply with in order to be applicable to *all* the applications listed here.

5.1.2 A Short History of File Comparison Utilities

Most previous work in change detection focused on computing differences between flat files. As already mentioned, the UNIX `diff` utility is the most famous one utilizing the LCS algorithm [106] to compare two plain text files. Our exemplary justification that this delta computation is not applicable to semi-structured documents (cf. Sec. 5.1.1), is encouraged by Chawathe et al. [23]. He pointed out, that these techniques cannot be generalized to handle structured data because they do not understand the hierarchical structure information contained in such data sets. Typical hierarchically structured data, e.g. XML, place tags on each data segment to indicate context. Standard plain-text change-detection tools have problems matching data segments between two versions of data.

In order to design an efficient algorithm to detect changes on XML documents, we first need to understand the hierarchical *structure* in XML. Fortunately, the increasing use of XML over the last years has motivated the development of many differencing tools capable of handling tree-structure documents. All of these tools work with either completely ordered trees [128, 135, 159, 63, 32, 26] or completely unordered trees [160, 145].

For the unordered case, it turns out that all of the problems in general are NP-hard. Indeed, the tree edit distance problem is even MAX SNP-hard [4]. However, under various restrictions, or for special cases, polynomial time algorithms are available. For instance, if a structure preserving restriction on the unordered tree edit distance problem is imposed, such that disjoint subtrees are mapped to disjoint subtrees, it can be solved in polynomial time. In [128] Selkow considered another approach to the ordered edit distance problem where insertions and deletions are restricted to leaves of the trees. This edit distance is sometimes referred to as the *1-degree edit distance*. He gave a simple algorithm using $O(|\mathcal{T}_1||\mathcal{T}_2|)$ time and space.

For the ordered version of the problems polynomial time algorithms exist. These are all based on the classic technique of dynamic programming [59] and most of them are simple combinatorial algorithms.

We now pick out some prominent algorithms and tools to analyze the different solutions. We study both the detection and the representation of changes. The role of a *diff* algorithm is twofold: first it matches nodes between the two versions of the documents. Second it generates a document, namely an edit script, representing a sequence of changes compatible with the matching. Some of the algorithms are able to detect *moves* between the two documents, whereas others do not. Note the fact that most formulations of the change detection problem with *move* operations are NP-hard [161]. So the drawback of detecting *moves* is that such algorithms will only approximate the minimum edit script, whereas most algorithms on ordered trees provide a minimal editing script in quadratic time. The improvement when using *move* operation is that in some applications, users will consider that a move operation is less costly than a *delete* and *insert* of the subtree. In general, *move* operations are important to detect from a semantic viewpoint because they allow to trace nodes through time better than *delete* and *insert* operations. The semantic of *move* is to identify subtrees even when their context has changed. Some algorithms consider more semantics in XML documents, for instance, they may consider keys (e.g. `ID` attributes in the DTD) and match with priority two elements with the same tag if they have the same key.

Intuitively, the result of optimal algorithms (quadratic complexity) is slightly better when change rate is low (up to ten percent), and significantly better when the change rate is very high (more than thirty percent). However, algorithms supporting *move* operations can be very efficient when large subtrees have moved.

Tree Correction Algorithms

As stated above, the problem of finding the changes between two XML documents can be seen as the tree-to-tree correction problem. This section covers several algorithms created to solve this problem.

The Extended Zhang and Shasha Algorithm. Barnard, Clarke and Duncan's paper [10] gives a concise overview of early (pre-1995) work on the tree-to-tree correction problem. As the early work has largely been superseded by later algorithms and papers, we will not consider it here. However the paper also proposes an algorithm based on Zhang and Shasha's work [159] which we will refer to as the Extended Zhang and Shasha (EZS) algorithm. The original algorithm by Zhang and Shasha runs in time $O(n^2 log^2 n)$ for balanced trees [23], where n is the number of tree leaves (worse for unbalanced trees). The algorithm uses the following *primitives* (basic operations): (1) *change* the value of a node to a new value, e.g. replace the text of a sentence, (2) *delete* a leaf node, and (3) *insert* a leaf node. Barnard, Clark and Duncan extended Zhang and Shasha's algorithm by adding the following primitives which act on subtrees rather than just nodes: 1. *deleteTree* deletes a subtree, 2. *insertTree* inserts a subtree, and 3. *swap* swaps a subtree with another subtree. These operations were added to give better edit scripts for documents; they allow operations closer to those a user could be expected to perform, such as merging and moving whole sections of text at a time. The impact these extensions have on the overall time is relatively negligible compared to the benefits. Note that the EZS algorithm will always produce a edit script that is minimal in terms of the costs of the operations. This algorithm is implemented in the XMLDIFF [151] program.

The Fast Match Edit Script Algorithm. Chawathe, Rajaraman, Garcia-Molina and Widom's paper [23] covers the Fast Match Edit Script (FMES) algorithm. The FMES algorithm was created after the EZS algorithm and is intended to be complementary to it. The FMES algorithm uses the following primitives: (1) *insert* a new leaf node, (2) *delete* a leaf node, (3) *update* the value of a node to a new value, e.g. replace the text of a sentence, and (4) *move* a subtree from one parent to another. The algorithm splits the tree-to-tree correction problem into two parts: finding a good matching between trees (Good Matching problem) and finding a Minimum Conforming Edit Script (MCES). A description of the operation of the algorithm can be found in [23]. In order to achieve good performance for the algorithm, it is assumed that for a leaf l in a document, there exists at most one leaf in the other document which "closely" resembles l. This assumption allows the algorithm to perform efficiently, but in cases where this assumption does not hold it may not produce a minimal edit script. The FMES algorithm runs in order $O(ne + e^2)$ time where n is the number of tree leaves and e is the "weighted edit distance" (described in the paper). Because of the tradeoffs between performance and minimality of edit scripts, the authors suggest using the EZS algorithm in domains where the amount of data is small and the FMES algorithm in domains where there is a large amount of data. The FMES algorithm is also implemented in the XMLDIFF [151] program.

The xmdiff Algorithm. The XMDIFF algorithm presented in [22] is unique in that it defines an external memory algorithm which can handle arbitrarily long files. The paper is written by Sudarshan Chawathe, a co-author of [23], and represents some subsequent work he has carried out in this area. The following primitives are used by XMDIFF: (1) *insert* a leaf node, (2) *delete* a leaf node, and (3) *update* the value of a node to a new value. The algorithm uses the idea of edit graphs to reduce the problem of finding a minimum cost edit script to the problem of finding a shortest path from one end of the edit graph to the other. In an external memory algorithm the overriding performance factor is the number of input/output (I/O) operations. The algorithm can make use of surplus RAM to reduce I/O cost. Given a block size of S, input trees of size M and N respectively, $m = M/S$ and $n = N/S$, the costs are $\text{I/O} = 4mn + 7m + 5n$, $\text{RAM} = 6S$, and $\text{CPU} = O(MN + (M+N)S^{1.5})$.

There exist many other algorithms and papers on the tree-to-tree correction problem, which are not covered in depth here, but deserve to be mentioned. Cole, Hariharan and Indyk's paper [27] achieves an impressive time bound, but is heavily mathematical and it would take some time to create an implementation based on it. Chawathe and Garcia-Molina's paper [125] covers the MH-DIFF algorithm. They include primitives to move and copy entire subtrees, which as discussed in the EZS algorithm, can lead to more appropriate deltas for documents. Their work covers unordered trees only which are not always applicable to XML documents.

Existing Products

There are several existing products for finding changes between XML instances. All of these tools are designed to take two XML files as input and somehow display the changes between them. IBM's XML Diff and Merge Tool [152] is not covered as it is not designed to produce standalone delta files. Instead the program highlights the differences within a JAVA graphical user interface (GUI). However, IBM's other product XML TreeDiff [153] is considered.

DeltaXML [37] is proprietary software created by Monsell EDM Ltd. Interestingly it can handle both ordered and unordered trees. If a DTD is present it is used to obtain entity expansions and default attribute values. Output is either a delta or the original document with changes tagged.

xmldiff [151] is GPL-licensed free software created by Logilab as part of the NARVAL project. The program was written in PYTHON and implements the FMES and EZS algorithms. It has two output formats for deltas, one of which is not in XML format and the other is in the XUpdate [79] language (discussed below). The program needs to hold the XML files in an internal structure in memory, hence it cannot handle very large files. Also there are several cases where the program produces incorrect output, due to coalescing of text nodes in XPath expressions.

XML TreeDiff [153] is proprietary software created by IBM. The program was written as a set of JAVA Beans intended to mimic the functionality of the traditional UNIX `diff` and `patch` programs. It purportedly achieves good performance by the use of "fuzzy subtree matching". The program has two output formats, FUL and XUL, of which we consider XUL later, as XUL is the successor of FUL.

XyDiff [155] was developed by the VERSO team for the french national institute for research into data processing and automation (INRIA). The program was developed for the XYLEME XML data warehousing project. The utility uses the XERCES [150] C++ parser. At its heart is a very fast algorithm able to difference large (> 10Mb) documents. However the algorithm often produces non-minimal output. XYDIFF was released under the open source Q Public License.

diffmk [40] is a PERL program written by Norman Walsh at Sun Microsystems. Although the source code is available, it does not appear to have an Open Source license and remains the copyright of Sun Microsystems. The program uses a PERL algorithm for computing the LCS of two strings. It does not always produce minimal, or even correct output. The output is the original document with changes marked. It is distributed with a utility which displays the differences between the files using colors in a way which is easy to read by humans.

XML Diff and Merge Tool [35] is proprietary software created by Dommitt Inc. There is no downloadable evaluation, only an on-line demonstration which invites the user to upload XML files. It uses the XMDIFF algorithm. The output is the original document with changes marked.

VM Tools [141] package contains XML differencing and patching tools. The tools are written in JAVA and have a defined API for integration with other JAVA programs. The package is released under their own VM Systems software license. VM Tools does not support differencing of XML processing instructions or comments, nor does it have support for large files.

X-Diff [149] is a tool for detecting the difference between two XML documents. It was originally developed in the Niagara Query Engine [107]. Unlike other XML *diff* tools, X-DIFF uses an unordered model (only ancestor relationships are significant) to compute difference between two XML documents. The authors believe that this approach is more suitable for most database applications and that by using an unordered model, change detection is substantially harder than using the ordered model, but the change result is more accurate. We could not further investigate this procedure due to the fact that the X-DIFF code is permanently offline.

Output Formats

All of the previously listed products have separate output formats. In this section we consider and contrast the best of them. The author has kept this section separate from the discussion of the products as the output formats can stand independent of their implementations. None of the output formats produce enough context information to produce accurate patches on files considerably different from those used in creating the delta. More useful context information would be, for example, showing any parent and sibling nodes. For the sake of clarity the examples given in this section have been indented and formatted; the reader should not expect the programs to produce identical output.

```
<xhtml:html xmlns:deltaxml="http://www.deltaxml.com/ns/well−formed−delta−v1"
     xmlns:xhtml="http://www.w3.org/1999/xhtml" deltaxml:delta="WFmodify">
<xhtml:html deltaxml:delta="WFmodify">
<xhtml:head deltaxml:delta="unchanged"></xhtml:head>
<xhtml:body deltaxml:delta= "WFmodify">
  <xhtml:table deltaxml:delta="WFmodify">
    <xhtml:tr deltaxml:delta="WFmodify">
      <xhtml:td deltaxml:delta="unchanged"></xhtml:td>
      <xhtml:td deltaxml:delta="unchanged"></xhtml:td>
      <xhtml:td deltaxml:delta="WFmodify">
        <deltaxml:PCDATAmodify>
          <deltaxml:PCDATAold>foo</deltaxml:PCDATAold>
          <deltaxml:PCDATAnew>bar</deltaxml:PCDATAnew>
        </deltaxml:PCDATAmodify>
      </xhtml:td>
    </xhtml:tr>
</xhtml:tablc>
</xhtrnl:body>
</xhtml:html>
```

Figure 5.2: A DELTAXML Output.

DeltaXML. An example of the DELTAXML output format is given in Fig. 5.2. The program has been used to produce a delta between two XHTML documents where the only change is that the text of a `<td>` element has been changed from `foo` to `bar`. Delta files produced by DELTAXML always have a namespace for DELTAXML associated with them. The DELTAXML format conveys change information in a non-complex fashion and precisely. However it does not make good use of XPath, and seems to contain a lot of redundant information (the unchanged nodes), yet does not provide the context information that is needed for patching changed files. A small number of attributes and elements are used in the delta file to provide all the information needed to represent changes. The attributes are added to existing elements. The additional attributes are: (1) `deltaxml:delta` to indicate how the containing element has been changed, and (2) `deltaxml:new-attributes` and `deltaxml:old-attributes` to show changes to attributes. The new delta elements are (1) `deltaxml:PCDATAmodify` to indicate a change to

PCDATA in an element, and (2) `deltaxml:exchange` to show one type of element exchanged with another, or an element exchanged with PCDATA. The attribute named `deltaxml:delta` can be found on many elements in the delta file. As noted above, it indicates why this element is present in the delta file, e.g. because it has been modified, added or deleted. Starting at the root element of the delta, this has a `deltaxml:delta` attribute which will have a value of `WFmodify` if anything has changed. The "WF" here means "Well Formed" to distinguish it from a modification that is based on the structure of the DTD. Below this root element, or indeed below any element with a `deltaxml:delta` attribute with value `WFmodify`, each element will also have a `deltaxml:delta` attribute with one of the following values (1) `add` if this element has been added, (2) `delete` if this element has been deleted, (3) `unchanged` if this element is unchanged, and (4) `WFmodify` if the attributes and/or content of this element have been modified. There are some constraints on how these are nested as summarized in [49]. Monsell has applied for a patent on the DeltaXML output format.

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
 <xupdate:append select="/addresses" child="last()">
    <xupdate:element name="address"><town>Bremen</town></xupdate:element>
  </xupdate:append>
</xupdate:modifications>
```

Figure 5.3: An XUpdate output.

XUpdate. The XUpdate format is used by XMLDIFF and has the advantage of being fully specified in a recommendation created by the XML:DB [79] initiative. XUpdate can be shaped to a certain extent by the implementation, but it essentially consists of commands as shown in Fig. 5.3. An important point of this format is that the delta itself is in a hierarchical format, which is helpful if we are to add context information. The delta represents appending an element `<address>` as the last child of `<addresses>`. The recommendation for XUpdate is easy to understand, and makes use of the XPath standard. The fact that there exists a standard for XUpdate enables it to be easily adopted by others. XUpdate's disadvantages are its verbosity and lack of support for context information for the purpose of patching documents other than those from which the original delta was computed. Also there is no support for selecting only part of a text node, which is useful in creating small deltas.

```
<node id="/*[1]"><node id="./*[2]" /><node op="add" name="d" /></node>
```

Figure 5.4: An XUL output.

XUL. The XUL output format is used by the IBM XML TreeDiff program. IBM has spent a reasonable amount of time developing XUL, using XUL to replace FUL as the default output format for XML TreeDiff. An example of XUL output is given in Fig. 5.4. The format is not very readable as nodes are referred to by numbers, not by their names or values. The delta here describes appending an element `d` to an element at XPath `/*[1]/*[2]`. For example, patching the XML document `<a><b/><c/></a>` with this delta results in `<a><b/><c/><d/></a>`. Although this output format is of limited help to a user, from a machine's point of view it could make for faster and easier patching by applying a depth-first traversal of the XUL, when patching one of the same documents on which the delta was produced.

From the algorithms covered earlier, the most appropriate algorithms seem to be XMDIFF and FMES. The XMDIFF algorithm allows differencing of large files and produces minimal edit scripts, both points which are important to creating a useful *diff* utility. The FMES algorithm does not always produce minimal deltas and only works in main memory, but should run substantially faster. In many applications it is preferable to quickly see the changes at a glance rather than wait longer and be given a slightly more minimal delta.

From the output formats described earlier the two most apt formats are XUpdate and XUL. XUpdate gives a more textual account of changes and is to some extent a standard, whilst XUL gives a precise and less verbose account of changes that is more appropriate for programs. Neither of the output formats support extra context information, which is necessary to produce good patches for documents other than those from which the delta was computed. Overall, although algorithms exist which are capable of efficiently calculating changes, there is no product which includes *all* of the following qualities:

- An output format that is good for patching changed files, i.e. the minimum changes should be represented. This implies that the changes should be identified at the lowest possible level in the XML tree structures.
- An output format that is symmetrical, i.e. it should contain a full record of both the position and content of items added and items deleted. Thus the delta file can be used as a basis for both update and undo operations.
- A fast and accurate algorithm, i.e. the delta file should contain all information necessary to identify the changes in both files but the delta file should not contain information that is unchanged between the two files.
- The ability to handle large files.
- An open source license.
- Not strongly tied to a particular XML parser.
- Has an independent and fully specified output format. In addition changes should be represented in XML so that all existing XML tools can be applied to them, for example to display them or to check them automatically.
- Notion of equality besides equality defined on canonical XML.

Especially the last point is not supported by any of the listed tools. None of these tools provide a stronger notion of equality, i.e. none of the existing products considers *semantics* of XML documents but syntactically different elements may appear to be semantically equivalent. In the following section we introduce our semantics-aware change detection utilizing semantic aspects of documents. In particular, we use the underlying semantics of document parts (1) to identify syntactically different fragments to be semantically equal and thus to minimize the number of affected fragments and (2) to frame the syntactical representation of fragments and thus to help to locate changes relative to the internal structure.

5.2 A Semantic Difference Analysis

In order to generate an efficient edit script for two XML documents, we need to detect their parts in which these documents differ from each other. In other words, we must be able to answer whether two XML elements are equal. As to XPath equality, two elements are considered to be weakly equal if their string values are equal. The string value is essentially all of the PCDATA between the element's start and end tags, even if the element has descendant elements. In this sense, for example, the `<w>` and `<z>` elements in the Fig. 5.5 are considered to be equal, because they both have a string value of `abcdefghi`. Their differences, however, are rather obvious: they have different names, `<w>` has its `def` in a child element that `<z>` does not have, and they have different attributes in different places. The "XQuery 1.0 and XPath 2.0. Functions and Operators" specification [85] provides a new equality criterion: deep equality. The term will not be completely new to JAVA developers and other users of object-oriented languages. In XPath, two deep-equal elements have the same XPath tree representing them (see [85] for a more technical definition).

```
<a>
  <w>abc<y color="blue">def</y>ghi</w>
  <z flavor="chocolate">abcdefghi</z>
</a>
```

Figure 5.5: XPath Weak Equality.

There is yet another equality specification with respect to the Document Object Model (DOM) [41]. Two nodes are said to be weakly equal if they have the same type (element, attribute, or text) and the same name. Also, if the corresponding elements contain data, it also must be the same. Finally, if the nodes have attributes then both collections of attribute names must be the same and the attributes corresponding by name must be equal as nodes. Two nodes are deeply equal if they are weakly equal, the child node sequences have the same length, the nodes corresponding by index are deeply equal, and the pairs of equal attributes must in fact be deeply equal.

In certain scenarios, however, it is important to be able to compare two XML trees for *equivalence*. For example, if one writes a web service that serves results of queries, and wants to cache query results so that duplicate queries use previously cached results instead of always accessing the underlying database. The senders of those queries may potentially be using a variety of tools to generate the queries, though, these tools may introduce trivial differences into the XML. The intent of the queries may be identical, but the above listed notions of equality return false even if the XML trees compared contain semantically equivalent, but trivially different queries. It is this class of equality that this finding addresses: given two distinct XML structures, can we decide if they convey "the same information".

Can XML normalization solve this issue? No, but it helps! After normalizing XML trees, there is a greater chance that semantically equivalent XML trees will evaluate as equivalent. The following is a list of issues that should be addressed when normalizing an XML tree:

- Insignificant white space should not exist in a normalized tree.
- Namespace prefixes and the use of default namespaces should not be significant. It is sufficient to compare QNames while disregarding whether the namespaces are serialized by a prefix, or as the default namespace, i.e. it is sufficient to compare extended QNames.
- Missing default elements and attributes should be added to the XML tree when normalizing.

- Entity references should be expanded when normalizing.
- Values of elements and attributes of certain data types can be normalized. Types that can be normalized include `xsd:boolean`, `xsd:dateTime`, `xsd:decimal`, `xsd:double`, `xsd:float`, `xsd:hexBinary`, and `xsd:language`.
- The attributes `xsi:schemaLocation` and `xsd:noNamespaceSchemaLocation` exist only to give hints to a schema processor about the location of the schema. These attributes should be discarded when normalizing an XML tree.
- Attributes should be ordered alphabetically by namespace and name, eliminating insignificant ordering differences.
- Comments and processing instructions are not semantically significant when comparing trees. These should be removed when normalizing.

We term a normalization *adequate*, if all here listed requirements are met. From here on we assume adequate normalizations.

5.2.1 Equivalence Systems on *fsp*-trees

XML differencing tools utilizing adequate normalizations immediately produce less verbose edit scripts but the weak point of these tools is that one can not define application-specific "normalization rules" without changing the implementation. User should be allowed to parametrize XML differencing tools through document type specific normalization rules. That means that regarding particular XML element types user should be able to specify equivalence rules to be considered in the difference analysis, like, for example, that those kind of elements are considered to be normalized modulo specific attributes or some other bespoke equivalence relation.

```
<guests>
 <header>...</header>
 <body>
  <person confirmed="true">
   <firstName>Manfred</firstName>
   <lastName>Stochl</lastName>
   <email>manfred@stochl.de</email>
  </person>
  <person confirmed="true">
   <firstName>Lilia</firstName>
   <lastName>Leisle</lastName>
   <email type="prv">lilia@leisle.de</email>
  </person>
  <person confirmed="false">...</person>
  [...]
 </body>
</guests>
```

```
<guests>
 <header>...</header>
 <body>
  <person confirmed="true">
   <firstName>Lilia</firstName>
   <lastName>Leisle</lastName>
   <email type="prv">lilia@leisle.de</email>
  </person>
  <person confirmed="false">
   <firstName>Manfred</firstName>
   <lastName>Stochl</lastName>
   <birthday>1/23/45</birthday>
   <email type="bus">manfred@stochl.de</email>
  </person>
  <person confirmed="false">...</person>
  [...]
 </body>
</guests>
```

Figure 5.6: Semantically Equivalent Guest Lists.

Example 5.1. To sharpen our intuition on equivalent XML fragments, let us pick up again the XML document depicted in Fig. 3.3. To clarify things we flattened the document, i.e. we substitute the include operation by the actual content. An excerpt of the resulting document is illustrated on the left side of Fig. 5.6.

Our focus in this scenario is on the identification of the invited guests. A guest is represented by an unordered `person` element with the addition of whether this is committed or canceled (`confirmed` attribute). Furthermore, information such as first name (`firstName`), surname (`lastName`), and e-mail address is stored. For the latter the distinction is between private address (`prv`) and business address (`bus`) encoded within a (`type`) attribute whereas the type is defaulted to be private. Optionally the date of birth is represented in a `birthday` element.

Over time, the entries in the guest list change. Thus, for example, on the right side of Fig. 5.6 the order of guests is permuted and for one `person` element the status of commitment and the e-mail address type changed. In addition, the date of birth has been registered.

At an early stage of planning, however, we might not be interested in the status regarding commitment or cancellation and we do not care if we send the invitation to the business address or home address, but we are only interested in the names with the associated e-mail address. Therefore, one can imagine that we define the primary key of a guest as a combination of first name, last name and e-mail address, i.e. comparing two `person` elements changes in confirmed status or e-mail address type are negligible as well as the existence of birthday information. Existing XML differencing tools, however, would even with an adequate normalization consider the two guest records to be different. The mere change of the confirmation status makes the two XML fragments unequal although regarding our primary key definition those two are equal. ⧫

In [105] we drafted a semantic notion of similarity between individual elements of XML documents to give users the maximal freedom in primary key definitions. We claimed that these *equivalence systems* give us a stronger notion of equality and in turn XML *diff* algorithms parametrized with those lead to more compact, less intrusive edit scripts. Here we give a precise definition of equivalence systems generalized on document type specific *fsp*-trees. The concrete differencing algorithm parametrized by such an equivalence system is treated in Sec. 5.2.2.

Definition 5.1 (Document Types & Equivalence Systems). A **document type specification** $\mathscr{D}$ is a tuple $\langle \underline{\mathbb{V}}_{\text{syn}}, \underline{\mathbb{E}}_{\text{syn}}, P \rangle$, where P is predicate on $\ddot{\mathbb{G}}^{\mathbb{E}_{\text{syn}}}_{\underline{\mathbb{V}}_{\text{syn}}}$ which specifies the following set of documents

$$\mathbb{FS}^{+}_{\mathscr{D}} := \{\, \mathscr{T} \in \ddot{\mathbb{G}}^{\mathbb{E}_{\text{syn}}}_{\underline{\mathbb{V}}_{\text{syn}}} \mid \mathscr{T} \textit{ is a fsp-tree and } P(\mathscr{T}) \textit{ holds} \,\}$$

Two document type specifications are **disjoint** if their respective node and edge types are pairwise disjoint.

An **equivalence system** $\mathscr{E}_{\mathscr{D}}$ on *fsp*-trees is a tuple $\langle \equiv, o \rangle$ where $\equiv$ is a congruence on $\mathbb{FS}^{+}_{\mathscr{D}}$ and o a predicate on $\mathbb{FS}^{+}_{\mathscr{D}}$. We denote the set of all equivalence systems with $\mathbb{Q}$ and omit the index of $\mathscr{E}$ unless context requires.

A document type specification characterizes the syntactically correct documents and the equivalence system defines equivalence classes on their subparts. As a typical document type specification one may think of a DTD with the predicate P representing the respective XML validness. Regarding an equivalence system, one may think of a factorization of $\mathbb{FS}^{+}_{\mathscr{D}}$ by $\langle \equiv, o \rangle$, $\mathbb{FS}^{+}_{\mathscr{D}} / _{\langle \equiv, o \rangle}$. With an equivalence system $\mathscr{E}_{\mathscr{D}}$ at hand, we can now resolve two *fsp*-trees being equivalent and decide with respect to $\equiv$ if two XML documents convey “the same information”, in particular: two XML documents are considered as “semantically” equal with respect to an equivalence system $\mathscr{E}_{\mathscr{D}}$ if and only if both respective *fsp*-trees are in the same equivalence class, i.e. two *fsp*-tree encoded XML documents $\mathscr{T}_1, \mathscr{T}_2 \in \mathbb{FS}^{+}_{\mathscr{D}}$ are equal if and only if $\mathscr{T}_1 \equiv_{\mathscr{D}} \mathscr{T}_2$ holds. To handle both cases, ordered and unordered XML fragments, we utilize the o predicate: a *fsp*-tree encoded XML fragment may be permuted *iff* $\neg o(\_)$ holds. Note that o is already subsumed by $\equiv$ but due to the high degree of usages, we have decided to explicitly include this predicate.

```
⟨eqspec⟩  ::= ‘equivspec’ ID ⟨dtd⟩? ⟨ext⟩? ‘{’ ⟨elem⟩* ‘}’
⟨dtd⟩     ::= ‘for’ DOCTYPE
⟨ext⟩     ::= ‘extends’ ID
⟨elem⟩    ::= ‘unordered’? ‘element’ TYPE (‘{’ ⟨alt⟩ ‘}’ | ⟨body⟩)
⟨alt⟩     ::= ‘alternative’ ‘{’ ⟨body⟩+ ‘}’
⟨body⟩    ::= ‘{’ ⟨annos⟩? ⟨consts⟩? ‘}’
⟨annos⟩   ::= ‘annotations’ ‘{’ (⟨topts⟩ | ⟨allbut⟩) ‘}’
⟨consts⟩  ::= ‘constituents’ ‘{’ ⟨oconst⟩? ⟨uconst⟩? ‘}’
⟨oconst⟩  ::= ‘ordered’ ‘{’ (⟨topts⟩ | ⟨allbut⟩) ‘}’
⟨uconst⟩  ::= ‘unordered’ ‘{’ (⟨topts⟩ | ⟨allbut⟩) ‘}’
⟨allbut⟩  ::= (‘_’ | ‘*’) ‘\’ ‘{’ ⟨types⟩ ‘}’
⟨topts⟩   ::= ⟨topt⟩ | ⟨topt⟩ , ⟨topts⟩
⟨topt⟩    ::= TYPE | TYPE ‘?’
⟨types⟩   ::= TYPE | TYPE , ⟨types⟩
```

Figure 5.7: EQ Syntax for an Equivalence System.

Architecturally, an equivalence system splits up the vocabulary of an XML document into similarity groups with respect to specific “normalization rules”. The problem of assigning elements to the appropriate groups is left to the user. However, this choice is generally made by analyzing the respective document type specification. Consequently, it allows to reuse the same equivalence system for all documents referring to that schema.

In Fig. 5.7 we present our declarative EQ syntax to specify equivalence systems and the respective predicates in particular. We approve as equivalence relations only those that are describable within the EQ syntax, i.e. we restrict ourselves to those sub-classes of equivalences. Empirical evaluations have shown that therein we can described the substantial equivalences for change management.

Note 5.1. A concrete specification of these subclasses, and a proof of decidability is part of future work (cf. Chap. 11).

An equivalence specification has an `ID` for identification. An `ID` is any alphanumeric string not beginning with a digit, but possibly including underscores, a number, or any quoted string possibly containing escaped quotes. Optionally the referred document type definition is declared by an ⟨*dtd*⟩ production rule whereas `DOCTYPE` is a relative or absolute URL. An equivalence specification can inherit from another equivalence specification. The `extends` clause defines inherited members of the equivalence specification whereas the body defines overriding or new members. In case of element capturing the behavior is undefined.

The set of members is specified by ⟨*elem*⟩ production rules. These rules frame the core of an equivalence specification. Altogether they define the predicates $\equiv$ and o of the described equivalence system. The concrete semantics is provided below. An element is declared by the `element` keyword. Elements are identified by `TYPE`. A `TYPE` is either a string or one of the following terminal symbols specific to XML: (1) `'<TEXT>'` referring to XML text nodes, (2) `'<COMMENT>'` referring to XML comment nodes, or (3) `'<PROCINST>'` referring to XML processing instruction nodes. If a DOCTYPE has been declared, TYPE has to be valid with respect to DOCTYPE, otherwise it will be ignored. With respect to the XML specification all elements are ordered *per se*. By prefixing an element specification with the keyword unordered one can break through this property.

The body of an element specification comprises either alternative constraints ($\langle alt \rangle$) or a singleton constraint ($\langle body \rangle$). An alternative is a sequence of disjoint constraints. With both production rules one determines the restrictions of two *fsp*-trees being equivalent. That is, two *fsp*-trees $\mathscr{T}_1$ and $\mathscr{T}_2$ with $type(\Uparrow\mathscr{T}_1) = type(\Uparrow\mathscr{T}_2) = \texttt{TYPE}$ are considered to be equivalent *iff* the specified constraint declared within a $\langle body \rangle$ production rule holds or, in case of an alternative declaration, if at least one constraint within the sequence holds.

A constraint ($\langle body \rangle$) consists of optional annotations ($\langle annos \rangle$) or constituents ($\langle consts \rangle$) specifications. Note that we used these terms rather than "attributes" and "children" to be more general with respect to further semi-structured document formats (e.g. LaTeX). The set of annotations specifies the element's annotations to be consider during a comparison. Annotations marked with a question mark (?) are optional, the others are required. This means that for semantic equivalence required annotations have to be present and equal but optional ones only have to be equal if present.

A constituents specification ($\langle consts \rangle$) declares the set of ordered ($\langle oconst \rangle$) and unordered ($\langle uconst \rangle$) children to be considered in an equivalence check. Again, one can specify optional constituents with the same semantics as for annotations. For the ordered case the specified list of constituents has to be pairwise equal. For the unordered case the set of constituents has to be equal. For both cases equality is with respect to the individual element specification.

To simplify matters we introduce a "syntactic sugar" production rule ($\langle allbut \rangle$) which allows for specifying annotations as well as constituents by the exclusion principle: all ($\_ \mid *$) members have to be considers but ($\backslash$) the ones specified in the set declared by the respective $\langle types \rangle$ production rule. The terminal symbol "$\_$" is used for early-binding (compile time) and "$*$" is used for late-binding (run time). Both terminal symbols are resolved to *all* members of a equivalence specification hierarchy including (if present) all members of DOCTYPE not explicitly declared. The late-binding may be used to dynamically add members during run time.

To strictly define the semantics of the EQ syntax, we algorithmically describe the predicates defined by a set of $\langle elem \rangle$ production rules (cf. Fig. 5.9). The respective algebraic and abstract data type declarations are preceded (cf. Fig. 5.8).

Note 5.2. All algorithms presented here are written concretely in the programming language SCALA [45]. Only for readability we marginally polished SCALA's notion of anonymous functions, utilize subscripts and left out implicit conversions as well as type parametrization, which play no further role for understanding. The complete code is available at [103].

```
type FS+ = scala.xml.Node
case class EquivSpec(val name: String, val alts : Map[Σ, Alts], val uord: Set[Σ])
case class Alts (constraints: List [Cons])
case class Cons(ra: Set[ℙ], oa: Set[ℙ], roc: Set[Σ], ooc: Set[Σ], ruc: Set[Σ], ouc: Set[Σ])
```

Figure 5.8: The Types of an Equivalence System.

We start our description by introducing the utilized data types. As our major concern here are *fsp*-tree encoded semi-structured documents, the abstract data type $\mathbb{FS}^+$ representing our fundamental data structure is defaulted to SCALA's algebraic type Node(label: String, attributes: MetaData, child: Seq[Node]) for XML representations. We employ $\mathbb{FS}^+$

rather than $\mathbb{FS}^+_{\mathscr{D}}$ as document type checking for a *fsp*-tree $\mathscr{T}$ with respect to a document type specification $\mathscr{D}$, i.e. verifying whether $\mathscr{T} \in \mathbb{FS}^+_{\mathscr{D}}$, is externalized to validating XML parsers, like SCALA's one. Furthermore, namespaces are neglected at this point but could be handled by SCALA's algebraic type `Elem` extending `Node`. The `MetaData` type is SCALA's representation of XML attributes. The mapping of *fsp*-tree functions onto $\mathbb{FS}^+$ is straightforward. For example, the node type[3] is retrieved by the `label` slot ($type(\Uparrow\mathscr{L}) = \mathscr{L}$.label), and the set of trees induced by the children of a node is represented by the `child` slot ($\mathscr{L}||_{\Uparrow\mathscr{L}} = \mathscr{L}$.child).

The algebraic type `EquivSpec` represents a parsed equivalence specification. An `EquivSpec` consists of a symbolic name (`name`), a partial function `alts` mapping element names to a sequence of alternative constraints (`Alts`), and a set `uord` specifying the unordered elements.

An alternative consists of a sequence of constraints (`Cons`) comprising required annotations (`ra`), optional annotations (`oa`), required ordered constituents (`roc`), optional ordered constituents (`ooc`), required unordered constituents (`ruc`), and optional unordered constituents (`ouc`).

```
def $o(\mathscr{L}:\mathbb{FS}^+) = !(type(\Uparrow\mathscr{L})) \in$ eqspec.uord)

def $\equiv(\mathscr{L}:\mathbb{FS}^+, \mathscr{R}:\mathbb{FS}^+)$ = if($type(\Uparrow\mathscr{L}) == type(\Uparrow\mathscr{R})$) eqspec.alts($type(\Uparrow\mathscr{L})$) match {
  case Some(Alts(Nil)) $\Rightarrow$ true                                   //equal by name
  case Some(Alts(cs))  $\Rightarrow$ cs exists ($\lambda c$ :Cons. $\mathscr{L} \sim \mathscr{R}$ wrt c) //equal by spec
  case None            $\Rightarrow \mathscr{L} == \mathscr{R}$                          //DOM deep equal
} else false

def $\sim(\mathscr{L}:\mathbb{FS}^+, \mathscr{R}:\mathbb{FS}^+$, c: Cons) = c match {
  case Cons(ra, oa, roc, ooc, ruc, ouc) $\Rightarrow \beta_{eq}(\mathscr{L}, \mathscr{R}$, ra, oa) && $||_{eq}(\mathscr{L}, \mathscr{R}$, roc, ooc, ruc, ouc)
}

def $\beta_{eq}(\mathscr{L}:\mathbb{FS}^+, \mathscr{R}:\mathbb{FS}^+$, ra: Set[$\mathbb{P}$], oa: Set[$\mathbb{P}$]) =
  (ra forall ($\lambda a:\mathbb{P}$. (($\beta(\Uparrow\mathscr{L}, a), \beta(\Uparrow\mathscr{R}, a)$) match {case (la, ra) $\Rightarrow$ la == ra case _ $\Rightarrow$ false}))) &&
  (oa forall ($\lambda a:\mathbb{P}$. (($\beta(\Uparrow\mathscr{L}, a), \beta(\Uparrow\mathscr{R}, a)$) match {case (la, ra) $\Rightarrow$ la == ra case _ $\Rightarrow$ true}))

def $||_{eq}(\mathscr{L}:\mathbb{FS}^+, \mathscr{R}:\mathbb{FS}^+$, roc: Set[$\Sigma$], ruc: Set[$\Sigma$], ooc: Set[$\Sigma$], ouc: Set[$\Sigma$]) =
  ((roc forall ($\lambda x:\Sigma$. ($\mathscr{L}||_{\Uparrow\mathscr{L}}$ exists ($type$(_) == $x$))) && ($\mathscr{R}||_{\Uparrow\mathscr{R}}$ exists ($type$(_) == $x$))) &&
  ($\equiv_{\text{seq}}(\mathscr{L}||_{\Uparrow\mathscr{L}}$ filter ($type$(_) $\in$ roc), $\mathscr{R}||_{\Uparrow\mathscr{R}}$ filter ($type$(_) $\in$ roc))))
  &&
  ((ruc forall ($\lambda x:\Sigma$. ($\mathscr{L}||_{\Uparrow\mathscr{L}}$ exists ($type$(_) == $x$))) && ($\mathscr{R}||_{\Uparrow\mathscr{R}}$ exists ($type$(_) == $x$))) &&
  ($\equiv_{\text{set}}(\mathscr{L}||_{\Uparrow\mathscr{L}}$ filter ($type$(_) $\in$ ruc), $\mathscr{R}||_{\Uparrow\mathscr{R}}$ filter ($type$(_) $\in$ ruc))))
  &&
  (($\lambda x:Set[\mathbb{P}]$. $\equiv_{\text{seq}}(\mathscr{L}||_{\Uparrow\mathscr{L}}$ filter ($type$(_) $\in$ x), $\mathscr{R}||_{\Uparrow\mathscr{R}}$ filter ($type$(_) $\in$ x)))
       (ooc filter ($\lambda x:\Sigma$. ($\mathscr{L}||_{\Uparrow\mathscr{L}}$ exists ($type$(_) == $x$) && $\mathscr{R}||_{\Uparrow\mathscr{R}}$ exists ($type$(_) == $x$)))))
  &&
  (($\lambda x:Set[\mathbb{P}]$. $\equiv_{\text{set}}(\mathscr{L}||_{\Uparrow\mathscr{L}}$ filter ($type$(_) $\in$ x), $\mathscr{R}||_{\Uparrow\mathscr{R}}$ filter ($type$(_) $\in$ x)))
       (ouc filter ($\lambda x:\Sigma$. ($\mathscr{L}||_{\Uparrow\mathscr{L}}$ exists ($type$(_) == $x$) && $\mathscr{R}||_{\Uparrow\mathscr{R}}$ exists ($type$(_) == $x$)))))
```

Figure 5.9: The Predicates an Equivalence System.

Now we come to the semantics of the predicates o and the equivalence relation $\equiv$ (cf. Fig. 5.9). A respective equivalence specification is parsed into an `eqspec` value, i.e. **`val`** `eqspec:`

[3]At this point no distinction as to XML reference statements is required.

`EquivSpec = EQParser.parse(spec).`

The predicate o is represented by the corresponding function $o\colon \mathbb{FS}^+ \to \mathbb{B}$. The interpretation is simple: all *fsp*-trees not contained in the set `uord` are order dependent.

The equivalence relation $\equiv$ is represented by the corresponding function $\equiv\colon \mathbb{FS}^+ \times \mathbb{FS}^+ \to \mathbb{B}$. Precondition for two *fsp*-trees being equivalent is the root node types are equal. Otherwise check for equivalence fails immediately. Element specific alternatives are retrieved by pattern matching on the equivalence specification slot (`alts`). In case no constraints are given (`Alts(Nil)`), the two *fsp*-trees are considered to be equal by name. Potential annotations as well as constituents are neglected. In case no specification exists at all (`None`), the equivalence check defaults to DOM deep equality. In case alternatives are specified, equivalence checks are performed with respect to the actual constraint. That is, for each constraint within an alternative sequence the annotations of the left *fsp*-tree are compared to the annotations of the right *fsp*-tree with respect to the specified required annotations (`ra`) and optional annotation (`oa`). In case of β_{eq} holds, the constituents are compared ($\|_{eq}$). First the required ones, ordered (`roc`) and unordered (`ruc`), and then the optional ones (`ooc` and `ouc`, respectively) are differentiated. The here not listed functions $\equiv_{\text{seq}}$ and $\equiv_{\text{set}}$ recursively compare the respective constituents by pairs. The former is order preserving and the latter order independent. As soon as a constraint is fulfilled, the two *fsp*-trees $\mathscr{L}$ and $\mathscr{R}$ are considered to be equivalent, short $(\mathscr{L},\mathscr{R}) \in \equiv$.

```
<person confirmed="true">
 <firstName>Manfred</firstName>
 <lastName>Stochl</lastName>
 <email>manfred@stochl.de</email>
</person>
<person confirmed="true">
 <firstName>Lilia</firstName>
 <lastName>Leisle</lastName>
 <email type="prv">lilia@leisle.de</email>
</person>
```

```
<person confirmed="true">
 <firstName>Lilia</firstName>
 <lastName>Leisle</lastName>
 <email type="prv">lilia@leisle.de</email>
</person>
<person confirmed="false">
 <firstName>Manfred</firstName>
 <lastName>Stochl</lastName>
 <birthday>1/23/45</birthday>
 <email type="bus">manfred@stochl.de</email>
</person>
```

Figure 5.10: Semantically Equivalent list of Persons.

Example 5.2. Let us take up again our scenario from Ex. 5.1. Still our focus is on the identification of the invited guests. For simplification this time we only concentrate on the two modified `person` elements as depicted in Fig. 5.10. As in the example before, we are not interested in the status regarding commitment or cancellation and we do not care if we send the invitation to the business address or home address, but we are only interested in the names with the associated e-mail address. Therefore, a `person` element is identified by its first name, last name and e-mail address. With an equivalence system, we can now exactly express this situation and consider the two data sets as equivalent.

Figure 5.11 represents in the EQ syntax the respective equivalence system for our scenario where $\mathscr{D}$ refers to some respective DTD. The equivalence specification states that all elements are ordered but the `person` element and the `email` element. Two `person` elements are considered equivalent *iff* their set of constituents comprising `firstName`, `lastName`, and `email` are equivalent with respect to their individual constraints. Note that the order of the constituents to take into account in the equivalence check of `person` elements does not matter. However, in the in-

```
equivspec E for 𝒟 {
  unordered element person     { constituents { unordered { firstName, lastName, email } } }
            element firstName  { constituents { unordered { <TEXT> } } }
            element lastName   { constituents { unordered { <TEXT> } } }
  unordered element email      { constituents { unordered { <TEXT> } } }
            element birthday   { constituents { unordered { <TEXT> } } }
```

Figure 5.11: An Equivalence System for the List of Persons in Fig. 5.10.

dividual equivalence checks for `firstName` elements and `lastName` elements, respectively, the element position matters.

For clarification let us assume the following adaptation to our scenario: in the second `person` element on the right side of Fig. 5.10 we interchange the elements `firstName` and `lastName`. Regarding identification of guests this modification does not matter. We can still identify the first `person` element on the left side of Fig.5.10 being equivalent to the second one on the right side of Fig. 5.10. One may think of a compound primary key for `person` elements whereas in this case a key is interpreted as a set. However, differentiating the sequence of children of both `person` elements the order of `firstName` and `lastName` (and `birthday`) matters as to `E`. Exactly this flexibility is covered by an equivalence system.

Back to our original scenario. Besides the fixed position of a `firstName` element and a `lastName` element, both elements are considered to be equivalent *iff* their nested text fragments are equal. An `email` element also depends on its nested text fragments for equivalence but is order independent. The last element in our artificial XML instance, the `birthday` element, is order dependent and, again, depends on nested text fragments for equivalence. Hence, by this equivalence specification we have achieved to identify both lists of persons to be considered equivalent, i.e. we expressed the fact that both sequences convey the same information module the equivalence system `E`. ♦

Note 5.3. By utilizing our syntactic sugar syntax the constraint for a `person` element could also be represented by the following term:

```
unordered element person { constituents { unordered { _ \ { email, birthday } } } }
```

How exactly an equivalence system is used in our difference analysis is discussed in the next section.

5.2.2 *sdiff*: A Semantic Differ

So far we have algorithmically described the two predicates $\equiv$ and o of an equivalent system. Before we get to the actual algorithm of the difference analysis, we need an identification mechanism to analyze sequences and sets of *fsp*-trees. Remember the fact that the order of `person` elements in our artificial format for a guest list does not matter, so comparing guest records as two sequences of `person` elements, we need a mechanism to identify equivalent elements between both. Note, that by utilizing the o predicate we can regard sequences as sets.

For further illustration let us denote the two sequences of `person` elements in Fig. 5.10 with $\langle \mathscr{M}, \mathscr{L} \rangle$ and $\langle \mathscr{L}', \mathscr{M}' \rangle$, respectively. We need to be able to identify corresponding *fsp*-trees with respect to an equivalence system, i.e. we need to be able to identify $\mathscr{M} \equiv \mathscr{M}'$ and $\mathscr{L} \equiv \mathscr{L}'$. We will

```
case class Slit (val before: Option[List[$\mathbb{FS}^+$]],val focus: Option[$\mathbb{FS}^+$],val after: List[$\mathbb{FS}^+$])

def $\equiv^*$($\mathscr{T}$: $\mathbb{FS}^+$, TS: List[$\mathbb{FS}^+$]) = $\equiv^*$($\mathscr{T}$, TS, Nil)

def $\equiv^*$($\mathscr{T}$: $\mathbb{FS}^+$, TS: List[$\mathbb{FS}^+$], acc: List[$\mathbb{FS}^+$]) = TS match {
  case Nil                    ⇒ Slit(None,None,acc)
  case _ if ($\mathscr{T} \equiv$ TS(0)) ⇒ if($o(\mathscr{T})$) acc match {
    case Nil ⇒ Slit(None, Some(TS(0)), TS.drop(1))
    case _   ⇒ Slit(Some(acc), Some(TS(0)), TS.drop(1))
  } else Slit (None, Some(TS(0)), acc ::: TS.drop(1))
  case _                      ⇒ $\equiv^*$($\mathscr{T}$, TS.drop(1), acc ::: List(TS(0)))
}
```

Figure 5.12: The Identification Mechanism of an Equivalence System.

see shortly that an differencing algorithm equipped with such an identification of syntactically different but semantically equal sequences clearly generates less intrusive edit scripts. In turn, the resulting compactness of edit scripts reduces storage space and improves query efficiency in document management systems, while minimal intrusiveness is important for humans to track and understand changes.

In the following we algorithmically describe our mechanism within an equivalence system for identification of equal *fsp*-trees with respect to $\equiv$ and o. Figure 5.12 illustrates the respective excerpt of an equivalence system. Note, in combination with the algebraic data structures and algorithms depicted in Fig. 5.9 this frames a complete `EquivalenceSystem`, our key data structure for difference analysis.

The entry point for identification of an equivalent *fsp*-tree within a sequence of *fsp*-trees is represented by the function $\equiv^*: \mathbb{FS}^+ \times List[\mathbb{FS}^+] \rightarrow$ `Slit`. For a *fsp*-tree $\mathscr{T}$ the algorithm scans the sequence `TS` for an equivalent *fsp*-tree with respect to the predicates $\equiv$ and o of the actual equivalence system. In case of an equivalent *fsp*-tree has been identified ($\mathscr{T} \equiv$ TS(0)) within the sequence `TS` a `Slit` depending on the ordering of $\mathscr{T}$ is returned.

The algebraic data type `Slit` describes the "slit" within the corresponding *fsp*-tree, i.e. the *fsp*-trees `before` and `after` the corresponding one (`focus`). In case of the current *fsp*-tree out of `TS` fails the equivalence check the rest of the sequence is verified. If no equivalent *fsp*-tree exists, an "empty" slit is returned. Here "empty" denotes the fact that there is no equivalent *fsp*-tree (`focus == None`) and consequently the sequence of preceding *fsp*-trees is empty as well (`before == None`). However, in order to process the siblings of $\mathscr{T}$ the accumulator for the subsequent *fsp*-trees is equal to the input sequence (`after == TS`).

For illustration of $\equiv^*$ and the respective `Slit`s, in particular, let us go back to our guest list example and the thereon defined equivalence system (cf. Fig. 5.11). An equivalence check between the sequence $\mathbb{S}_1 = \langle \mathscr{M}, \mathscr{L} \rangle$ of `person` elements depicted on the left side of Fig. 5.10 and the sequence $\mathbb{S}_2 = \langle \mathscr{L}', \mathscr{M}' \rangle$ of `person` elements depicted on the right side of Fig. 5.10 is accomplished by for each element $e \in \mathbb{S}_1$ scanning $\mathbb{S}_2$ for an equivalent one. In case of $e = \mathscr{M}$ the identification mechanism $\equiv^*(e, \mathbb{S}_2)$ extends the equivalence relation $\equiv$ by the pair $\langle \mathscr{M}, \mathscr{M}' \rangle$ represented by the following slit:

```
Slit (None,
  Some(<person confirmed="false"><firstName>Manfred</firstName>...</person>),
  List(<person confirmed="true"><firstName>Lilia</firstName>...</person>))
```

This slit identifies $e \equiv$ `focus` whereas `focus` $= \mathscr{M}'$. Due to order independence there are no elements within $\mathbb{S}_2$ to be considered before but one pending.

For comparison, let us consider again our adapted scenario where the elements `firstName` and `lastName` in $\mathscr{M}'$ are swapped and this time additionally being order dependent. An equivalence check between $e = \mathscr{M}|_{\text{firstName}}$ and $\langle \mathscr{M}'|_{\text{lastName}}, \mathscr{M}'|_{\text{firstName}}, \mathscr{M}'|_{\text{birthday}}, \mathscr{M}'|_{\text{email}} \rangle$ then results in the following slit:

```
Slit (Some(List(<lastName>Stochl</lastName>)),
  Some(<firstName>Manfred</firstName>),
  List(<birthday>1/23/45</birthday>, <email type="bus">manfred@stochl.de</email>))
```

The `focus` slot is assigned to the `firstName` element expressing the fact that $e \equiv$ `focus`. Furthermore, due to the fact that `firstName` element and `lastName` element are swapped in $\mathscr{M}'$ but regarding the equivalence specification are order depended, the `Slit` marks the preceding and subsequent elements within $\mathscr{M}'$. The `before` slot points to `<lastName>Stochl</lastName>` and `after` holds the list of the subsequent `focus` elements.

With the formal and algorithmic introduction of an equivalence system including the predicates $\equiv$ and o and the identification mechanism $\equiv^*$ we now have a sophisticated apparatus for describing and identifying sequences of *fsp*-trees expressing semantically equal content. In addition, the `Slit` data structure is an adequate construct for marking up equivalent elements within semi-structured documents taking order dependence into account. The usage and interpretation of both an equivalent system and `Slit`s within our difference analysis is depicted in Fig. 5.14. The corresponding algebraic data types are depicted in Fig. 5.13.

The main entry point for a semantic difference analysis on *fsp*-trees is represented by the function *sdiff*: $\mathbb{Q} \times List[\mathbb{FS}^+] \times List[\mathbb{FS}^+] \rightarrow \Delta$. The algorithm pairwise compares the two sequences of *fsp*-trees, $\mathscr{L}$ and $\mathscr{R}$, with respect to the equivalence system E and computes an edit script Δ such that $spatch(sdiff(E, \mathscr{L}, \mathscr{R}), \mathscr{L}) == \mathscr{R}$ holds modulo permutations of unordered elements[4]. The output format follows the example of XUpdate: a command comprises a *fs*-path slot (π) and a sequence of arguments (`args`). The *fs*-path specifies the node the sequence of arguments are applied to.

```
abstract class Command(val π: Path, val args: Seq[Arg])
case class append(val π: Path, val args: Seq[Arg]) extends Command
case class remove(val π: Path, val args: Seq[Arg]) extends Command
case class update(val π: Path, val args: Seq[Arg]) extends Command
case class before(val π: Path, val args: Seq[Arg]) extends Command
case class patch(val π: Path, val args: Seq[Arg])  extends Command

abstract class Arg
case class node(e: Seq[𝔽𝕊⁺]) extends Arg
case class anno(e: (ℙ,𝔻))     extends Arg
case class delta(e: String)   extends Arg
```

Figure 5.13: The *sdiff* Output Commands.

Supported commands are: (1) `append` appends `args` as right most child of node at π, (2) `remove` removes note at π, (3) `update` substitutes node at π by `args`, (4) `before` puts `args`

[4]For *spatch* we currently utilize [15].

as direct left sibling of node at π, and (5) `patch` substitutes node at π by `args`. Note that as to version control, edit scripts have to be invertible to re-construct previous versions. Therefore, in contrast to the corresponding XUpdate command, the `args` slot of a `remove` command represents the deleted node. The `patch` command is special as well in sense of there is no counterpart in XUpdate. The *sdiff* algorithm utilizing the LCS algorithm to support fine-grained change detection on XML text nodes represents the respective edit script within a `patch` command.

Supported command arguments are: (1) `node` creates the *fsp*-trees defined in the sequence `e`, (2) `anno` creates an annotation pair `(key, value)`, and (3) `delta` creates an XML text node representing in the standard UNIX `diff` format the differences between the two compared versions.

```
def sdiff(E: $\mathbb{Q}$, $\mathscr{L}$: List[$\mathbb{FS}^+$], $\mathscr{R}$: List[$\mathbb{FS}^+$]) = $\delta_{\mathbb{FS}^+}$(E, $\mathscr{L}$, $\mathscr{R}$, $\varepsilon$, Nil).reverse

def $\delta_{\mathbb{FS}^+}$(E: $\mathbb{Q}$, $\mathscr{L}$: List[$\mathbb{FS}^+$], $\mathscr{R}$: List[$\mathbb{FS}^+$], $\pi$: Path, cs: List[Command]) = ($\mathscr{L}$, $\mathscr{R}$) match {
  case (Nil, Nil)                  $\Rightarrow$ cs
  case (Nil, $\mathscr{R}_1$::$\mathscr{R}_n$)          $\Rightarrow$ $\delta_{\mathbb{FS}^+}$(E, Nil, $\mathscr{R}_n$, $\pi$, append($\pi\uparrow$, node($\mathscr{R}_1$)) :: cs)
  case ($\mathscr{L}_1$::$\mathscr{L}_n$, Nil)          $\Rightarrow$ $\delta_{\mathbb{FS}^+}$(E, $\mathscr{L}_n$, Nil, $\pi+$, remove($\pi$, node($\mathscr{L}_1$)) :: cs)
  case (($\mathscr{L}_1$ : $\mathbb{t}$)::$\mathscr{L}_n$, ($\mathscr{R}_1$ : $\mathbb{t}$)::$\mathscr{R}_n$) $\Rightarrow$
    if ($\gamma(\Uparrow\mathscr{L}_1)$ != $\gamma(\Uparrow\mathscr{R}_1)$)
      $\delta_{\mathbb{FS}^+}$(E, $\mathscr{L}_n$, $\mathscr{R}_n$, $\pi+$, {if(gnuDiff) patch($\pi$,delta($\delta_{\mathbb{TXT}}(\gamma(\Uparrow\mathscr{L}_1),\gamma(\Uparrow\mathscr{R}_1))$)) else update($\pi$, node($\mathscr{R}_1$))} :: cs)
    else cs
  case ($\mathscr{L}_1$::$\mathscr{L}_n$, _)              $\Rightarrow$ ($\lambda s$: Slit. $\delta_{\mathbb{FS}^+}$(E, $\mathscr{L}_n$, s.after, $\pi+$, s.focus match {
    case Some($\mathscr{F}$) $\Rightarrow$ $\delta_{\mathbb{FS}^+}$($\mathscr{L}_1||_{\Uparrow\mathscr{L}_1}$, $\mathscr{F}||_{\Uparrow\mathscr{F}}$, $\pi\downarrow$, $\delta_\beta$($\mathscr{L}_1$, $\mathscr{F}$, $\pi$) ::: insert(s.before) ::: cs)
    case _        $\Rightarrow$ insert(s.before) ::: List (remove($\pi$,node($\mathscr{L}_1$))) ::: cs
  })) ($\mathscr{L}_1 \equiv^*_{\mathrm{E}} \mathscr{R}$)
}

def $\delta_\beta$($\mathscr{L}$: $\mathbb{FS}^+$, $\mathscr{R}$: $\mathbb{FS}^+$, $\pi$: Path) =
  (($\beta(\Uparrow\mathscr{R})$ map ($\lambda$(rk,rv): ($\mathbb{P}$,$\mathbb{D}$). ($\beta(\Uparrow\mathscr{L})$ find ($\lambda$(lk,lv): ($\mathbb{P}$,$\mathbb{D}$). rk == lk) match {
    case Some((lk,lv)) if rv != lv $\Rightarrow$ Some(update($\pi$, anno((lk, rv))))
    case None                $\Rightarrow$ Some(append($\pi$, anno((rk, rv))))
    case _                   $\Rightarrow$ None
  }))) filter (None !=) map (_.get)) ::: (
   $\beta(\Uparrow\mathscr{L})$ filter ($\lambda$(lk,lv): ($\mathbb{P}$,$\mathbb{D}$). $\beta(\Uparrow\mathscr{R}$,lk) = $\perp$) map ($\lambda$(lk,lv): ($\mathbb{P}$,$\mathbb{D}$). remove($\pi$,anno((lk,lv)))))
```

Figure 5.14: The *sdiff* Algorithm.

The computation of an edit script is accomplished by pattern matching on *fsp*-tree node types with special handling for XML nodes of type text, comment, and processing instruction. The handling of text nodes within the tree traversal function $\delta^+_{\mathbb{FS}}$ is depicted in Fig. 5.14 whereas handling of comments and processing instructions has been skipped for readability. In case two XML text nodes are different in terms of DOM deep equality, a global switch (`gnuDiff`) directs *sdiff* to utilize either the UNIX `diff` ($\delta_{\mathbb{TXT}}$) and represent the resulting edit script within a `patch` command or to simply update the entire text node. Difference analysis on *fsp*-tree properties is performed by the function δ_β.

Another salient case is the last one in $\delta^+_{\mathbb{FS}}$. Depending on the identification mechanism of the parametrizes equivalence system ($\equiv^*_{\mathrm{E}}$) the resulting `Slit s` is evaluated. In case an equivalent *fsp*-tree has been identified the respective attributes are differentiated by δ_β and respective adjustments are added to the edit script. In case of failure of the equivalence check a substitution in sense of an `insert` and a `remove` command is added to the edit script to replace the left-hand sided *fsp*-tree by the corresponding right-hand sided identified by the `s.before` slot.

Following the example of XPath we have defined the following postfix operators on *fs*-paths: (1) $\pi+$ refers to the right sibling of the node at π, (2) $\pi-$ refers to the left sibling of the node at π, (3) $\pi\downarrow$ refers to the left most child of the node at π, and (4) $\pi\uparrow$ refers to the parent of the node at π.

The function literals `insert` and β, respectively, may be understood as the following simple typed Lambda expressions:

```
val insert = λ x: Option[List[𝔽𝕊⁺]]. x match {
  case Some(es) ⇒ before(path, (es map elem)) :: Nil
  case _        ⇒ Nil
}
val β = λ v: 𝕍(𝒯). P map (β(v, _)) filter (⊥ !=)
```

In order to demonstrate the *sdiff* algorithm in action we apply it twice on the two guest records depicted in Fig. 5.10. Both use cases are illustrated in Fig. 5.15. In the first case we do not declare an

```
val left  = XMLParser.parse(cf. Fig 5.10 left side) withNormalization
val right = XMLParser.parse(cf. Fig 5.10 right side) withNormalization
val spec  = EQParser.parse(cf. Fig 5.11)

val concise =
  update(Root childAt 1 attribute "confirmed", node(Text("false"))) ::
  update(Root childAt 1 childAt 3 attribute "type", node(Text("bus"))) ::
  append(Root childAt 1, node(<birthday>1/23/45</birthday>)) :: Nil

val verbose =
  remove(Root childAt 1,node(<person confirmed="true"><firstName>Manfred</firstName>...)) ::
  append(Root, node(<person confirmed="false"><firstName>Manfred</firstName>...)) :: Nil

sdiff(left,right) must_== verbose                                  // w/o equivalence system
sdiff(new EquivalenceSystem(spec),left,right) must_== concise // w/ equivalence system
```

Figure 5.15: A Semantic Differencing Analysis on the Guest Records in Fig. 5.10.

equivalence system resulting in a difference analysis utilizing an adequate normalization only. In the second case we parametrize the *sdiff* algorithm with the equivalence specification illustrated in Fig. 5.11. We denote edit scripts computed on the basis of equivalence systems *semantically minimized*, short *concise*, and otherwise *verbose*. Therefore the computed edit script of the first case is represented in the `verbose` field and the edit script of the second case is represented in the `concise` field. Obviously, the concise edit script describes the differences on a more fine-granular level than the verbose one.

In summary, the *sdiff* algorithm is a difference analysis function similar to the classic UNIX `diff` or previously mentioned structure-aware difference analysis algorithms. The difference to standard tree difference analysis algorithms is, that it relies on an equivalence system for subtrees to identify corresponding subtrees that need not be syntactically equal. The advantage is that one can indicate on which basis syntactically different subtrees are identified, such as indicating primary and secondary key attributes or sub-elements as illustrated in Ex. 5.2. This allows for a more fine-tuned control compared to the heuristic approaches of standard tree difference analysis algorithms that rely on some built-in syntactic metric to measure the similarity of subtrees.

Remark 5.1. Scientists considering unification in their research try to handle equality for years — and they are not done yet. As our difference analysis highly depends on the set of equivalences this is going to be a big task (complexity/termination analysis and soundness) we will investigate further but declare to future work at this point (cf. Chap. 11). However, a first complexity analysis approach can be found in [143].

5.2.3 Semantics-Based Version Control

In the final part of our semantic differencing analysis, we are dealing with the integration of *sdiff* into conventional versioning workflows. The main problem here is the output format of *sdiff*. Common version control systems expect edit scripts to be either in the *context* format or the *unified* format [82]. We call such output formats *ordinary*. Our semantic differencing algorithm *sdiff*, however, utilizes a generic output format following the example of XUpdate. We term such output formats *treeish*. Hence, *sdiff* is not applicable for common version control systems like SUBVERSION.

To yet gain the benefits of our semantic difference analysis we now introduce a proposal for an integration framework Michael Kohlhase and the author are currently working on. The $\delta\mu\widetilde{\mu}$-*framework* provides a transformation from treeish edit scripts to ordinary but concise ones and a projection to semantically minimize ordinary edit scripts. The framework comprises two operators: (1) the μ operator performing a *sdiff* difference analysis transforms the computed treeish and concise edit script to the corresponding ordinary and concise edit scripts, and (2) the $\widetilde{\mu}$ operator semantically minimizes an ordinary edit script with respect to the semi-structured document the ordinary delta was computed for while complying to the respective output format.

Definition 5.2 (Adequate Edit Scripts). Let $\mathcal{T}_0$ and $\mathcal{T}_1$ be two *fsp*-trees, Δ an ordinary edit script turning $\mathcal{T}_0$ into $\mathcal{T}_1$ and E a respective equivalence system. We call a tuple $\langle\mu,\widetilde{\mu}\rangle$ a $\boldsymbol{\delta\mu\widetilde{\mu}}$**-framework** where μ is a function with

$$\mu\colon \mathbb{Q}\times\mathbb{FS}^+\times\mathbb{FS}^+\to\wp(\Delta)\colon E\times\mathcal{T}_0\times\mathcal{T}_1\mapsto \texttt{diff}(\mathcal{T}_0, \mathit{spatch}(\mathit{sdiff}(E,\mathcal{T}_0,\mathcal{T}_1),\mathcal{T}_0))$$

and $\widetilde{\mu}$ is a function with

$$\widetilde{\mu}\colon \wp(\Delta)\times\mathbb{FS}^+\times\mathbb{Q}\to\wp(\Delta)\colon \delta\times\mathcal{T}_0\mapsto\mu(E,\mathcal{T}_0,\texttt{patch}(\delta,\mathcal{T}_0))$$

We call μ a δ-transformation, $\widetilde{\mu}$ a δ-projection, and term $\delta\mu\widetilde{\mu}$-computed edit scripts **adequate**.

Note the fact that even with no explicit equivalence system, i.e. utilizing adequate normalizations only, our proposed $\delta\mu\widetilde{\mu}$-framework already “semantically” minimizes ordinary edit scripts.

We now demonstrate a $\delta\mu\widetilde{\mu}$-framework application by the example of SUBVERSION `update` and `commit` commands. Following the example of SUBVERSION we denote versioned items located within a working copy by a subscript W, the corresponding base version located within the respective administrative directory by a subscript B, and the corresponding remote item located in the repository by a subscript S.

A SUBVERSION `update` command on a document D_W requests from the repository the respective ordinary edit script δ_S. Semantically minimizing $\widetilde{\delta}_S=\widetilde{\mu}(\delta_S,D_B)$ and transforming local modifications performed on D_W into an adequate edit script $\delta_W=\mu(D_B,D_W)$ gives us all the information necessary for three-way merging changes into $D'_B=\texttt{merge}(\widetilde{\delta}_S,D_B,\delta_W)$. The merged base version then becomes the current working copy version, $D'_W=D'_B$.

In case of a `commit` command the integration is even simpler by just applying μ on the base and current version of D, i.e. $\mu(D_B, D_W)$ and sending the resulting adequate edit script to the repository.

We are confident that via our $\delta\mu\widetilde{\mu}$-framework we have enabled a straightforward integration of our semantic differencing algorithm *sdiff* into conventional versioning workflows giving users fine-grained control on tracking relevant changes which in turn gives us the base for a sophisticated change impact analysis.

5.3 Conclusion

The key ingredients of the presented *sdiff* algorithm are that changes are determined using a generic semantic tree difference analysis parametrized over document type specific equivalence specifications. This grouping condenses the required change related knowledge and in turn facilitates to accomplish a more precise impact analysis.

The algorithm has been implemented in [103, 102] exactly following the principles of a equivalence system. The primary application scenario for *sdiff* is to identify syntactically different but semantically equivalent sequences and to generate less intrusive edit scripts. In turn, the resulting compactness of edit scripts reduces storage space and improves query efficiency in document management systems, while minimal intrusiveness lessens the rippling of effects and simplifies the tracking and understanding of changes for humans.

First experiments in [5, 6] provide evidence that *sdiff* can indeed significantly help to identify and reduce the effects of changes in a collection of documents.

Acknowledgments. While the definition of an equivalence system and the EQ syntax, respectively, as well as the definition of the *sdiff* algorithm is ours, its conception owes to discussions with Serge Autexier and Marc Wagner.

Chapter 6

Change Impact Analysis

One change always leaves the way open for the establishment of others.

— Niccolò Machiavelli

In a large modern enterprise, a rigorously defined framework is necessary to be able to capture the vision of an "entire system" in all its dimensions and complexity. All involved parts have a certain role in enterprise business processes and their relationships can be rather complex but are fundamental to achieve the goal of an enterprise.

The development and maintenance of these intrinsic complex business processes involves a large number of artifacts, capturing an entire system's requirements, designs, implementations, testing suites, and maintenance records. Traditionally, changes to such document collections — recall, we conceive business process to be *document-centric* (cf. Sec. 1.3) — are referred to those related to document maintenance. But even during the *initial* development of a system, there are on-going requirements for change due to changed requirements, error correction and development improvement. Therefore, the requirement for change is an inherent characteristic of the business development and maintenance process. In any case, the required changes are eventually reflected in changes to the artifacts involved and since these documents are logically related to each other, an initially proposed change may involve a large number of modifications to various depending artifacts. As such, the process of changes (cf. Sec. 1.3.2) within business processes needs to be managed and assisted by automated tools.

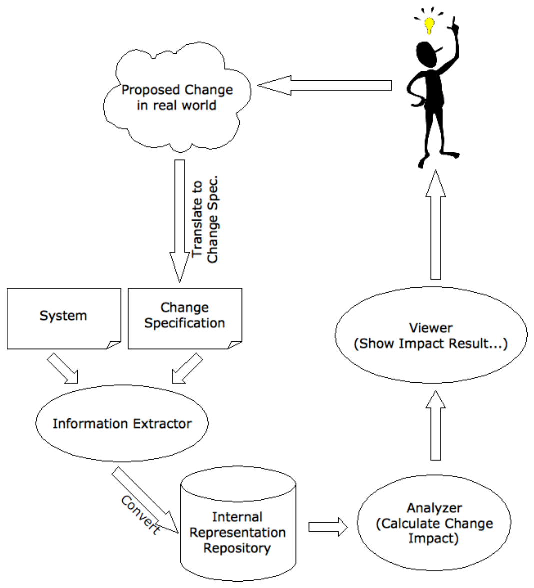
The issues of change management on semi-structured documents range from identifying the need for change, assessing the impact of a proposed change in an entire system, carrying out the initial and consequent modifications, managing the versions of the changed artifacts, through to collecting change-related data. In this chapter we concentrate on assessing the impact of a proposed change in an entire system, i.e. the ripple effect that a change may cause in a document collection. The goal of our change impact analysis slice (cf. Fig. 1.1) is to see what would happen if a change occurs, before the change really takes place. This information can then be used to help in making a decision on the necessity of a change. Carrying out the consequent modifications is described in Chap. 7.

6.1 Introduction

Traceability is the common term for mechanisms to record and navigate relationships between artifacts produced by development processes [88]. It is often used interchangeably with the term *requirements traceability*, which refers to the study of requirements throughout the whole product development life-cycle. The latter links requirements in the requirements analysis to design and realization documents that implement the requirements. It also links related requirements in the requirements analysis. An overview of recent traceability techniques in the field of requirements traceability can be found in [11, 56, 46]. One major challenge in this area is overcoming the heterogeneity of the document formats. Different formats are used at each stage of the development process, but the relationships between the different documents are nonetheless manifold. A more general approach to traceability between any heterogeneous artifacts not limited to requirements engineering can be found in [2].

An important distinction made in the field of traceability is the one between *traceable* and *traced* documents. A traced document is a document for which relationships have been identified and made explicit. If the relationships in a document can be deduced from the structure, the document is called traceable [148]. Usually a richer structure makes a document more traceable.

Traceability of a document is often considered a prerequisite for sophisticated *impact analysis*. The term impact analysis (IA) is used in many different contexts and it is not always clear what it comprises. Here impact analysis is taken to generally refer to the identification of potential consequences of a change. Therefore, we call this type of IA as *change impact analysis* (CiA). A typical change impact analysis process is illustrated in Fig 6.1. Regarding to Moreton [92], change impact analysis can be broken down into following stages: (1) convert proposed change into a system change specification, (2) extract information from information source and convert into internal representation repository, (3) calculate change impact for these change proposals; do stage (1) – (3) again for other competing change proposals, (4) develop resource estimates, based on considerations such as size and [...] complexity, (5) analyze the cost and benefits of the change request, in the same way as for a new application, and (6) the maintenance project manager advises the users of the implications of the change request, in business rather than in technical terms, for them to decide whether to authorize proceeding with the change.


Proposed Change in real world
Translate to Change Spec.
System
Change Specification
Viewer (Show Impact Result...)
Information Extractor
Convert
Internal Representation Repository
Analyzer (Calculate Change Impact)
Figure 6.1: A Typical Change Impact Analysis Process [92].

In this chapter we deal with the "Viewer" and the "Analyzer" with a focus on the major concept in this area, *rippling of effects*, which occurs when a comparatively small change to a document affects many other parts of the respective document collection. Our enabling idea is to represent in a single graph all related structured documents and employ graph rewriting techniques to identify the effects of changes.

6.1.1 Why use Graph Rewriting?

Many structures in computer science can be described by graphs: computer networks, communicating processes, pointer structures on the heap, UML diagrams, and many others. Typically, nodes represent objects or concepts, and edges represent relationships among them. Auxiliary information is expressed by adding attributes to nodes or edges. These graphs are mostly static descriptions of system states. However, our application domain, change management of semi-structured documents, requires adding dynamics to model the evolution of document collections over time.

Adding dynamics requires some means to describe state changes, such as graph rewriting rules. The notion of graph rewriting as understood within this work is a method for declaratively specifying "changes" to a graph. This is comparable to the well-known term rewriting. The theory of graph rewriting systems has its origins in the 1970's and a rich theory is now available [121]. In general, one or more graph rewrite rules are employed to accomplish a certain task. In the simplest case such a graph rewrite rule consists of a tuple $L \rightarrow R$, whereas L — the left hand side of the rule — is called pattern graph and R — the right hand side of the rule — is the replacement graph. The transformation is done by application of a rule to a host graph H, i.e. to find an occurrence of the pattern graph in the host graph. Afterwards the matched spot of the host graph is changed, such that it becomes an isomorphic subgraph of the replacement graph R.

Graph rewriting has found many applications in the modeling of concurrent systems and in other areas such as database design, bioinformatics and visual languages [43, 44]. For our concerns graph rewriting is well-suited too. The specification of dynamically evolving structures, say collection of documents, and possessing features such as dynamic creation of objects, say semantic annotations as well as impacts information, are intuitively expressible within graph rewriting system. Document parts of semi-structured documents considered as *fsp*-trees are represented by nodes, intra-/ as well as inter-relations are represented by edges, and an analysis of effects of modifications on documents are modeled by graph rewriting rules. Consequently, all of these aspects make graph rewriting systems suitable for an underlying specification language, on which we can base our fundamental methods for a change impact analysis on dynamically evolving structures.

Given that our focus is on change impact analysis and not on the building technique, we in the following give an overview of the state-of-the-art change impact analysis approaches instead of the history of graph rewriting. However, it is anticipated that none of these approaches, consider the technique of graph rewriting to analysis the rippling of effects.

6.1.2 A Short History of Change Impact Analysis Approaches

Modeling data, control, and component dependency relationships are useful ways to determine change impacts within the set of documents. The up to now considered change impact analysis techniques to support these kinds of dependencies are data flow analysis [74, 147, 60], data dependency analysis [93], control flow analysis [80, 89], program slicing [146, 64, 81, 76], test coverage analysis [38, 111, 112], cross referencing, browsing [13], semantics- and logic-based defects detection [78, 77, 126] and reverse engineering algorithms [68].

Below we will give a brief outline of the various approaches in order to distance ourselves from them and finally to discuss the benefits of our novel CiA approach. A more detailed explanation of the terminology and differences between CiA approaches can be found in [3].

Impact Analysis. The Yau and Patkow models are useful in evaluating the effects of change on the system to be maintained. Yau [157] focuses on software stability through analysis of the ripple effect of software changes. A distinctive feature of this model is the post-change impact analysis provided by the evaluation of ripple effect. This model of software maintenance involves (1) determining the maintenance objective, (2) understanding the program, (3) generating a maintenance change proposal, (4) accounting for the ripple-effects, and (5) regression testing the program. Rombach and Ulery [120] proposed a method for software maintenance improvement that focuses on the goals, questions, and specific measurements associated with activities in the context of a software maintenance organization. However, their method does not specify a framework that supports impact analysis in the software maintenance process. Pfleeger and Bohner [113] recognize impact analysis as a primary activity in software maintenance and present a framework for software metrics that could be used as a basis for measuring stability of the whole software system. The framework is based on a graph, called the traceability graph, which shows the interconnections among source code and test cases. It is anticipated, though, that the level of detail on the diagram is insufficient to make detailed stability measurements.

Inferencing. Marvel [72], an intelligent assistance for software development and maintenance, is an environment that supports two aspects of an intelligent assistant: it provides insight into the system and it actively participates in development through opportunistic processing. It has insight, which means it is aware of the user's activities and can anticipate the consequences of these activities based on an understanding of the development process and the produced software. It performs opportunistic processing, which means it undertakes simple development activities so programmers need not be bothered with them. It models the development process as rules that defines the preconditions and postconditions of development activities, and gathers collections of rules into strategies.

Control Flow, Data Flow and Data Dependency. Control flow tools identify calling dependencies, logical decisions, and other control information to examine control impact. Loyall and Mathisen [80] present a language-independent definition of inter-procedural dependence analysis and have implemented it in a prototype tool. Their prototype tool indicates different control dependencies among different procedures of a program. Moser [93] created a compositional method for constructing data dependency graphs for Ada programs based on composition rules. This method combines composition rule techniques with data dependency graphs to construct larger constructive units. These rules match other composition-based program development techniques, and enable data dependency graphs for complex programs to be constructed from the simpler graphs for the units of which they are composed. Moser examines composition rules for iteration, recursion, exception handling, and tasking. Graphs for primitive program statements are combined together to form graphs for larger program units. Keables, Roberson and Mayrhauser [74] presented an algorithm that limits the scope of recalculation of data flow information for representative program changes. Their prototype data flow analysis program works on a subset of the Ada language. The research project [29] at Arizona State University that started in 1983 tried to develop a practical software maintenance environment. The ASU tool operates on simplified Pascal programs that are expected to be error free. It displays the structure chart of the Pascal code, displays the parameters used in the module call and the global variables referenced in the current module. The McCabe Battlemap Analysis Tool (BAT) [89] decomposes source code into its control elements to create a view of the program that specifies the control flow for analysis.

Slicing. Program slicing provides a mechanism for constraining the view and behavior of a program to a specific area of interest [146, 64]. Program slices focus attention on small parts of the program by eliminating parts that are not essential for the evaluation of the specific variables at a certain location. Horwitz, Reps, and Binkley [64] concentrated their work on inter-procedural slicing, and generated a new kind of graph called the system dependence graph, which extends previous dependence representations to incorporate collections of procedures rather than just monolith programs. Their inter-procedural slicing algorithms were restricted to certain types of slices: rather than permitting a program to be sliced with respect to program point p and an arbitrary variable, a slice must be taken with respect to a variable that is defined or used at p. The Unravel tool developed by James Lyle of NIST [81] can be used to slice C programs.

Requirements Tracing. Requirement traceability (RT) is defined as the ability to describe and follow the life of a user requirement both in forward and backward direction. For instance, in many safety critical systems each part of an implementation has to be traced back to some initial requirements (cf. [122] regulating software development in avionics). Requirement Tracing (cf. e.g. [117]) comprises techniques to document, manage, and propagate requirements. Commercial systems support the capturing of requirements (providing textual/graphical interfaces to the user), allocating the requirements to system elements (allowing one to annotate weights, costs and risks), determining inconsistencies in terms of unlinked requirements or system parts, illustrating the (explicitly stated) dependencies, and journalizing the history of development (see [57] for an overview over the current state of the art). Current RT approaches, however, are either very general and do not offer tool support or are tied to a specific part of the software development process and a specific semantics of the objects and relations, which makes an adaptation of these tools impossible. Popular RT tools like the DOORS [136] and SLATE [42] system use script languages like TCL/TK, DXL or command line macros for this task.

System Ontologies and Consistency Management. In order to facilitate change management on XML documents, Krieg-Brückner et al. [78] describe the respective document format and its way to structure documents (i.e. the relations between the artifacts) in terms of a *system ontology*. This is an ontology describing the data model of a representation format independently of its respective syntactical realization. Krieg-Brückner et al. utilize system ontologies for the modeling of extensive semantic interrelation of documents in order to achieve sustainable software development [77]. Document Type Definitions and XML Schemata that are used to validate an XML document as a member of a document format are instances of such system ontologies: they specify the syntactical containment relations that yield the document tree. But there are also more elaborated system ontologies. For instance, the CDet [126] system for consistent document engineering allows to specify a system ontology of consistency rules such that the system is able to capture informal consistency requirements. In case of rule violation the system generates consistency reports as s-DAGs (suggestion directed acyclic graphs). These S-DAGs provide a convenient way to visualize inconsistencies and repair actions. However, system ontologies for document formats do not constitute a management of change in themselves, even though they may be extremely useful in detecting (and rejecting) changes that result in documents that are not consistent with the intended document format any more. Even the work in the system ontology based MMiSS project reported in [84] does not really go beyond this.

Development Graphs. Industrial applications of Formal Methods revealed that an efficient, evolutionary formal development approach is absolutely indispensable as it was hardly ever the case that the development steps were correctly designed in the first attempt. The search for formally correct software and the corresponding proofs is more like a formal reflection of partial developments rather than just a way to assure and prove more or less evident facts. Therefore, Dieter Hutter developed the formal notion of a development graph [66] as a logical representation of a consistent and structured formal software development. The nodes of such a development graph correspond to various entities of a formal development (like theories or modules) while links between nodes represent the given or postulated relations between them (e.g. “satisfies”, “implements”, or “using”). The development graph serves as a structured data base for a verification system which is incrementally built up during the software development process. The purpose of a development graph is twofold. On the one hand it provides a general representation language for structured specifications (and also as the underlying data base on which a theorem prover will operate) and on the other hand it serves as a truth maintenance system on the level of theories as it keeps track of the relations between different theories when changing the development. To support such a reuse of proof work, the development graph can only be changed by a limited set of basic operations which automatically keep track of the validity of the stored properties and proofs [7]. The idea of development graphs turned out to be very fruitful [67, 95, 124, 96] and various research groups adopted this approach. For example, Till Mossakowski at Bremen University extended the scope of development graphs towards supporting different logics (heterogeneous development graphs [62, 94]) and the proof semantics of CASL (Common Algebraic Specification Language) was defined with the help of development graphs [97]. In practice, the work on development graphs resulted in the implementation of the MAYA-system [8] and Hets [62, 94] to support the development of structured formal developments.

The listed approaches solely focused on the area of software engineering. Source code is exclusively included as the documents to be managed. Software artifacts in development processes, however, comprise requirements as well as documentations and all these documents are seldom isolated artifacts but are intentionally related and intertwined and as such represent a collection of documents to be managed. Changing one document within a collection requires possible adaptations to other documents and due to the huge amount of documents a collection may include, there is a need for a reliable and efficient system support. While dedicated authoring and maintenance tools ranging from simple text-editors to integrated development environments may provide some assistance when changing documents, they typically are restricted to single documents or documents of a specific type.

In order to resolve that discontinuity, we present a framework that embraces existing document types, allows for the declarative specification of semantic annotation and propagation rules inside and across documents of different types, and on that basis define semantic annotation and change impact analysis for heterogeneous collections of documents.

6.2 A Model for Impacts Identification Based on Graph Rewriting

The enabling idea is to represent in a single graph all related structured documents together with that part of their intentional semantics contained necessary to analyze specific semantic properties of the document, for instance consistency checks, as well as to analyze the impact of changes on these semantic properties. The whole framework is based on document models defining the syntax, semantics and annotation languages for specific document types as well as graph transformations to obtain the semantic annotation and to propagate the effect of changes for documents of this type. For the interaction between documents of different types our framework builds on interaction models specifying graph transformations propagating semantic information over document boundaries.

Our framework — also implemented in our *locutor* library — builds on top of the graph rewriting tool GrGen.NET [53], where annotation and propagation rules can be specified in the declarative GrGen.NET syntax and used to semantically annotate collections of documents and to analyze the impact of changes on the whole the collection. In the following we introduce our central concept for change impact analysis, semantic document impact graphs. Together with document models and interaction models these define our model for change propagation based on graph rewriting.

6.2.1 Semantic Document Impact Graphs

A *semantic document impact graph* is designed around the following idea: an SDI graph comprises both the syntactic parts of a document, such as the actual files which are, for instance, given in some XML format as shown in Fig. 5.10. Additionally, the graph contains a separated explicit representation of the intentional semantics of the file content, which can be similar to the actual syntax but also quite different; in general, this will be a qualitative representation of the semantic entities of these documents and the relationship among them. For instance, these are the guests from the guest list and links indicating which guests belong together. The idea of the semantic entities is that the semantic entity for a specific guest remains the same, even if its syntactic structure may change. For instance, the semantic entity of guest `Manfred` remains the same even though the subtree is changed by updating the `confirmed` status.

That way each SDI graph by design has interesting properties, which can be methodologically exploited for the change impact analysis: indeed, it can contain parts in the document subgraph, for which there exists no semantic counter-part, which can be exploited during the analysis to distinguish added from old parts. Conversely, the semantic graph may contain parts, which have no syntactic origin, that is they are *dangling semantics*. This can be exploited during the analysis to distinguish deleted parts of the semantics from preserved parts of the semantics. The information about added and deleted parts in an SDI graph is the basis for rippling the effect of changes throughout the semantic graph structure. This exemplifies the benefit of the dual representation of the documents with their syntactic structure and their intentional semantics in separate graph, and we call this the *explicit semantics method*.

Thus a SDI graph of a document collection will consists of the documents, the semantic information computed from the documents, and the impact information containing explicit information how the document semantics is affected. This is represented by dividing the entire graph into subgraphs according to the specific node types and edge types: we introduce semantic node and edge types ($\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}$) for the semantic parts and impact node types and edge types ($\underline{\mathbb{V}}_{\text{imp}}, \underline{\mathbb{E}}_{\text{imp}}$) for impact information parts. For the document parts we use the previously imposed syntactic node and edge types ($\underline{\mathbb{V}}_{\text{syn}}, \underline{\mathbb{E}}_{\text{syn}}$).

An entire semantic document impact graph $\overleftarrow{\mathcal{D}}$ is a typed graph with respect to all these node types and edge types: (1) the *document graph* is the projection of $\overleftarrow{\mathcal{D}}$ to the syntactic node and edge types and must be typed graph, (2) the *semantics graph* is the projection of $\overleftarrow{\mathcal{D}}$ to the semantic node and edge types and must also be a typed graph, and finally (3) the *impacts graph* is obtained by projecting $\overleftarrow{\mathcal{D}}$ on the impact node and edge types which must not form a typed graph on its own but all edges must connect one node from the impacts graph to a node from the document graph.

Definition 6.1 (SDI Graph). Let $\underline{\mathbb{V}} = \underline{\mathbb{V}}_{\text{syn}} \uplus \underline{\mathbb{V}}_{\text{sem}} \uplus \underline{\mathbb{V}}_{\text{imp}}$ be the disjoint union of syntactic, semantic and impact node types and $\underline{\mathbb{E}} = \underline{\mathbb{E}}_{\text{syn}} \uplus \underline{\mathbb{E}}_{\text{sem}} \uplus \underline{\mathbb{E}}_{\text{imp}}$ the disjoint union of syntactic, semantic and impact edge types. A $(\underline{\mathbb{V}}, \underline{\mathbb{E}})$-typed graph $\overleftarrow{\mathcal{D}} = \langle \mathbb{V}, \mathbb{E}, \prec \rangle$ is a **semantic document impact graph** if and only if

(1) the **document graph** $\overleftarrow{\mathcal{D}}_{|\underline{\mathbb{V}}_{\text{syn}},\underline{\mathbb{E}}_{\text{syn}}}$ is a collection of $(\underline{\mathbb{V}}_{\text{syn}},\underline{\mathbb{E}}_{\text{syn}})$-typed *fsp*-trees.[1]

(2) the **semantics graph** $\overleftarrow{\mathcal{D}}_{|\underline{\mathbb{V}}_{\text{sem}},\underline{\mathbb{E}}_{\text{sem}}}$ is a $(\underline{\mathbb{V}}_{\text{sem}},\underline{\mathbb{E}}_{\text{sem}})$-typed graph

(3) the **impacts graph** $\overleftarrow{\mathcal{D}}_{|\underline{\mathbb{V}}_{\text{imp}},\underline{\mathbb{E}}_{\text{imp}}}$ is a $(\underline{\mathbb{V}}_{\text{imp}},\underline{\mathbb{E}}_{\text{imp}})$-typed pre-graph and for all $v \rightarrowtail v' \in \mathbb{E}_{\underline{\mathbb{E}}_{\text{imp}}}$ it holds: $v \in \mathbb{V}_{\underline{\mathbb{V}}_{\text{imp}}}$ and $v' \in \mathbb{V}_{\underline{\mathbb{V}}_{\text{syn}}}$ (or vice-versa).

We agree on the following notation for SDI graphs that makes the three parts document graph $\mathcal{O}$, semantics graph $\mathcal{S}$, and impacts graph $\mathcal{I}$ explicit: $\overleftarrow{\mathcal{D}} = \langle \mathcal{O}, \mathcal{S}, \mathcal{I} \rangle$.

In our wedding example, a document subgraph represents the tree-structured syntactical content of the documents. The semantics graph holds information, for example, of the individual guests and if they have been added, deleted or maintained. Moreover, it links guests with respect to their relationship. The impacts subgraph marks the ripple effects resulting from modifications, for example, on a guest's status.

$$\begin{tikzcd}
\langle \mathcal{O}, \bullet, \bullet \rangle \arrow[d, "\text{annotate}"] \\
\langle \mathcal{O}, \mathcal{S}, \mathcal{I} \rangle
\end{tikzcd}$$

(a) Semantic Annotation

$$\begin{tikzcd}
\langle \mathcal{O}, \mathcal{S}, \bullet \rangle \arrow[d, "\text{patch}"] \\
\langle \mathcal{O}', \mathcal{S}, \bullet \rangle \arrow[d, "\text{annotate}"] \\
\langle \mathcal{O}', \mathcal{S}', \mathcal{I} \rangle
\end{tikzcd}$$

(b) Change Impact Analysis

Figure 6.2: Semantics-Based Analysis Techniques.

Graph Rewriting on SDI Graphs

Given a graph, we want to denote transformations of this graph into a new graph. In general given a typed graph $\mathcal{G}$ a graph transformation results in a new typed graph and given some node and edge types $\underline{\mathbb{V}}$ and $\underline{\mathbb{E}}$, a graph transformation is a total function with $\mathbb{G}^{\underline{\mathbb{E}}}_{\underline{\mathbb{V}}} \to \mathbb{G}^{\underline{\mathbb{E}}}_{\underline{\mathbb{V}}}$. However, we want to distinguish different kinds of graph transformation with respect to how they affect the document graph, the semantics graph and the impacts graph of a SDI graph. We consider the following two basic graph transformations (cf. Fig. 6.2):

Semantic Annotation is a graph transformation starting with an SDI graph with empty impacts graph and returns an SDI graph with an updated semantics graph and possibly some updated impacts graph. We will also write "annotation" instead of "semantic annotation", if the context is clear.

Change Impact Analysis is a graph transformation starting with an SDI graph with empty impacts graph, applies a *patch graph transformation* which only affects the document graph, followed by a semantic annotation graph transformation.

This completes the kernel theory for our management change methodology up to adjustment (cf. Chap. 7). We are now concerned with refining in order to meet the real world.

[1]A collection of typed trees (cf. Sec. 2.1) is naturally extended to *fsp*-trees (cf. Note 3.1)

6.2.2 Document Models

For adaptation to real conditions, we have to refine these two basic graph transformations for the application context, where we want to treat collections of documents, which are each of a specific document type. For the refinement we have to answer the following questions:(1) Where does a patch transformation come from? (2) Where does the annotation transformation come from?

For the patches we rely on a generic tree patch mechanism, like [15], and our *sdiff* difference analysis algorithm (cf. Sec. 5.2.2) taking equivalence specifications of documents into account. This stronger notion of equality leads to more compact, less intrusive edit scripts so that it is possible to ignore semantically insignificant differences being irrelevant for a change impact analysis.

$$\begin{tikzcd}
\langle \mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \mathcal{S}, \bullet \rangle \arrow[d, "\text{patch}(\delta_1),\ldots,\text{patch}(\delta_n)"] \\
\langle \mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}, \bullet \rangle \arrow[d, "\text{annotate}"] \\
\langle \mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}', \mathcal{I} \rangle
\end{tikzcd}$$

Figure 6.3: Patch Transformations.

We will describe these semantic properties combined with the graph transformations in *document models*. For a given collection of documents we assign specific document models to each document type. In order to determine the changes between two versions of a collection of documents we use the individual equivalence specifications to determine the changes between associated documents within the collection. Refining the change impact analysis mechanism representation from Fig. 6.2b, we assume $\mathcal{O} = \mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n$ and we get a collection of patch descriptions $\delta_1, \ldots, \delta_n$ for each document $\mathscr{T}_i$. The refined change impact analysis is illustrated in Fig. 6.3.

The intentional semantics of documents of a specific type is represented explicitly in semantics graphs with specific node and edge types and defined in *semantic models*. Furthermore, a semantic model defines the *impact* nodes and edges to annotate the syntactic parts of a semantic document impact graph.

Definition 6.2 (Semantic Model). Let $\mathscr{D} = \langle \underline{\mathbb{V}}_{\text{syn}}, \underline{\mathbb{E}}_{\text{syn}}, P \rangle$ be a document type specification, then $\mathscr{S} = \langle \mathscr{D}, (\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}), (\underline{\mathbb{V}}_{\text{imp}}, \underline{\mathbb{E}}_{\text{imp}}) \rangle$ is a **semantic model for** $\mathscr{D}$ if $\underline{\mathbb{V}}_{\text{syn}}$, $\underline{\mathbb{V}}_{\text{sem}}$ and $\underline{\mathbb{V}}_{\text{imp}}$ are pairwise disjoint node types and $\underline{\mathbb{E}}_{\text{syn}}$, $\underline{\mathbb{E}}_{\text{sem}}$, and $\underline{\mathbb{E}}_{\text{imp}}$ are pairwise disjoint edge types. Two semantic models are **disjoint** if their respective document type specifications, semantic node types and edge types as well as impact node types and edge types are pairwise disjoint.

We agree to denote by $\mathscr{S}_{\mathscr{D}}$ that the semantic model $\mathscr{S}$ belongs to the document type specification $\mathscr{D}$. For our wedding example, the semantic model for the guest list contains a node type for individual persons and an edge type for their relationship.

To compute the semantic annotation we use graph rewriting rules that operate on SDI graphs. Like the document type specific equivalence specifications these rewriting rules are document type specific and are also part of a document model. The overall semantic annotation mechanism is entirely parametric in these document type specific graph rewriting rules. In our wedding example, these are, for instance, rules that lift syntactic entries in the guest lists to semantic objects in the semantic model and stores the origin of semantic nodes. However, if we assume an additional document, for example, a seating arrangement, we cannot yet check the seatings, as we have no links from persons to seats in the semantic graph. Indeed, so far, semantic models only allow us to semantically annotate single documents, but not to semantically annotate across document boundaries. For these we introduce *interaction models* (cf. Sec. 6.2.3) for a set of document models that specify annotation systems that operate between such documents. In our wedding example, these would be graph rewrite rules that lift the declarative assignment of guests to seats given in the documents to semantic links between the respective individual persons and seats obtained before by the document model annotation rules and checks consistency of seating arrangements, for example, the condition of guests being rotationally arranged by gender. This in turn makes it unclear how to combine the different annotation graph transformations: indeed, some information propagated across document boundaries needs to be propagate further inside these documents, whereas some information needs to be first propagated inside the documents before the information can be propagate further to other documents using the propagation information from an interaction model.

Thus, the introduction of cross-document interaction models requires the specification how the different graph transformations must be orchestrated. To this end we subdivide the annotation graph transformations methodologically into three phases, namely

(i) an *abstraction* phase which synchronizes the semantics graph with the (new) document trees,

(ii) a *propagation* phase which propagates the information inside the semantics graph only, and

(iii) a *projection* phase which first dumps the information from the semantics graph into the impacts graph and then cleans up the semantics graph with respect to obsolete abstraction information

Note 6.1. The dump of impacts into the document graph is discussed in Chap. 7.

Note 6.2. A abstraction graph transformation pre-requires *status information* on nodes as well as on edges. At the current status of this work this must be handled on the level of graph transformations and the declaration of semantic node and edge types, respectively. We suggest to enrich each semantic node and edge type by a `status` slot with domain `new`, `preserved`, and `deleted`. For example, if a syntactical node has been removed from a document, then an abstraction on the new document in combination with respective semantics graph of the previous version of the document must not delete the corresponding semantic nodes and edges of the removed syntactic node but set the status of those, for instance, to `deleted`. Consequently, fresh semantic nodes and edges get the status `new` and `preserved` otherwise. For illustration we refer to Ex. 6.1.

The overall annotation graph transformation then consists of the three phases, which results in the detailed model of the change impact analysis depicted in Fig. 6.4. The abstraction, propagation and projection graph transformations are defined for each document model as well as for each interaction model such that

(i) in the abstraction phase first all abstraction graph transformations of each involved document model are executed, followed by the abstraction graph transformations of the interaction models,

$$\begin{tikzcd}
(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \mathcal{S}, \bullet) \arrow[d, "\text{patch}(\delta_1),\ldots,\text{patch}(\delta_n)"] & \\
(\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}, \bullet) \arrow[r, "\text{abstraction}"] \arrow[d, dashed, "\text{annotate}"] & (\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}', \bullet) \arrow[d, "\text{propagation}"] \\
(\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}''', \mathcal{I}) & (\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}'', \bullet) \arrow[l, "\text{projection}"']
\end{tikzcd}$$

Figure 6.4: Change Impact Analysis in Detail.

(ii) in the propagation phase all propagation graph transformations are applied until completion, and

(iii) in the projection phase the projection graph transformations for each document are applied individually to store the impact information in the impacts graph and at the very end the semantics graph is cleaned up with respect to obsolete abstraction information, i.e. removal of semantic nodes and edges with status `deleted`.

Following the general methodology, the annotation graph transformation for a specific document type is subdivided into abstraction, propagation and projection graph transformations combined in the concept of annotation models defined as follows:

Definition 6.3 (Annotation Model). Let $\mathscr{S} = \langle \mathscr{D}, (\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}), (\underline{\mathbb{V}}_{\text{imp}}, \underline{\mathbb{E}}_{\text{imp}}) \rangle$ be a semantic model, then an **annotation model $\mathscr{A}$ for $\mathscr{S}$** is a tuple $\langle \alpha_{\mathscr{S}}, \sigma_{\mathscr{S}}, \iota_{\mathscr{S}} \rangle$ of graph transformations of the following form:

Abstraction: $\alpha_{\mathscr{S}} : \wp(\mathbb{FS}^+_{\mathscr{D}}) \times \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}} \to \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}}$ is a mapping synchronizing the semantics graph with the document graph where fresh nodes and edges get the status `new`, removed nodes and edges get the status `deleted`, and `preserved` otherwise. Its homomorphic extension $\alpha^{\#}_{\mathscr{S}}$ to SDI graphs is only applicable on semantic document impact graphs with empty impacts graph and defined by $\alpha^{\#}_{\mathscr{S}}(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n \uplus \mathscr{T}, \mathcal{S}, \bullet) = (\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n \uplus \mathscr{T}, \mathcal{S}', \bullet)$ where $\mathscr{T}$ contains no documents of type $\mathscr{D}$ (i.e., $\mathscr{T} \cap \mathbb{FS}^+_{\mathscr{D}} = \emptyset$), $\mathcal{S}' := (\mathcal{S} \setminus \mathcal{S}_{|\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}}) \cup \alpha_{\mathscr{S}}(\{\mathscr{T}_1, \ldots, \mathscr{T}_n\}, \mathcal{S}_{|\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}})$, and $\mathscr{T}_i \in \mathbb{FS}^+_{\mathscr{D}}$.

Propagation: $\sigma_{\mathscr{S}} : \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}} \to \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}}$ propagates semantic information in the semantics graph. Its homomorphic extension $\sigma^{\#}_{\mathscr{S}}$ to semantic document impact graphs is only applicable on semantic document impact graphs with empty impacts graph and defined by $\sigma^{\#}_{\mathscr{S}}(\mathcal{O}, \mathcal{S}, \bullet) = (\mathcal{O}, \mathcal{S}', \bullet)$ where $\mathcal{S}' := (\mathcal{S} \setminus \mathcal{S}_{|\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}}) \cup \sigma_{\mathscr{S}}(\mathcal{S}_{|\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}})$.

Projection: $\iota_{\mathscr{S}} : (\wp(\mathbb{FS}^+_{\mathscr{D}}) \times \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}}) \to (\wp(\mathbb{FS}^+_{\mathscr{D}} \times \mathbb{G}^{\underline{\mathbb{E}}_{\text{imp}}}_{\underline{\mathbb{V}}_{\text{imp}}}) \times \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}})$ builds up the impacts graphs for these documents and cleans up the semantics graph with respect to node and edge types of status `deleted`. That is if $\iota_{\mathscr{S}}(\{\mathscr{T}_1, \ldots, \mathscr{T}_n\}, \mathcal{S}) = (\{(\mathscr{T}_1, \mathcal{I}_1), \ldots, (\mathscr{T}_n, \mathcal{I}_n)\}, \mathcal{S}')$, then for all $1 \leq i \leq n$ holds that $\mathscr{T}_i \uplus \mathcal{I}_i$ are typed graphs (not pre-graphs) and $\mathcal{S}'$ is a semantics graph where the status of all nodes and edges is either `new` or `preserved`. Its homomorphic extension $\iota^{\#}_{\mathscr{S}}$ to semantic document impact graphs is defined by $\iota^{\#}_{\mathscr{S}}(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n \uplus \mathscr{T}, \mathcal{S}, \mathcal{I}) = (\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n \uplus \mathscr{T}, \mathcal{S}', \mathcal{I} \uplus \mathcal{I}_1 \uplus \ldots \uplus \mathcal{I}_n)$ where $\mathscr{T}_i \in \mathbb{FS}^+_{\mathscr{D}}$, $\mathscr{T} \cap \mathbb{FS}^+_{\mathscr{D}} = \emptyset$ and $(\{(\mathscr{T}_1, \mathcal{I}_1), \ldots, (\mathscr{T}_n, \mathcal{I}_n)\}, \mathcal{S}') = \iota_{\mathscr{S}}(\{\mathscr{T}_1, \ldots, \mathscr{T}_n\}, \mathcal{S}_{|\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}})$.

We agree to denote by $\mathscr{A}_{\mathscr{S}}$ that the annotation model $\mathscr{A}$ belongs to the semantic model $\mathscr{S}$ and subsequently will not distinguish the mappings of the annotation models and their homomorphic extensions.

Note 6.3. The propagation is defined for a document type (i.e. $\mathscr{D}(\mathscr{S})$) and thus already shall include the propagation of semantic information between different documents of the same document type. The correlation between the syntactic part and the semantics/ impacts part of a document is solved by the assumption that each document has its own document model, and if there are actually two documents with the same model, then, for example, one is completely renamed. In the realization, however, the correlation has to be explicitly stored such that, for example, projections only project into the correlating syntactic documents. Therefore we suggest to enrich semantic nodes as well as impact nodes with an `origin` slot comprising the respective URL combined with the respective XPath of the corresponding syntactic node.

With the introduction of the semantic entities of equivalence systems, semantic models, and annotation models, we can now summarize these under the concept of document type specific document models.

Definition 6.4 (Document Model). Let $\mathscr{D}$ be a document type specification, $\mathscr{E}_{\mathscr{D}}$ an equivalence specification for $\mathscr{D}$, $\mathscr{S}$ a semantic model for $\mathscr{D}$ and $\mathscr{A}$ an annotation model for $\mathscr{S}$. Then $\mathscr{M} = \langle \mathscr{E}_{\mathscr{D}}, \mathscr{S}, \mathscr{A} \rangle$ is a **document model for** $\mathscr{D}$. Two document models are **disjoint** if their respective semantic models are disjoint.

We agree to denote by $\mathscr{M}_{\mathscr{D}}$ that the document model $\mathscr{M}$ belongs to the document type specification $\mathscr{D}$.

Example 6.1. Now with the definition of a document model let us perform a change impact analysis on our wedding example (cf. Ex. 5.2) to get a comprehensive understanding. We start our example with a document type specification as the basis for the following semantic model which in turn represents the basis for the required annotation model. Together with the equivalence system introduced in Fig. 5.11 the individual components, result in the document model $\mathscr{M}$ for our guest list.

Note 6.4. All type declarations as well as all graph transformations presented here are written concretely in the respective GrGen.NET language [12], i.e. node and edge type declarations are written in the GrGen.NET graph model language and graph transformations are written in the GrGen.NET rule language. Only for readability we marginally polished GrGen.NET's notion of connection assertions and utilized subscripts. The complete example code is available at [99].

As the `guests.xml` document has no explicit referenced grammar rules, like a DTD, the document type specification $\mathscr{D}$ of `guests.xml` is inferred by the XML labels and the XML structure of the document, respectively. Figure 6.5 depicts the respective node types and edge types utilizing the GrGen.NET graph model language. The keywords `node class` define a new node type. Node types can inherit from other node types defined within the same model via the `extends` clause. Optionally nodes can possess attributes. The keywords `edge class` define a new edge type. Again, edge types can inherit from other edge types defined within the same model via the `extends` clause and optionally edges can possess attributes. A connection assertion `connect` specifies that certain edge types only connect specific nodes a given number of times whereas `[+]` is an abbreviation for `[1:*]`. In order to apply the connection assertions of the supertypes to an edge type, one may use the keywords `copy extends`. The `copy extends` assertion "imports" the connection assertions of the direct ancestors of the declaring edge. This is a purely syntactical simplification, i.e. the effect of using `copy extends` is the same as copying the connection assertions from the direct ancestors by hand. The keyword `abstract` indicates that one cannot instantiate graph elements of this type. Instead one has to derive non-abstract types to create graph

node class $\underline{\mathbb{f}}$ { url : string ; }
directed edge class $\underline{\ni}$ **connect** $\underline{\mathbb{f}} \rightarrow \underline{\mathbb{f}}[+]$;
abstract node class $\underline{\mathbb{V}}_{\text{syn}}$ **extends** $\underline{\mathbb{f}}$;
node class root **extends** $\underline{\mathbb{V}}_{\text{syn}}$;
node class guests **extends** $\underline{\mathbb{V}}_{\text{syn}}$;
node class header **extends** $\underline{\mathbb{V}}_{\text{syn}}$;
node class title **extends** $\underline{\mathbb{V}}_{\text{syn}}$;
node class hosts **extends** $\underline{\mathbb{V}}_{\text{syn}}$;
node class host **extends** $\underline{\mathbb{V}}_{\text{syn}}$ { gender:string; text:string; }
node class body **extends** $\underline{\mathbb{V}}_{\text{syn}}$;
node class person **extends** $\underline{\mathbb{V}}_{\text{syn}}$ { gender:string; confirmed:string; text:string; }
directed edge class $\underline{\text{ɔ}}$ **extends** $\underline{\ni}$ **connect copy extends**;

Figure 6.5: The Document Type Specification $\mathscr{D} = \langle \underline{\mathbb{V}}_{\text{syn}}, \underline{\mathbb{E}}_{\text{syn}}, P \rangle$ of `guests.xml`.

elements. With respect to the requirements of *fsp*-trees (cf. Def. 3.1) our basic graph element types are the solid node and edge types $\underline{\mathbb{V}}_{\text{sld}}$ and $\underline{\mathbb{E}}_{\text{sld}}$, respectively, preluded as supertypes to any chosen set of node types and edge types. As previously mentioned, the XML labels of `guest.xml` frame the syntactic node types ($\underline{\mathbb{V}}_{\text{syn}}$) and respective XML attributes become type attributes, like, for instance, the XML `gender` attribute of an XML `host` element. The syntactic edge types ($\underline{\mathbb{E}}_{\text{syn}}$) are represented by the simple extension $\underline{\text{ɔ}}$ of the solid edge type $\underline{\ni}$. The predicate $P(\mathscr{D})$, not explicitly listed here, is represented by the XML well-formedness criteria (cf. Sec. 2.3).

enum Status {new, preserved, deleted}
abstract node class $\underline{\mathbb{V}}_{\text{sem}}$ { origin:string; status:Status = new; }
node class bride **extends** $\underline{\mathbb{V}}_{\text{sem}}$;
node class groom **extends** $\underline{\mathbb{V}}_{\text{sem}}$;
node class witness **extends** $\underline{\mathbb{V}}_{\text{sem}}$;
directed abstract edge class $\underline{\mathbb{E}}_{\text{sem}}$ **connect** $\underline{\mathbb{V}}_{\text{sem}}[+] \rightarrow \underline{\mathbb{V}}_{\text{sem}}[+]$ { status:Status = new; }
directed edge class assists **extends** $\underline{\mathbb{E}}_{\text{sem}}$ **connect** witness $\rightarrow \underline{\mathbb{V}}_{\text{sem}}$;
directed edge class hasRings **extends** $\underline{\mathbb{E}}_{\text{sem}}$ **connect** witness $\rightarrow \underline{\mathbb{V}}_{\text{sem}}$;
edge class panic **extends** $\underline{\mathbb{E}}_{\text{sem}}$ **connect** $\underline{\mathbb{V}}_{\text{sem}} \longleftrightarrow \underline{\mathbb{V}}_{\text{sem}}$;
abstract node class $\underline{\mathbb{V}}_{\text{imp}}$;
node class goShopping **extends** $\underline{\mathbb{V}}_{\text{imp}}$;
node class newWitness **extends** $\underline{\mathbb{V}}_{\text{imp}}$;
directed abstract edge class $\underline{\mathbb{E}}_{\text{imp}}$ **connect** $\underline{\mathbb{V}}_{\text{syn}} \rightarrow \underline{\mathbb{V}}_{\text{imp}}$;
edge class haveTo **extends** $\underline{\mathbb{E}}_{\text{imp}}$ **connect copy extends**;
edge class needs **extends** $\underline{\mathbb{E}}_{\text{imp}}$ **connect copy extends**;

Figure 6.6: The Semantic Model $\mathscr{S} = \langle \mathscr{D}, (\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}), (\underline{\mathbb{V}}_{\text{imp}}, \underline{\mathbb{E}}_{\text{imp}}) \rangle$ for $\mathscr{D}$.

Regarding the declaration of a semantic model we first must consider which semantic information is contained in a guest list and which of them come for us within our scenario. For this we take a simple scenario at hand: we want to annotate the witnesses of the bride and the groom, respectively, and we want to make sure that one of the witnesses has the wedding rings. If this is actually not the case, then the couple will be in "panic mode" and should get new rings as soon as possible. The semantic as well as the impact node types and edge types are depicted in Fig. 6.6. Following the advice at Note 6.2, our semantic model begins with the declaration of the possible states of semantic node types and edge types utilizing GrGen.NET enumeration declarations. An enumeration type is a collection of so called enumeration items that are associated with integral numbers, each. The `new` enumeration item is utilized to markup fresh semantic node types and edge types, the `preserved` enumeration item is utilized to identify existing semantic node types and edge types during an abstraction graph transformation, and the `deleted` enumeration item is utilized to markup semantic node types and edge types the corresponding syntactic item has been removed from the document. These status information represent the domain of the type attribute `status` of a semantic node type $\underline{\mathbb{V}}_{\text{sem}}$. For syntactic and semantic correspondence (cf. Note 6.3) we employ an `origin` slot pointing to the respective syntactic item via its URL and XPath. In order to represent a bride, a groom, and the witnesses, we introduce the respective semantic node types `bride`, `groom`, and `witness`. In order to interrelate nodes of those types we introduce the semantic edge types `assists` and `hasRings` extending the abstract edge class $\underline{\mathbb{E}}_{\text{sem}}$. The former represents the basic interrelation between a bride and a groom, respectively, and a witness. The fact that one of the witnesses must have the wedding rings is modeled by the latter semantic edge type `hasRings`. The impact node types and edge types are employed to model the fact that if a witness lost the wedding rings, the couple has to go shopping for new wedding rings (`goShopping`) and in case a witness disappeared at all, the bride or the groom has to look for a new witness (`newWitness`).

rule α {
  pattern {}
  modify {**exec**(syncStatus* & witness2bride & witness2groom);}}
rule σ {
  pattern {w:witness $\xrightarrow{e:\underline{\mathbb{E}}_{\text{sem}}}$ b:bride; **if** {((w.status==deleted) || (e.status==deleted));} g:groom;}
  modify {b $\overset{:\text{panic}}{\longleftrightarrow}$ g;}}
rule ι {
  pattern {b:bride $\overset{:\text{panic}}{\longleftrightarrow}$ g:groom; cm:host $\xrightarrow{:\underline{\mathfrak{I}}}$ hs:hosts $\xleftarrow{:\underline{\mathfrak{I}}}$ nm:host; **if** {corr(cm,b) && corr(nm,g);}
  modify {gs:goShopping; nw:newWitness; hs $\xrightarrow{:\text{haveTo}}$ gs; cm $\xrightarrow{:\text{needs}}$ nw;}}

Figure 6.7: The Annotation Model $\mathscr{A} = \langle \alpha, \sigma, \iota \rangle$ for $\mathscr{S}_{\mathscr{D}}$.

Regarding the annotation model we utilize the GrGen.NET rule language. The rule language forms the core of GrGen.NET. A rule set refers to zero or more graph models and specifies a set of graph rewrite rules. The rule language covers a pattern specification and a replace/modify specification. Attributes of graph elements can be re-evaluated during an application of a rule. The part introduced by the keyword `pattern` denotes the pattern graph consisting of typed nodes and/or edges. Anonymous nodes and edges can be denoted by a `:<type>` declaration. With negative application conditions (keyword `negative`) one can specify graph patterns which forbid the application of a rule if one of them is present in the host graph. The attribute conditions (keyword `if`) in a pattern part allows for further restriction of the applicability of a rule. If a rule is applied, then the `modify` part modifies attributes of matched nodes and edges as well as inserts new graph elements. Figure 6.7 illustrates the three graph transformations, abstraction, propagation, and projection in the GrGen.NET rule language with respect to the annotation model for our wedding example. At the beginning of an abstraction graph transformation (`rule` α) the status of all semantic node types and edge types is synchronized (`rule syncStatus`). Again, fresh semantic node types and edge types get the status `new`, identified ones get the status `preserved` and `deleted` otherwise. In the second step of an abstraction the bride and her witness are identified and added to the semantics graph (`rule witness2bride`). The same holds for the groom and his witness (`rule groom`). In addition, the `witness2bride` rule states the fact that the witness of the bride holds the wedding rings. The propagation (`rule` σ) checks for the witness of the bride and in case she disappeared some how marks the bridal couple to be in panic mode. Eventually, the projection (`rule` ι) dumps the semantic information into the impacts graph stating in case of panic mode that the hosts corresponding to the bridal couple (`corr(cm,b)` resp. `corr(nm,g))` have to go shopping for new wedding rings and the host corresponding to the bride has to look for a new witness. For readability we left out the graph transformations `syncStatus`, `witness2bride`, `witness2groom`, and `corr`. These graph transformations are also available at [99]. Note that the knowledge of which host and person, respectively, represent the bride, groom or the assisting witness is encoded within the pattern/ modify part of these graph transformations. However, one could imagine to employ one of the metadata schemes discussed in Sec. 4.1.2 to generalize those rules, i.e. make them independent off our concrete wedding example.

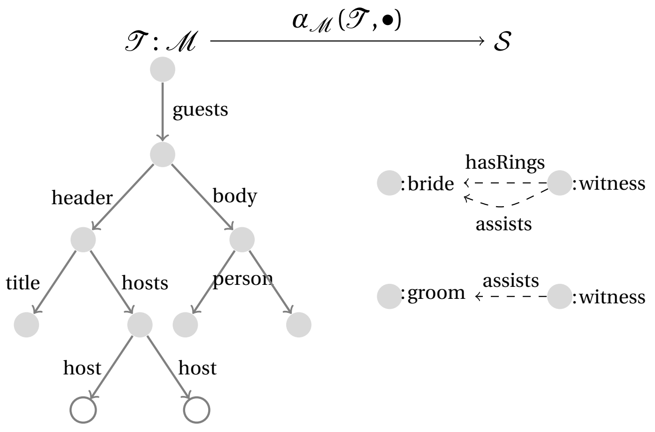


Figure 6.8: The Initial Abstraction on `guest.xml`.

Now that we have all the components of a document model $\mathcal{M}$ present in concrete form, let us perform a specific change impact analysis. Figure 6.8 depicts the initial abstraction graph transformation whereas we denote by $\mathcal{T} : \mathcal{M}$ the fact that $\mathcal{M}$ is associated to $\mathcal{T}$ and by ●:$\mathbb{t}$ the fact that node ● has type $\mathbb{t}$.

The left side in Fig. 6.8 illustrates our guest list document whereas, for convenience, we left out the text nodes as well as the subtrees of the two `person` elements which in turn represent the two witnesses, the only invited guests. The actual content of the XML text nodes and XML `person` elements, however, is encoded in the type attribute `text` of syntactic nodes of type `host` and `person`, respectively (cf. Fig 6.5). For instance, the `text` slot of the syntactic node of type `host` at XPath `/*[1]/*[1]/*[2]/*[1]` holds the string `Christine Mueller`.

The right side of Fig. 6.8 represents the resulting semantics graph after applying the graph transformation α on the document $\mathcal{T}$ and the empty semantics graph $\bullet$. The resulting semantics graph $\mathcal{S}$ represents the fact that the witnesses assist the bride and the groom, respectively, and one of the witnesses holds the wedding rings. The respective `origin` slots hold the URL/XPath of the corresponding syntactic node. For instance, the semantic node of type `bride` refers to the syntactic node of type `host` via `file:///home/nmueller/vc/m2.de/wedding/guests.xml/*[1]/*[1]/*[2]/*[1]`.

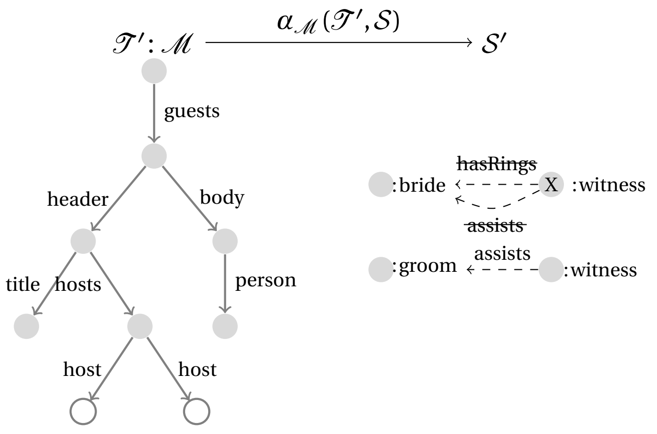


Figure 6.9: The Subsequent Abstraction on `guest.xml`.

Figure 6.9 demonstrates the next status of our scenario: one of the witnesses has been removed from the document resulting in the next document version $\mathcal{T}'$. The subsequent abstraction graph transformation on $\mathcal{T}'$ and the previously computed semantics graph $\mathcal{S}$ results in the semantics graph $\mathcal{S}'$ depicted on the right side of Fig. 6.9. For readability we marked semantic nodes with status `deleted` by an `X` and regarding deleted semantic edges we crossed out the respective edge type.

The result of the propagation of these changes on the semantics graph $\mathcal{S}'$ is depicted on the left side of Fig. 6.10. As desired in our scenario, the propagation graph transformation identifies the bridal couple to be in panic mode resulting in the semantics graph $\mathcal{S}''$. At this point it is clear that an abstraction graph transformation must not delete semantic nodes without syntactic correspondence. Only on document collections with corresponding marked semantics graph, i.e. a semantics graph with nodes labeled as to their status, a change propagation is feasible. Otherwise, we lose essential information on the semantic level, for example, the fact that from the syntactic level nodes were deleted. Clearly, only after a complete semantic annotation the semantic level can be cleaned up.

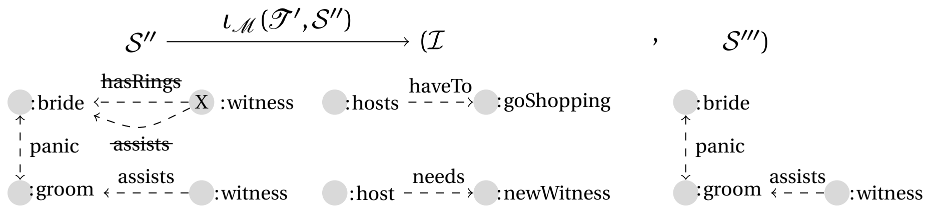


Figure 6.10: The Projection of Changes into the Impacts Graph $\mathcal{I}$.

The final phase of our change impact analysis, the projection of the semantics graph $\mathcal{S}''$ into the impacts graph $\mathcal{I}$ is illustrated on the right side of Fig. 6.10. The resulting impacts graph $\mathcal{I}$ denotes the fact that the bridal couple has to go shopping for new wedding rings and the bride as to look for a new witness. In addition the semantics graph $\mathcal{S}''$ is cleaned up with respect to node and edge states resulting in the semantics graph $\mathcal{S}'''$ which in turn represents the accurate state on the semantic level for a subsequent change impact analysis on $\mathcal{T}'$.

$$\begin{tikzcd}
(\mathscr{T}:\mathscr{M},\bullet,\bullet) \arrow[r, "\alpha_{\mathscr{M}}"] \arrow[dd, dashed, "\mathsf{CiA}"'] & (\mathscr{T}:\mathscr{M},\mathcal{S},\bullet) \arrow[d, "\text{patch}(\delta)"] & \\
 & (\mathscr{T}':\mathscr{M},\mathcal{S},\bullet) \arrow[r, "\alpha_{\mathscr{M}}"] & (\mathscr{T}':\mathscr{M},\mathcal{S}',\bullet) \arrow[d, "\sigma_{\mathscr{M}}"] \\
(\mathscr{T}':\mathscr{M},\mathcal{S}''',\mathcal{I}) & & (\mathscr{T}':\mathscr{M},\mathcal{S}'',\bullet) \arrow[ll, "\iota_{\mathscr{M}}"']
\end{tikzcd}$$

Figure 6.11: The Entire Change Impact Analysis on `guests.xml`.

This concludes our change impact analysis example. For an overview, we provide in Fig. 6.11 a compact description of the entire change impact analysis applied on our wedding scenario. ⧫

Employing our semantic difference analysis in combination with the three graph transformations abstraction, propagation, and eventually projection we have shown by an abstract example how our management of change methodology on semi-structured documents can help to identify relevant changes and how to compute the ripple effects, say the impacts. The discussion on how our methodology helps to adjust ripple effects is given in the Chap. 7.

In the following we address the previously mentioned interaction models in order to extend our change impact analysis across document type boundaries.

6.2.3 Interaction Models

An interaction intensionally defines the propagation of semantic information between documents of different types. Thus, an interaction model presupposes a set of document models and extends it by an additional propagation graph transformation between the different semantics graphs of the given document models. To do so, it may require additional types of semantic nodes and edges, which are also specified in an interaction model.

Definition 6.5 (Interaction Model). Let $\mathscr{M}_1,\ldots,\mathscr{M}_n$ be disjoint document models with respective semantic node types $\underline{\mathbb{V}}^i_{\text{sem}}$ and edge types $\underline{\mathbb{E}}^i_{\text{sem}}$, $1 \leq i \leq n$. Furthermore, let $\underline{\mathbb{V}}_{\text{sem}}$ and $\underline{\mathbb{E}}_{\text{sem}}$ be node types and edge types such that $\underline{\mathbb{V}}_{\text{sem}} \cap \underline{\mathbb{V}}^i_{\text{sem}} = \underline{\mathbb{E}}_{\text{sem}} \cap \underline{\mathbb{E}}^i_{\text{sem}} = \emptyset$, $1 \leq i \leq n$ and let $\underline{\mathbb{V}}^*_{\text{sem}} = (\bigcup_{i=1}^n \underline{\mathbb{V}}^i_{\text{sem}}) \cup \underline{\mathbb{V}}_{\text{sem}}$ and $\underline{\mathbb{E}}^*_{\text{sem}} = (\bigcup_{i=1}^n \underline{\mathbb{E}}^i_{\text{sem}}) \cup \underline{\mathbb{E}}_{\text{sem}}$. Then an **interaction model $\mathscr{I}$ between** $\mathscr{M}_1,\ldots,\mathscr{M}_n$ is a tuple $\langle(\underline{\mathbb{V}}_{\text{sem}},\underline{\mathbb{E}}_{\text{sem}}),\sigma\rangle$ where a **propagation** graph transformation $\sigma : \mathbb{G}^{\underline{\mathbb{E}}^*_{\text{sem}}}_{\underline{\mathbb{V}}^*_{\text{sem}}} \to \mathbb{G}^{\underline{\mathbb{E}}^*_{\text{sem}}}_{\underline{\mathbb{V}}^*_{\text{sem}}}$ propagates semantic information in the joint semantics graph of the different document models. Its homomorphic extension $\sigma^\#$ to semantic document impact graphs is only applicable on semantic document impact graphs with empty impact graph and defined by $\sigma^\#(\mathcal{O},\mathcal{S},\bullet) = (\mathcal{O},\mathcal{S}',\bullet)$ where $\mathcal{S}' = (\mathcal{S} \setminus \mathcal{S}_{|\underline{\mathbb{V}}^*_{\text{sem}},\underline{\mathbb{E}}^*_{\text{sem}}}) \cup \sigma(\mathcal{S}_{|\underline{\mathbb{V}}^*_{\text{sem}},\underline{\mathbb{E}}^*_{\text{sem}}})$.

As for the graph transformations of annotation models, we will not distinguish the propagation graph transformations of interaction models and their homomorphic extensions.

6.2.4 Document Models and Interaction Models Combined

We now consider semantic document impact graphs composed from documents of different types. We assume one document model for each document type is given as well as one interaction model that specifies the interaction between these different document types.

Definition 6.6. Let $\mathscr{M}_i = \langle \mathscr{E}_{\mathscr{D}_i}, \mathscr{S}_i, \mathscr{A}_i \rangle$ be document models for the $\mathscr{D}_i$, $1 \leq i \leq n$ and $\mathscr{I}$ be an interaction model for the $\mathscr{M}_i$. An SDI graph $\overleftarrow{\mathcal{D}}$ is **compatible** with the $\mathscr{M}_i, 1 \leq i \leq n$ and the $\mathscr{I}$ if it contains only node types and edge types from the document models and the interaction model.

The compatibility restriction ensures the existence of a document model for each to be managed document type as well as the existence of a appropriated interaction model. Furthermore, it provides an implicit notion of scoping, so that within an SDI graph may exclusively occur documents adequate to the referenced document models.

The combined annotation graph transformation for the whole graph consist of (1) the abstraction graph transformations from each document model, followed by (2) an exhaustive application of the propagation graph transformations from the document models *and* the interaction model, and (3) a final phase where all projection functions from the document models are applied.

Definition 6.7 ($\mathscr{M}$ and $\mathscr{I}$ Combined). Let $\mathscr{M}_i = (\mathscr{E}_{\mathscr{D}_i}, \mathscr{S}_i, (\alpha_{\mathscr{S}_i}, \sigma_{\mathscr{S}_i}, \iota_{\mathscr{S}_i})$ be document models for $\mathscr{D}_i$, $1 \leq i \leq n$ and $\mathscr{I} = \langle (\underline{\mathbb{V}}_{\text{sem}}, \underline{\mathbb{E}}_{\text{sem}}), \sigma_{\mathscr{I}} \rangle$ an interaction model for the $\mathscr{M}_i$ and let $\overleftarrow{\mathcal{D}}$ be a compatible semantic document impact graph. Then the combined graph transformations are defined by:

Abstraction: The combined abstraction of some $\overleftarrow{\mathcal{D}}$ is the application of the combined abstraction $\alpha :=\alpha^{\#}_{\mathscr{S}_n} \circ \ldots \alpha^{\#}_{\mathscr{S}_1}$.

Propagation: The combined propagation on some $\overleftarrow{\mathcal{D}}$ is the exhaustive application of the intermediate combined propagation $\sigma :=\sigma_{\mathscr{I}} \circ \sigma^{\#}_{\mathscr{S}_n} \circ \ldots \sigma^{\#}_{\mathscr{S}_1}$ on $\overleftarrow{\mathcal{D}}$. I.e., we apply σ on $\overleftarrow{\mathcal{D}}$ until we reach a fixpoint which is easily expressed using a fix point combinator *FIX* defined by $(\mathit{FIX}\ F) = F(\mathit{FIX}\ F)$ on $F = \lambda f.\, \lambda g.\, (if\ (g = \sigma(g))\ g\ else\ f(\sigma(g))$.[2]

Projection: The combined projection of some $\overleftarrow{\mathcal{D}}$ is the application of the combined projection $\iota :=\iota^{\#}_{\mathscr{S}_n} \circ \ldots \iota^{\#}_{\mathscr{S}_1}$.

From the disjointness of the document models it follows that the order of the combinations of the abstraction and projections in the definition of the combined abstraction and projections is irrelevant.

Based on the notion of combined document models, we can now precisely define the semantic annotation and change impact analysis for a heterogeneous collection of documents. Assume a set of document models with associated interaction model and a collection of documents $\mathscr{T}_1, \ldots, \mathscr{T}_n$ each belonging to one of the document models. Let further be (α, σ, ι) the respective graph transformations of the combined document and interaction model.

The *semantic annotation of the document collection* consists of applying the three graph transformations on the SDI graph $(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \bullet, \bullet)$ for the given collection of documents with empty semantics graph and empty impacts graph:

$$(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \mathcal{S}, \mathcal{I}_1 \uplus \ldots \uplus \mathcal{I}_n) :=\iota \circ \sigma \circ \alpha(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \bullet, \bullet)$$

Here $\mathcal{I}_i$ contains further annotations for parts of this document, such as, for instance, consistency information, errors, and others.

Analogously, the *change impact analysis of the document collection* consists of applying the patches for each individual document, and followed by the application of the three graph transformations of the combined document and interaction model. Again, in the resulting graph $(\mathscr{T}'_1 \uplus \ldots \uplus \mathscr{T}'_n, \mathcal{S}, \mathcal{I}_1 \uplus \ldots \uplus \mathcal{I}_n)$ where the $\mathscr{T}'_i$ is the new version of document $\mathscr{T}_i$ after patch application. The impacts graph $\mathcal{I}_i$ contains the annotations for parts of this document, such as, for

[2]To wit: $(\mathit{FIX}\ F)\ g \to (F\ (\mathit{FIX}\ F))\ g \to (if\ (g = \sigma(g))\ g\ else\ (\mathit{FIX}\ F)(\sigma(g))$

instance, consistency information, errors, etc. obtained by ripple effects from *all* patches via the combined document and interaction model graph transformations.

6.3 Conclusion

We have presented a framework to model the annotation of semantic properties for heterogeneous collections of documents and to design change impact analysis procedures in a user-friendly declarative style. The key ingredients are (i) changes are determined using a generic semantic tree difference analysis parametrized over document type specific equivalence specifications, (ii) explicit representation of both the syntactic documents and their intentional semantics in a single graph, (iii) view of the semantic annotation process as a specific graph transformation process and its decomposition into the three phases abstraction, propagation and projection which allows one to combine different document types via interaction models.

The framework has been implemented in the *locutor* library exactly following the principles of the framework, based on the graph rewriting tool GrGen.NET. The primary application scenario for *locutor* is to bridge the gap between existing tools supporting document type specific change impact analysis. However, it can also be applied to add change impact analysis support to existing systems. First experiments provide evidence that *locutor* can indeed significantly help to identify and manage effects of changes in an environment of heterogeneous documents. Although an evaluation regarding scalability for large collections of heterogeneous documents is missing, the possibility to parse only semantically relevant parts of the documents into the graph makes us confident that the approach scales.

Acknowledgments. While the specific definitions and the remaining results are ours, the whole chapter benefited strongly from discussions and collaboration with Serge Autexier also published in the joint paper [9].

Chapter 7

Adjustment

We cannot direct the wind, but we can adjust the sails.
— Dolly Parton

"Press F1!". This is one of the popular jokes of computer scientist to get around to answer questions for assistance. This key code is supported on various operating systems, but in most cases there is no adequate answer behind. How about if "Press F1!" not only hides, for example, a "Please contact your system administrator" behind, but a range of concrete solution proposals? In this Chapter we want to deal with the generation of solution proposals with regard to implicit changes on semi-structured documents.

7.1 Introduction

Changes in an enterprise's business processes — recall, inscribed within (collection of) documents — may have significant consequences within all domains of the enterprise, such as the software systems, data management and technical infrastructure. In Chap. 6 we exhaustively studied the identification of those ripple effects that a change may cause.

The goal of a change impact analysis is to see what would happen if a change occurs, before the change really takes place. This information can then be used to help in making a decision on the necessity of a change but also to propose adjustments on the ripple effects to get an entire document collection back to a consistent state. Proposing adjustments is what we want to put behind "Press F1!" in the scope of change management on a collection of semi-structured documents. Therefore we follow an advice by Terry Paulson, the author of Paulson on Change:

It's easiest to ride a horse in the direction it is going.

In other words, one should not struggle against change but learn to use it to ones advantage. In our management of change methodology, we use the advantage of impacts graphs and model adjustments similar to semantic annotations by graph rewriting rules on SDI graphs.

7.2 A Model for Impacts Adjustment Based on Graph Rewriting

There are two types of ripple effects: (1) implicit modifications and (2) conflicts. The former are automatically adjustable modifications. For example, think of a syntactical change, like, a renaming.

$$\begin{tikzcd}
(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \mathcal{S}, \bullet) \arrow[d, "\text{patch}(\delta_1),\ldots,\text{patch}(\delta_n)"] & \\
(\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}, \bullet) \arrow[r, "\text{abstraction}"] \arrow[dd, dashed, "\text{annotate}^+"] & (\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}', \bullet) \arrow[d, "\text{propagation}"] \\
 & (\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}'', \bullet) \arrow[d, "\text{projection}"] \\
(\mathscr{T}_1'' \uplus \ldots \uplus \mathscr{T}_n'', \mathcal{S}'', \mathcal{I}) & (\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}'', \mathcal{I}) \arrow[l, "\text{adjustment}"']
\end{tikzcd}$$

Figure 7.1: Adjustment in Detail

The latter is a conflict caused by semantic modifications generating inconsistency in the dependent documents. Consequently, an implicit modification does not harm common version control workflows, but increases the set of modified items. A conflict, however, will prevent from committing the semantically affected items and request for user assistance.

We handle both cases on the level of graph transformations on semantic document impact graphs affecting the document graph only. In order to distinguish an adjustment graph transformations with respect to how they affect the document graph of an SDI graph, we consider the following additional basic graph transformation (cf. Fig. 7.1):

Adjustment is a graph transformation starting with an SDI graph with non-empty semantics graph and non-empty impacts graph and returns an SDI graph with an updated document graph whereas implicit modifications have been adjusted and conflicts have been marked up.

We do not require highlighting of conflicts within valid documents to result in valid documents with respect to a document type specification. Quite the contrary, the highlighting of conflicts can lead to invalidness of documents, so that conflicts can already be identified on the syntactic level by, for example, validating XML parsers. An alternative would be to extend the syntactic node types and edge types by appropriate ones, such that conflicts can be marked with valid syntactic constructs.

Definition 7.1 (Adjustment Model). Let $\mathscr{A}_{\mathscr{S}} = \langle \alpha_{\mathscr{S}}, \sigma_{\mathscr{S}}, \iota_{\mathscr{S}} \rangle$ be an annotation model, then an **adjustment model $\mathscr{P}$ for $\mathscr{A}$** is a tuple $\langle \mathscr{A}, \psi \rangle$ where

Adjustment: $\psi_{\mathscr{A}} : \wp(\mathbb{FS}^+_{\mathscr{D}} \times \mathbb{G}^{\underline{\mathbb{E}}_{\text{imp}}}_{\underline{\mathbb{V}}_{\text{imp}}}) \times \ddot{\mathbb{G}}^{\underline{\mathbb{E}}_{\text{sem}}}_{\underline{\mathbb{V}}_{\text{sem}}} \to \wp(\mathbb{FS}^+_{\mathscr{D}})$ adjusts the documents with respect to the corresponding impacts. Its homomorphic extension $\psi^{\#}_{\mathscr{A}}$ to semantic document impact graphs is defined by $\psi^{\#}_{\mathscr{A}}(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n \uplus \mathscr{T}, \mathcal{S}, \mathcal{I}_1 \uplus \ldots \uplus \mathcal{I}_n \uplus \mathcal{I}) = (\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n' \uplus \mathscr{T}, \mathcal{S}, \mathcal{I}_1 \uplus \ldots \uplus \mathcal{I}_n \uplus \mathcal{I})$ where $\mathscr{T} \cap \mathbb{FS}^+_{\mathscr{D}} = \emptyset$ and $\{(\mathscr{T}_1', \ldots, \mathscr{T}_n'\} = \psi_{\mathscr{A}}(\{\mathscr{T}_1, \mathcal{I}_1), \ldots (\mathscr{T}_n, \mathcal{I}_n)\}, \mathcal{S})$.

This final extension of our semantic annotation graph transformation entirely completes the kernel theory for our management change methodology whereas the user has the freedom to distinguish between a change impact analysis and an adjustment by choosing either an annotation model or an adjustment model. The *adjustment of a document collection* consists of applying the four graph transformations on the SDI graph $(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \mathcal{S}, \mathcal{I})$ for the given collection of documents:

$$(\mathscr{T}_1' \uplus \ldots \uplus \mathscr{T}_n', \mathcal{S}', \mathcal{I}) \colon = \psi \circ \iota \circ \sigma \circ \alpha(\mathscr{T}_1 \uplus \ldots \uplus \mathscr{T}_n, \mathcal{S}, \mathcal{I})$$

Here $\mathcal{T}_i'$ is the new version of document $\mathcal{T}_i$ after patch application, semantic annotation *and* adjustment.

Note 7.1. Empirical surveys within enterprises, like, the Gordian Consulting GmbH [55] and the microtec consulting GmbH [90], and within institutions, like the Jacobs University Bremen [71], the DFKI Bremen [39], and the Universität des Saarlandes [139], have shown that users do not want fully automatic adjustments but want to manually examine both implicit modifications as well as conflicts. Therefore, according to the realization of an adjustment (cf. Part III), implicit modifications as well as conflicts are wrapped by TEI `change` elements to require explicit user affirmation. This shows once more that the evaluation of the various metadata schemes (cf. Sec. 4.1.2) was worthwhile, even if they were not directly of use for our consolidation slice.

Example 7.1. Now with the definition of a adjustment model let us keep up Ex. 6.1 again but — as previously mentioned — add another document `seatings.xml` of a different document type than `guests.xml`. Within the `seatings.xml` document the seating order of the guests for the celebration is arranged such next to a male always sits one female. This time our scenario is, one male guests cancels the wedding such that the seating arrangement has to be adjusted with respect to the "consistency condition" that a male is always surrounded by females.

Note 7.2. We re-use the previously declared semantic model (cf. Fig. 6.6) as well as the annotation model (cf. Fig. 6.7). The former is extended by `hasMaleNeighbor` and `hasFemaleNeighbor`, respectively. The respective declaration of an interaction model only comprising the edge type `hasSeat` as well as of an adjustment model and adjustment graph transformations, in particular, is available at [99].

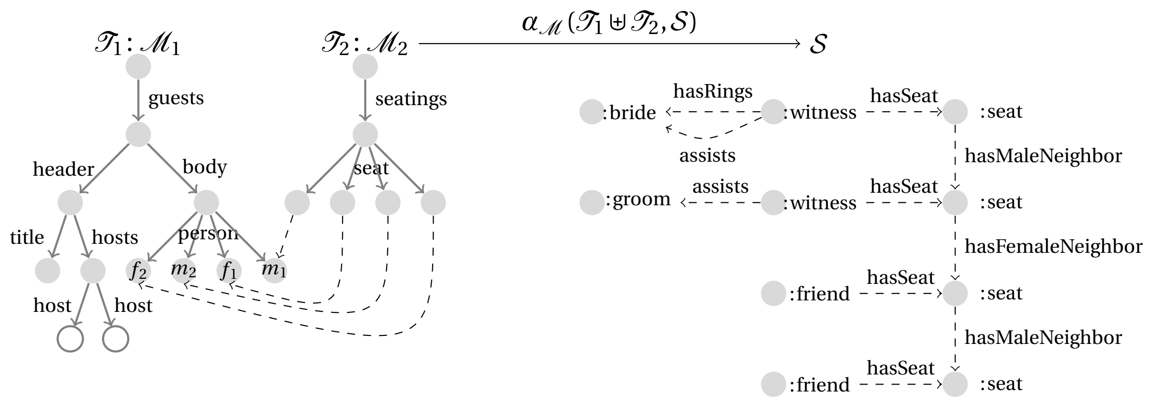


Figure 7.2: A Guest List with a Seating Order

The document collection comprising the guest list and the seating order is depicted on the left side of Fig. 7.2. For illustration purposes we marked up XML `person` elements by their gender whereas f denotes a female person and m denotes a male person. In addition we visualized the links between a `person` element and the respective XML `seat` element. One could imagine the links between `person` and `seat` elements represented by XML `for` attributes within the `seatings.xml` document.

At the beginning of the wedding plan the seating arrangement is consistent and the respective abstraction on $\mathcal{T}_1 \uplus \mathcal{T}_2$ results in the semantics graph $\mathcal{S}$ depicted on the right side of Fig. 7.2. The semantics graph $\mathcal{S}$ represents the fact that the witnesses assist the bride and the groom, respectively, and one of the witnesses holds the wedding rings. In addition utilizing an interaction model

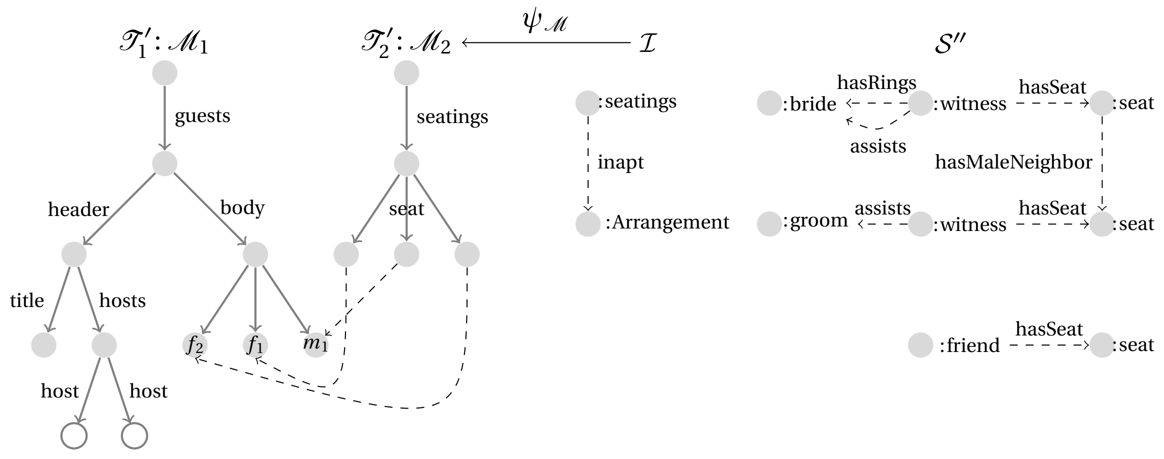


Figure 7.3: The Projection of Changes $\psi_{\mathcal{M}}(\{(\mathcal{T}_1,\bullet),(\mathcal{T}_2,\mathcal{I}),\mathcal{S}''\})$.

the witnesses as well as the friends are associated with their seats and each seat is associated with the seat next to the right denoting its female and male neighbor, respectively.

Now the situation is that one person, a male friend, cancels to participate at the wedding celebration. A subsequent abstraction on the modified document collection $\mathcal{T}_1' \uplus \mathcal{T}_2$ followed by an change propagation results in a semantics graph $\mathcal{S}'$ with semantic nodes and edges marked with respect to their status. The projection of $\mathcal{S}'$ to the impacts graph $\mathcal{I}$ is depicted on the right side on Fig. 7.3 and $\mathcal{I}$ denotes the fact that the seating order is inaptly arranged. The semantics graph $\mathcal{S}'$ has been cleaned up with respect to deleted nodes and edges. The resulting semantics graph $\mathcal{S}''$ is depicted on the very right of Fig. 7.3.

Now we apply an adjustment graph transformation on the document graph $\mathcal{T}_1' \uplus \mathcal{T}_2$ in combination with the impacts graph $\mathcal{I}$ associated to $\mathcal{T}_2$ and the overall semantics graph $\mathcal{S}''$. The resulting adjusted documents $\mathcal{T}_1' \uplus \mathcal{T}_2'$ are depicted on the left side of Fig. 7.3. The adjustment re-established the consistency criteria by rearranging the seating order. Now a male person is again surrounded by female persons and the wedding celebration can start. ◆

7.3 Conclusion

We have presented an extension to the change impact analysis framework through to model adjustments for heterogeneous collections of documents. The key extension is the view of adjustments to be just one more specific graph transformation process preceded by the previously introduced semantic annotation process which allows one to ask for assistance incorporating/ fixing changes due to ripple effects.

In the subsequent parts we will respond to the realization of our management of change methodology in order to ultimately conclude this work with a summary of results and future work items.

Part III

Realization

Chapter 8

The *locutor* System

Resistance is futile!

— The Borg

The management of change methodology presented in this work has been realized in the prototype tool *locutor* that analyses, annotates and adjusts the change impacts on collections of semi-structured documents. The tool is parameterized over document models and interaction models specified in an extension to the declarative GrGen.NET language.

8.1 The *locutor* System Architecture

To begin with, we explain where the name "locutor" has its origin. Star Trek is an American science fiction entertainment series and media franchise. Therein the Borg are a fictional pseudo-race of cyborgs. Exhibiting a rapid adaptability to any situation or threat, with encounters characterized by matter of fact "resistance is futile"-type imperatives, the Borg develop into one of the greatest threats to the Federation. In one of the episodes they capture and assimilate Jean-Luc Picard into the collective by surgically altering him, creating Locutus of Borg [16]. This being said, the author, as a Star Trek fan (*aka.*. Treky), chose the name "locutor" for his management of change system aiming (1) to express rapid adaptability support concerning any *change* on documents and (2) to emphasize gapless alignment within and between the collective (*here:* the collection of to be managed documents) by *propagating* compelling changes along assimilated document *dependencies*.

Below, we first explain the overall *locutor* system architecture and then discuss the individual components in more detail. The *locutor* system[1] illustrated in Fig. 8.1 is a Java/Scala management of change system on semi-structured documents based on the version control system Subversion utilizing the SVNKit [131] library. SVNKit is a pure Java client library for working with data versioned by the Subversion version control system right within Java applications. The library is structured into two main layers [133]: (1) a high-level layer to manage working copies and a corresponding API gives one the ability to manage working copies just as the Subversion command line client does and (2) a low-level layer similar to the Subversion repository access layer representing a driver for direct working with a Subversion repository. The high-level API is similar to the commands of the Subversion native command line client. All operations for managing working copies are logically divided and combined in different `SVN*Client` classes. For example,

[1]The system architecture diagram is a modified version of SVNKit's system architecture [131].

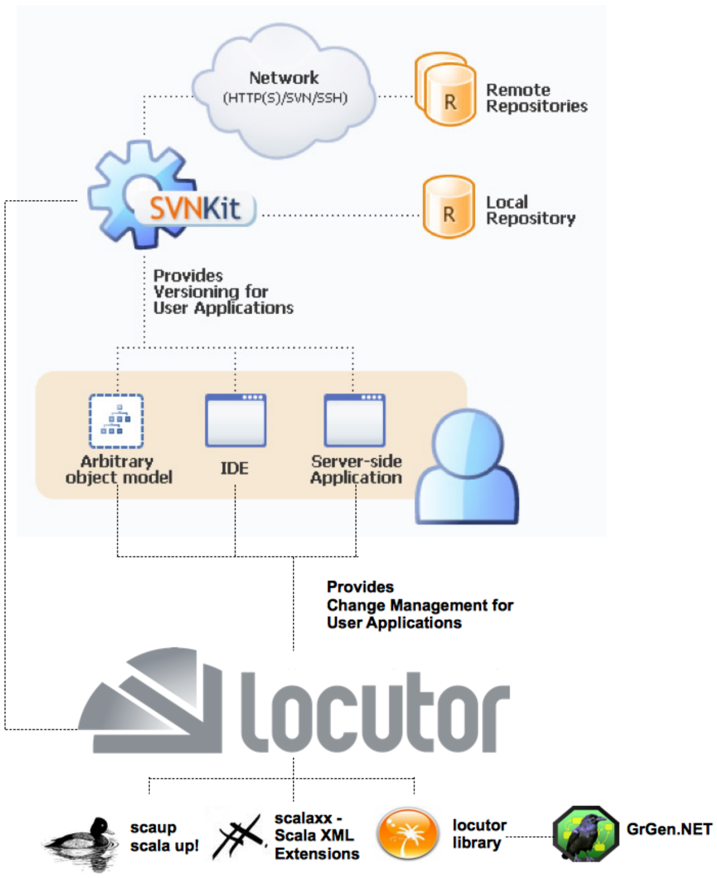


Figure 8.1: The *locutor* System Architecture.

all working copy update operations (check out, update, switch, etc.) are performed by the single `SVNUpdateClient` class. Arguments which are accepted by methods of these classes are similar to arguments of the SUBVERSION command line client. When an access to a repository is indeed required, the high-level layer uses the low-level one. This layer represents an abstract SUBVERSION repository access protocol layer. There are two important things about this layer: (1) it does not deal with working copies at all, since working copies are on a higher layer than this one. This layer knows how to speak to SUBVERSION repositories via different protocols (in fact, this layer implements such protocols) and (2) data structure is handled by this layer as an abstract data hierarchy, what gives an ability to represent more sophisticated (than just files and directories) abstractions as a tree-like hierarchy and keep such a hierarchy under version control. A comprehensive documentation of the SVNKIT API is available at [132].

The *locutor* system employs the high-level layer of the SVNKIT library only. All version control functionalities are passed from the *locutor* command line client [100] to the SVNKIT high-level API. A semantic difference analysis is handled by the SCAUP library [103] and the SCALAXX library [103], respectively. Both are SCALA libraries whereas the former one is a general utility library for the SCALA language, as a companion to the standard library. The latter is an extension to SCALA's library support for XML processing. A change impact analysis as well as an adjustment is handled by the *locutor* core library [99] also implemented in the SCALA language. The actual graph rewriting is handled by GRGEN.NET [53] accessed via the *locutor* core library.

8.2 The *locutor* Command Line Client

The *locutor* command line client being an extension to the SVNKit command line client is a full, but improved, substitution of the native Subversion client through to registry support, redundancy resolution, change impact analysis and adjustment. The registry support as well as the redundancy resolution if fully implemented and has been successfully used for more than two years at the Jacobs University Bremen. Since most researchers in this area prefer to use a Unix-like environments, the elaboration is tailored to these environments. Nonetheless, the basic principles can be easily applied to a Windows environment too. Users familiar with the native Subversion client learn the basics in just a few minutes and, due to its low overhead, can apply management of change to the smallest of projects with ease. Its simplicity means you will not have a lot of abstruse concepts or command sequences competing for mental space with whatever you are really trying to do. At the same time, due to SVNKit's high performance, *locutor* let you scale painlessly to handle large projects. At the current state of this work the change impact analysis as well as the adjustment has not been fully integrated into the *locutor* command line client but is already fully implemented within in the underlying *locutor* core library.

8.2.1 Commands

The basic usage of the *locutor* command line client is

```
locutor␣<sub-command>␣[options]␣[args]
```

Most sub-commands take file and/or directory arguments, recursing on the directories. If no arguments are supplied to such a command, it recurses on the current directory (inclusive) by default. There was some discussion to set the default behavior of the *locutor* command line client regarding recursing on directories opposite to the native Subversion command line client, i.e. recursion should be disabled by default. As we are arguing the *locutor* command line client behaves just like the native Subversion command line client the author took the decision to stick to defaults behavior, i.e. recursion is enabled by default and, if applicable, may be disabled by the `-N` option. However, there is one minor distinction to the native Subversion command line client: in the *locutor* command line client there is no distinction made between directory paths with a tailing path-separator or none, i.e., for example, `/home/nmueller/locutor/` and `/home/nmueller/locutor` are considered to be equal with respect to the provided sub-commands.

The *locutor* command line client covers *all* standard Subversion sub-commands without any distinction to the native behavior. For completeness, these are the supported standard Subversion sub-commands: (1) add, (2) blame, (3) cat, (4) changelist, (5) checkout, (6) cleanup, (7) commit, (8) copy, (9) delete, (10) diff, (11) export, (12) help, (13) import, (14) info, (15) list, (16) lock, (17) log, (18) merge, (19) mergeinfo, (20) mkdir, (21) move, (22) propdel, (23) propedit, (24) propget, (25) proplist, (26) propset, (27) resolve, (28) resolved, (29) revert, (30) status, (31) switch, (32) unlock, and (33) update. A comprehensive documentation on these commands is available at [28].

The *locutor* command line client adds to the standard commands the following specific ones:

Command	Description
`regadd`	Register a working copy. In the first step a `regadd` sub-command traverses the directory tree up from the current path, searching for the working copy root $\Uparrow\mathcal{H}$ with $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ to be registered. In the second step a `regadd` sub-command traverses the directory tree down starting at $\Uparrow\mathcal{H}$ identifying $\mathbb{X}_{\mathcal{H}}$ such that $(w, \mathbb{X}_{\mathcal{H}}) \in$ Graph($\mathbb{REG}$). A `regadd` sub-command within an external e of a working copy w stops at root of $w\|_e$.
`regdel`	Unregister a working copy. A `regdel` sub-command traverses the directory tree up from the current path, searching for the working copy root $\Uparrow\mathcal{H}$ with $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ to be unregistered. The respective directory tree is reset with respect to transformed externals. Depending transformed externals are reset as well. A `regdel` sub-command within an external $e \in \mathbb{X}_{\mathcal{H}}$ unregisters e as well as $w\|_e$.
`regsync`	Synchronizes the registry. All registry entries are synchronized with the respective file system entry. Working copies removed from the file system get automatically unregistered via a `regdel` sub-command. Modifications on working copies with respect to externals definitions are incorporated. If an external $e = \langle \pi, \Theta \rangle \in \mathbb{X}_{\mathcal{H}}$ with $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ is registered twice, i.e. $(w, \mathbb{X}_{\mathcal{H}}) \in \mathbb{REG}$ *and* $(w\|_e, \mathbb{X}_{\mathcal{H}\|_\pi}) \in \mathbb{REG}$, then $w\|_e$ is unregistered.
`sanitize`	Performs a redundancy resolution on a registered working copy. A `sanitize` sub-command traverses the directory tree up from the current path, searching for the working copy root $\Uparrow\mathcal{H}$ with $w = \langle \mathcal{H}, \omega, \mathbb{A} \rangle$ to be sanitized[2]. In the second step a `sanitize` sub-command resolves all externals $e \in \mathbb{X}_{\mathcal{H}}$ for which an redundant registry entry exists. If a working copy m and an external e' are potential candidates for a redundancy resolution, then $e \gg m$. Transitive symbolic links are automatically resolved, i.e. if $n \gg m \gg w$ then $n \gg w$ *and* $m \gg w$. Externals definitions mapping a local directory to a parent path of the owner are transformed as well. Note that SUBVERSION in this case would loop infinite. If an external contains local modifications, transformation is aborted. Recursion on transformed externals is not yet supported. That is, a `sanitize` sub-command is local to a working copy.
`update-all`	Update all registered working copies. An `update-all` sub-command brings changes from the respective repositories into all registered working copies.

Continued on next page

[2]The sub-command “resolve” is already reserved by SUBVERSION.

***locutor* Commands – continued from previous page**

Command	Description
`clean-all`	Clean up all registered working copies. A `clean-all` sub-command recursively cleans up all registered working copies, removing locks, resuming unfinished operations, etc.

The herein described *locutor* sub-commands describe the latest system state. Via the registration process the user obtains full freedom on activation of the enhanced features of the *locutor* system. That is, working copies not registered are not affected by any of the enhanced *locutor* system functionalities. Another approach would have been to modify the standard SUBVERSION sub-commands. For example, a checkout could entail an automated registration processes. We decided, however, to leave the decision to the individual user aware of the fact that omitting these system guidance but gaining more authority entails an increase of watchfulness. For example, I/O removing of a registered working copy without synchronizing the registry may cause inconsistent depending registered resources. But of course it is almost trivial to define augmented sub-commands such as `regexp` as `regdel;export`.

The following table gives an overview of the suggested change impact analysis sub-command and adjustment sub-command, respectively, to be implemented in the next system release:

Command	Description
`upcia`	Update a working copy followed by a change impact analysis. An `upcia` sub-command employs the standard SUBVERSION `update` command followed by a change impact analysis (`update >> cia`). The SDI graph is inferred by all the documents located in the directory tree at the current path whereas the respective semantic models and annotation models must be located at *locutor*'s command line client home directory. Optionally (`withAdjustment`) a subsequent adjustment is initiated (`update >> cia >> adjust >> interact`). Following the example of SCALA's parser combinators, the >> operator refers to piping information, like, the set of updated version controlled items, to the next operation.
`ciaci`	Perform a change impact analysis followed by a commit. A `ciaci` sub-command employs the standard SUBVERSION `commit` command preceded by a change impact analysis (`cia >> commit`). Optionally (`withAdjustment`) a subsequent adjustment is initiated (`cia >> adjust >> interact >> commit`).

Due to the fact, that the *locutor* core library already provides the `cia` and `adjust` operations and the SVNKIT library provides the `interact` operation, we are confident that the implementation of `upcia` and `ciaci` are straightforward. The main feature of both former sub-commands is the fact that an SUBVERSION changeset [28] is implicitly extended. In contrast, a native SUBVERSION `update` sub-command brings only local changes from the repository into the working copy, i.e. without ripple effects. A preceding or subsequent change impact analysis, however, ripples the effects through the collection of documents marking or adjusting implicitly affected fragments. This is essential in a working environment where some users utilize the *locutor* command line client and some users utilize the native SUBVERSION command line client. In case of a homogenous working environment, i.e. users are exclusively using the *locutor* command line client, a `ciaci` sub-command initiated by one of the users entails a simple native SUBVERSION `update` command for all the other users modulo their local changes. Thus a semantic annotation and a adjustment, respectively, only has to be performed once with respect to the local changes at the side of the user initiating the commit.

8.2.2 The *locutor* Registry

The *locutor* registry is represented as an XML instance of the document type definition illustrated in Fig. 8.2. The document element `registry` comprises zero or more working copy entries `wc`. For

```
<!ELEMENT registry ( wc* ) > <!ATTLIST registry version NMTOKEN
   #REQUIRED >

<!ELEMENT wc ( external* ) >
<!ATTLIST wc root CDATA #REQUIRED >
<!ATTLIST wc url CDATA #REQUIRED >
<!ATTLIST wc xml:id CDATA #REQUIRED >

<!ELEMENT external EMPTY >
<!ATTLIST external own CDATA #REQUIRED >
<!ATTLIST external rev NMTOKEN #REQUIRED >
<!ATTLIST external tgt NMTOKEN #REQUIRED >
<!ATTLIST external url CDATA #REQUIRED >
<!ATTLIST external xml:id CDATA #REQUIRED >
<!ATTLIST external xref CDATA #IMPLIED >
```

Figure 8.2: The DTD of a *locutor* Registry

compatibility aspects a `registry` element holds a `version` attribute stating the current version of the *locutor* command line client. A `wc` element comprises zero or more externals `external`. The working copy root is stored in the `root` attribute, the repository URL in the `url` attribute, and the `xml:id` attribute holds the MD5 encoding of the `root` attribute for unique identification. An `external` element is an XML empty element comprising the following attributes: (1) the `own` attribute represents the path of the owner the external is defined on, (2) the `rev` attribute represents the external's revision, (3) the `tgt` attribute represents the target path of the externals definition, (4) the `url` attribute represents the URL of the externals definition, and (5) the `xml:id` attribute holds the MD5 encoding of the owner path combined with the target. The `xref` attribute is for transformed externals pointing to the `xml:id` of the registry entry the external has been resolved to. The actual symbolic file system link is computed by subtracting the repository URL of the `xref` target from the repository URL of the transformed external and appending the result to the owner path in combination with the target path of the `xref` target.

For example let us consider the simple registry illustrated in Fig. 8.3. The registry comprises one registered working copy at `/Users/nmueller/locutor` from the repository at

```
<registry version="0.1" xmlns="http://code.google.com/p/locutor">
  <wc xml:id="md5(/Users/nmueller/locutor)"
      url="https://locutor.googlecode.com/svn/trunk"
      root="/Users/nmueller/locutor">
    <external xml:id="md5(/Users/nmueller/locutor/prj/plugin)"
              own="/Users/nmueller/locutor/prj"
              rev="-1"
              url="https://kwarc.info/locutor/src/plugin"
              tgt="plugin"/>
    <external xml:id="md5(/Users/nmueller/locutor/doc/proposal)"
              own="/Users/nmueller/locutor/doc"
              rev="-1"
              url="https://kwarc.info/locutor/src/plugin/doc/proposal"
              tgt="proposal"
              xref="md5ref(/Users/nmueller/locutor/prj/plugin)"/>
  </wc>
</registry>
```

Figure 8.3: A *locutor* Registry.

https://locutor.googlecode.com/svn/trunk. In turn, the working copy comprises two externals. One defined on /Users/nmueller/locutor/prj and one defined on /Users/nmueller/locutor/doc. The former externals definition is bound to the repository at https://kwarc.info/locutor/src/plugin putting the data into plugin sub-directory of the respective owner path. The latter externals definition is bound to the repository at https://kwarc.info/locutor/src/plugin/doc/proposal putting the data into proposal sub-directory of its owner path. Both externals definition are sticked to the HEAD revision of the respective repository. The second external, however, is redundant to the first one, such that the external with xml:id="md5(/Users/nmueller/locutor/doc/proposal)"[3] is symlinked to /Users/nmueller/locutor/prj/plugin/doc/proposal.

8.3 The Semantic Differ

The semantic differ *sdiff* is implemented within the SCALAXX library utilizing equivalence systems implemented within the SCAUP library. Both the *sdiff* algorithm and an equivalence system has already been illustrated in Chap. 5. The presented code there is complete but marginally beau-

```
trait EquivalenceSystem[M <: {def get(key: String): Option[Seq[T]]}, −T]

class Differ[M <: MetaData, T >: Node](eqsys: scaup.eq.EquivalenceSystem[M, T])
```

Figure 8.4: The Equivalence System & Differ Interface.

tified. Thus, here we only want to address the concrete signature of both items with respect to their type parameters. In Fig. 8.4 the first line represents the interface to an equivalence system.

[3]The md5/md5ref functions are *locutor* internal functions for MD5 string encoding. For convenience we used the function calls rather than the resulting MD5 encoding.

An equivalence system is a trait. Traits are a fundamental unit of code reuse in SCALA [110]. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits. However, an equivalence system is not a type because it takes type parameters. The first one `M` is restricted to be a subtype of some type with a `get` method defined on. `M` is a structural type: `M` does not denote a class but all objects whose structure include a method called "get" taking a `String` parameter and returning a `String`. The second type parameter `T` has no further typing restrictions but is variance annotated. We also call an equivalence system a type constructor, because with it one can construct a type by specifying a type parameter. The type constructor `EquivalenceSystem` "generates" a family of types. One can also say `EquivalenceSystem` is a generic trait.

The combination of type parameters and subtyping poses some interesting questions. For example, are there any special subtyping relationships between members of the family of types generated by `EquivalenceSystem[M,T]`? Or more generally, if `S` is a subtype of type `T`, then should `EquivalenceSystem[M,S]` be considered a subtype of `EquivalenceSystem[M,T]`? If so, one could say that trait `EquivalenceSystem` is *covariant* (or "flexible") in its type parameter `T`. This would mean, for example, that one could pass a `EquivalenceSystem[_,String]` to a method which takes a value parameter of type `EquivalenceSystem[_, Any]`. Intuitively, all this seems OK, since a equivalence system of `String`s looks like a special case of an equivalence system of `Any`. In SCALA, however, generic types have by default nonvariant (or, "rigid") subtyping. However, one can demand covariant subtyping by prefixing the formal type parameter with a `+`. By adding this single character, one is telling SCALA that `EquivalenceSystem[_,String]`, for example, to be considered a subtype of `EquivalenceSystem[_,Any]`. The compiler will check that this subtyping is sound.

Besides `+`, there is also a prefix `-`, which indicates *contravariant* subtyping. Then if `T` is a subtype of type `S`, this would imply that `EquivalenceSystem[_,S]` is a subtype of `EquivalenceSystem[_,T]`. Whether a type parameter is covariant, contravariant, or nonvariant is called the parameter's variance. The + and − symbols you can place next to type parameters are called variance annotations.

We utilized contravariant annotation in an equivalence system to model the following fact: if `S <: T` then `EquivalenceSystem[_,T] <: EquivalenceSystem[_,S]` with respect to the amount of equivalent elements. For example, `XML <: Any` holds but within an equivalence system for `XML` more elements are considers to be equivalent than within an equivalence system for `Any`.

This reasoning points to a general principle in type system design: it is safe to assume that a type `T` is a subtype of a type `U` if you can substitute a value of type `T` wherever a value of type `U` is required. This is called the *Liskov Substitution Principle*. The principle holds if `T` supports the same operations as `U` and all of `T`'s operations require less and provide more than the corresponding operations in `U`. A prominent example taken from SCALA is its function traits. Whenever one write the function type `A => B`, SCALA utilizes its in-build `trait Function1[-S, +T]`. The definition of `Function1` in the standard library uses both covariance and contravariance: the `Function1` trait is contravariant in the function argument type `S` and covariant in the result type `T`. This satisfies the Liskov substitution principle, because arguments are something that is required, whereas results are something that is provided.

All methods within `EquivalenceSystem` have to respect this substitution principle. Unfortunately these methods have `T` as a type of their return value. Fortunately, there SCALA has a way to get unstuck: one can generalize these methods by making them polymorphic (i.e., giving the

```
def findEquivTo[S <: T](elem: S, ctx: Seq[S]): Slit [S]
```

Figure 8.5: Polymorphic Equivalence System Method.

append method itself a type parameter) and using a lower bound for its type parameter. For example, the definition for the `findEquivTo` method (cf. Fig. 8.5) gives a type parameter `S`, and with the syntax, `S <: T`, defines `T` as the upper bound for `S`. As a result, `S` is required to be a subtype of `T`. The return value of `findEquivTo` is now of type `Slit[S]` instead of type `Slit[T]`. This definition of `findEquivTo` is arguably better than the old one, because it is more general.

The `Differ` interface to the *sdiff* algorithm respects these variance annotations and subtyping restrictions by lower bounding its type parameter `T` to the type `Node` such that the resulting `Slits` after an equivalence identification are a subtype of `Slit[Node]`. As the *sdiff* algorithm operates on semi-structured documents and `Node` is the default type for representing semi-structured documents, this is exactly what we want from an equivalence identification: the slit in a sequence of semi-structured document fragments identifying the equivalent one.

8.4 The *locutor* Core Library

The main feature of the *locutor* core library is the functionality of a change impact analysis as well as of an adjustment. In the first versions of the library both functionalities were implemented by the author himself. In other words, the library had its own graph rewriting component. Thanks Serge Autexier, who works on the DocTip [5] project at the German Research Center for Artificial Intelligence (DFKI GmbH) implementing a similar but more specialized library `gmoc` [6], the author has been advised of GrGen.NET. Because of the long testing and reliability of GrGen.NET, the author then decided to throw his own development on graph rewriting overboard and adapt his functionality to GrGen.NET.

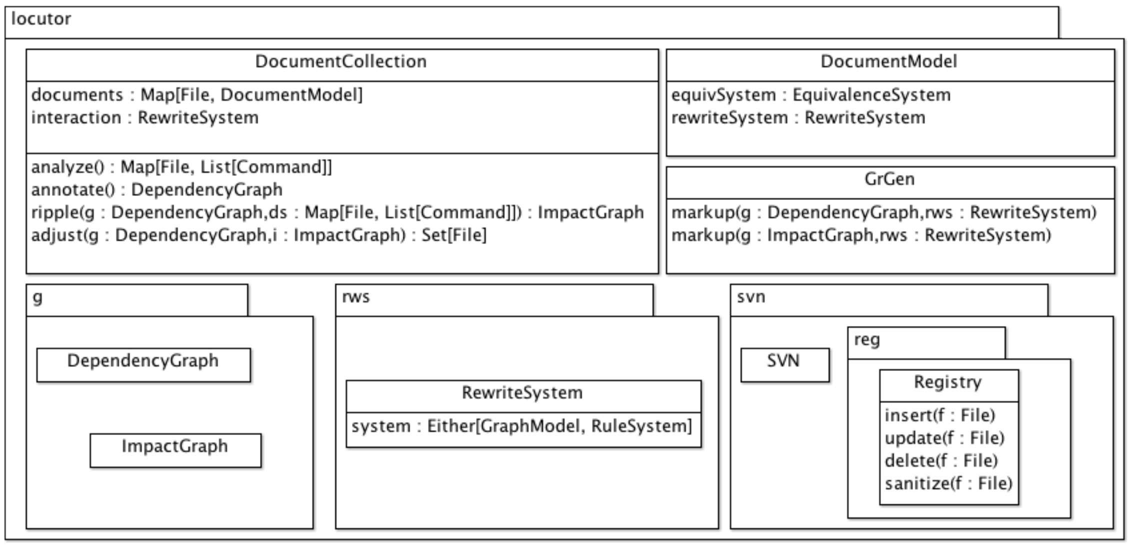

locutor
DocumentCollection
documents : Map[File, DocumentModel]
interaction : RewriteSystem
analyze() : Map[File, List[Command]]
annotate() : DependencyGraph
ripple(g : DependencyGraph,ds : Map[File, List[Command]]) : ImpactGraph
adjust(g : DependencyGraph,i : ImpactGraph) : Set[File]
DocumentModel
equivSystem : EquivalenceSystem
rewriteSystem : RewriteSystem
GrGen
markup(g : DependencyGraph,rws : RewriteSystem)
markup(g : ImpactGraph,rws : RewriteSystem)
g
DependencyGraph
ImpactGraph
rws
RewriteSystem
system : Either[GraphModel, RuleSystem]
svn
SVN
reg
Registry
insert(f : File)
update(f : File)
delete(f : File)
sanitize(f : File)

Figure 8.6: The *locutor* Core Library.

Figure 8.6 illustrates the essential package structure of the *locutor* core library. Within the svn package the library provides some "SUBVERSION boost" functionalities much faster than the one provided by SVNKIT. For example, the identification of a working copy root is tremendously faster then the implementation within SVNKIT.

The sub-package reg holds the Registry class. This class implements the entire previously mentioned registry functionality. Via a simple insert, update, delete, and sanitize commands one can control a *locutor* registry.

The at the beginning of this section mentioned interface to GRGEN.NET is implemented in the GrGen class. Respective GRGEN.NET graph models in combination with GRGEN.NET rules systems are represented by the RewriteSystem class within the rws package. As a rewrite system might exists of either a graph model or a rule system refering to one or more graph models, the system slot of the RewriteSystem class is of type GraphModel + RuleSystem. The here not listed parser package comprises the respective GRGEN.NET parsers for the GRGEN.NET graph model language, the GRGEN.NET rule system language, and GRGEN.NET script language. These parsers are capable to fully parse the latest GRGEN.NET syntax. However, we extended the GRGEN.NET graph model language as well as the GRGEN.NET rule system language by a module system. Figure 8.7 illustrates an instance *locutor* rule system. The left side of Fig. 8.7 illustrates a

```
  graphmodel MoC {
  enum Effect {none, local, implicit}
  abstract node class Infom {
    url    : string;
    path   : string;
    effect : Effect = Effect :: none;
    ct     : set<string>;
  }
  edge class dependsOn connect Infom[+] → Infom[+] {
    via: set<string>;
  }
  edge class affectedBy connect Infom → Infom[+];
  edge class childOf    connect Infom → Infom;
}
graphmodel Wedding extends "MoC" {
  node class Parents extends Infom;
  edge class hasParents connect Infom   → parents;
  edge class isFather   connect parents → Infom;
  edge class isMother   connect parents → Infom;
  edge class friendOf   connect Infom[+] → Infom[+];
}
```

```
rulesystem MoC with "MoC" {
  // rules
  rule slurp {
    x: Infom;
    negative {
      if {x. effect==Effect::local || x. effect==Effect::implicit;}
    }
    replace {}
  }
  // patterns
  pattern findDependant(provider:Infom, dependant: Infom) {
    alternative {
      base { dependant−:dependsOn−>provider; }
      rec  {
        negative { −:dependsOn−>provider; }
        provider−:childOf−>next:Infom;
        :findDependant(next,dependant); } } }
}
rulesystem Wedding with @COLLECTION extends "MoC" {
  rule σ ...
}
```

Figure 8.7: A *locutor* Rewrite System.

locutor graph model. A *locutor* graph model is declared by the graphmodel keyword. Such graph models are identified via a name. A name is any alphanumeric string not beginning with a digit, but possibly including underscores, a number, or any quoted string possibly containing escaped quotes. A graph model can inherit from another graph model. The extends clause refering to the super graph model via name defines inherited members of the graph model whereas the body defines overriding or new members. In case of element capturing the behavior is undefined. The syntax of a graph model body has not been changed with respect to the GRGEN.NET graph model language. Passing *locutor* graph models to GRGEN.NET precedes an automatic flatting.

The right site of Fig. 8.7 illustrates a *locutor* rule system. A *locutor* rule system is declared by the rulesystem keyword. Such rule systems are identified via a name. A name is any alphanumeric string not beginning with a digit, but possibly including underscores, a number, or any quoted string possibly containing escaped quotes. A rule system can inherit from another rule system. The `extends` clause refering to the super rule system via name defines inherited members of the rule system whereas the body defines overriding or new members. In case of element capturing the behavior is undefined. The `with` clause refers to the utilized *locutor* graph model via name. The special `@COLLECTION` keyword refers implicitly to the current document collection. This is particularly useful for interaction models. The syntax of a rule system body has not been changed with respect to the GrGen.NET rule system language. Passing *locutor* rule systems to GrGen.NET precedes an automatic flatting.

In both cases, i.e. for *locutor* graph models as well as for *locutor* rule systems, only singe inheritance is support in the current *locutor* system.

The sub-graphs of a SDI graph are implemented within the graph package (`g`). The `DependencyGraph` class represents both a document sub-graph and a semantics sub-graph. Respective filter methods enable to extract either the document sub-graph or the semantics sub-graph. An impacts graph is represented by the `ImpactGraph` class. Respective filter methods enable to define node type and/or dependency type dependent views on a impacts graph. All internal graph representation have a transformation method to the GrGen.NET script language.

The `DocumentModel` class represents a document model comprising an equivalence system and a rewrite system which in turn represents both a semantic model as well as an annotation model.

The main interface to the *locutor* core library is represented by `DocumentCollection` class. The documents within the collection are stored to the `documents` slot including the associated document models. The interaction model for the collection of documents is stored in the `interaction` slot. The rewrite system of a document collection is automatically computed during run-time, i.e. the rewrite systems of each document are joined with the interaction model. The `analyze` method invokes a semantic difference analysis on the collection of documents resulting in a function mapping each file to its respective edit script. A semantic annotation is initiated via an `annotate` call. The resulting dependency graph comprises the document sub-graph as well as the semantics sub-graph of the SDI graph for the current document collection. Passing such a dependency graph together with the previously computed edit scripts to the `ripple` method completes a change impact analysis on the document collection. The resulting impacts graph can then be passed to the `adjust` method in order to perform an adjustment on the collection of documents. Implicit modifications as well as conflicts are wrapped following the example of TEI `change` elements with respect to Note 7.1.

```
  val C = DocumentCollection(
  "wedding",
  Some("wedding/wedding.rs"),
  ("etc/data/wedding/guests.xml", DocumentModel("wedding/guests.eq", "wedding/guests.rs")),
  ("etc/data/wedding/seating.xml", DocumentModel("wedding/seating.eq"))
)
val Δ = C.analyze               // Semantic difference analysis
implicit val D = C.annotate     // Semantic annotation
val I = C ↩ Δ                   // Change impact analysis
```

Figure 8.8: A Change Impact Analysis Powered by *locutor*.

Figure 8.8 illustrates a change impact analysis utilizing the *locutor* core library. First the document collection is defined. The first parameter gives the collection name. The second one is an optional interaction model specification. All parsing of specifications is handled internally by the library. The last parameter is an ellipse specifying the set of documents with associated document models. The first parameter of a document model is an equivalence specification and optionally a rewrite system. As mentioned above, here one can pass either a *locutor* rule system or a *locutor* graph model. The thereby defined collection of documents is then analyzed with respect to semantic differences resulting in a set of edit scripts associated to the respective documents. A subsequent annotation performs an abstraction on the collection of documents. The final rippling ($\hookleftarrow$) performs a propagation and projection graph transformation utilizing the implicitly given dependency graph. The resulting impacts graph represents the ripple effects on the collection.

To visualize a document collection *locutor* employs the GrGen.NET graph visualization tool `ycomp` [54]. Figure 8.9 illustrates a section of the document graphs of the document collection de-

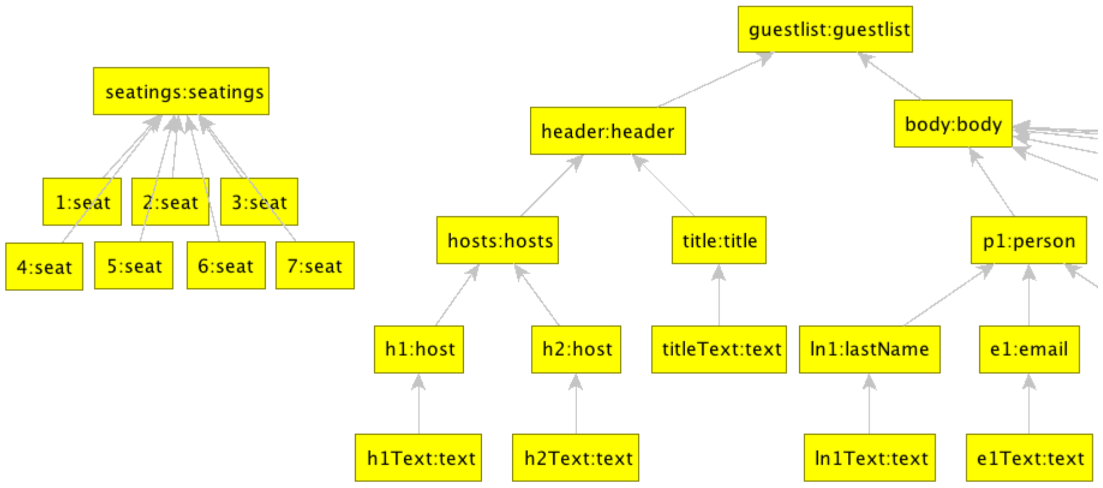


Figure 8.9: The Wedding Document Collection.

fined in Fig. 8.8: to the left the document sub-graph of the `seatings.xml` document is depicted and to the right an excerpt of the document graph of the `guests.xml` document.

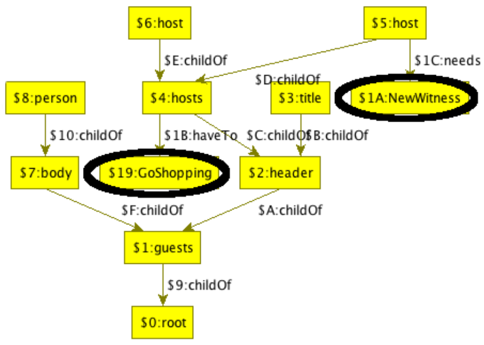


Figure 8.10: Marked Impacts on `guests.xml`.

Figure 8.10 illustrates the result of the change impact analysis performed in Fig. 8.8[4]. The `host` element, representing the bride, needs a new witness and the `hosts` element, representing the bridal couple, has to go shopping for new wedding rings.

[4]Graph transformations are dynamically visualized within `ycomp`, i.e. user can visually inspect how a graph gets transformed by each single graph transformation. A very nice feature of `ycomp`.

Chapter 9

The Translucent Box

The *locutor* command line client has been exhaustively tested by the KWARC group. They employ for their day-to-day version control business the *locutor* command line client as a substitution to the native SUBVERSION command line client. Due to the huge amount of interrelated working copies within the KWARC group and the resulting continuously initiated redundancy resolutions, the author is confident that the *locutor* command line client is actually usable and has its right to exists.

The SCAUP/SCALAXX library and the therein implemented *sdiff* algorithm, in particular, is highly used by the DocTip project [5], the `gmoc` library [6], the TNTBase database [162], and the SCALA community in general. Again this makes the author confident that both libraries are robust and reliable. Although we have neither yet performed a complexity analysis nor a termination analysis for *sdiff*, the feedback from the main developers of each project signals that *sdiff* is fast, correct and compact in sense of the produced edit scripts.

The change impact analysis and the adjustment functionality of the *locutor* core library, however, has only been black & white box tested, yet. Black-box and white-box are test design methods. Black-box test design treats the system as a "black-box", so it does not explicitly use knowledge of the internal structure. Black-box test design is usually described as focusing on testing functional requirements. Synonyms for black-box include: behavioral, functional, opaque-box, and closed-box. White-box test design allows one to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data. Synonyms for white-box include: structural, glass-box and clear-box.

While black-box and white-box are terms that are still in popular use, many people prefer the terms "behavioral" and "structural". Behavioral test design is slightly different from black-box test design because the use of internal knowledge is not strictly forbidden, but it is still discouraged. In practice, it has not proven useful to use a single test design method. One has to use a mixture of different methods so that they are not hindered by the limitations of a particular one. We call this "gray-box" or "translucent-box" test design. Our translucent-box for the all here mentioned libraries, in particular, the change impact analysis and adjustment functionalities of the *locutor* core library are available at each's library web site. Currently we have round about 500 successful behavior-driven-design specifications mixed with structural specifications. This makes the author confident that the previously mentioned integration of full change management support into the *locutor* client is straightforward. However, concerning the "semantic costs", i.e. the time for the system to compute a change impact as well as the time for the user in case of interaction requests, must still be tested in a comprehensive system evaluation.

Part IV

Conclusion & Future Work

Chapter 10

Conclusion

Comprehensive up-front analysis of requirements during any kind of development pays high dividends by reducing the risk of costly rework and the potential for errors in planning estimates. The same concept appears to hold true for change impact analysis on heterogenous collections of documents. By identifying potential impacts before making a change, we greatly reduce the risks of embarking on a costly change because the cost of unexpected problems generally increases with the lateness of their discovery.

Change impact analysis information can be used for planning changes, making changes, accommodating certain types of changes, and tracing through the effects of changes. It makes the potential effects of changes visible before the changes are implemented to make it easier to perform changes more accurately and identifies the consequences or ripple effects of proposed changes during development and maintenance. There is often more than one change that can solve the same problem or satisfy the same requirement. Assessing the complete impact of each change is often necessary to be able to choose which change to apply. There are also, sometimes, external constraints that must be taken into account when designing the change, such as work packages to be interfaced with or parts of the system that must not be impacted. Change impact analysis helps to identify work products impacted by changes. Such analysis not only permits evaluation of the consequences of planned changes; it also allows trade-offs between suggested change approaches to be considered.

Change impact analysis can be used as a measure of the cost of a change. The more the change causes other changes, the higher the cost is. Carrying out this analysis before a change is made allows an assessment of the cost of the change and helps management choose tradeoffs between alternative changes. It allows to evaluate the appropriateness of a proposed modification. If a change that is proposed has the possibility of impacting large, disjoint sections of a project, the change might need to be re-examined to determine whether a safer change is possible.

In software engineering, change impact analysis can be used to drive regression testing, i.e., to determine the parts of a program that need to be re-tested after a change is made. Regression test is a software maintenance activity that refers to any repetition of tests (usually after software or data changes) intended to show that the software's behavior is unchanged except insofar as required by the change to the software or data [17]. To save effort, regression testing should retest only those parts that are impacted by the changes. During maintenance, when some changes have been made to the system, we need to estimate how many classes need to be retested. Retesting too many classes in the system will increase the cost of testing, but retesting too few classes in the system might adversely impact the quality of the software. Change impact analysis can also be used to indicate the vulnerability of critical sections of code. If a procedure that provides critical functionality is dependent on many different parts of a program, its functionality is susceptible to changes made in these parts.

A major goal of change impact analysis is to identify the work products impacted by proposed changes. Evaluating change impacts requires identifying what will be impacted by a change and relies on the "impact assessment" to determine quantitatively what the impact represents. Conceptually, it takes a list of life-cycle objects — e.g. from specifications to programs — analyzes these objects with respect to the change, and produces a list of items that should be addressed during the change process. Staff can use the information from such analysis to evaluate the consequences of planned changes as well as the trade-offs among the approaches for implementing the change.

We have seen the problems with a poorly implemented or inadequately designed process to handle change. So what are some of the positive organizational benefits of instituting and developing our mature change management process? They include:

- Improved visibility into and communication of changes across distributed enterprises.
- Improved ability to assure that only changes that provide true business benefit are approved.
- Improved ability to assure that all proposed changes are scheduled based on business priority, infrastructure impact and service risk.
- Improved ability to smoothly regress to a previous state in the event of change failure or unanticipated results.
- Time to implement changes is reduced.
- Disruptions to ongoing service provision are minimized.

In summary, the key aspect of this work is the fact that we have found the essential ingredients to bake a tasty MoC cake addressing all the here mentioned gustative nerves in sense of benefits. The author believes that the understanding of the structure and the interplay between the slices of our layered cake is his significant scientific contribution. Not only the individual layers themselves are an achievement but also their arrangement and interaction. Furthermore, the author sees his work as a kind of migration and integration work. Many scientists have discussed the individual layers and have achieved remarkable insights, solutions and tools. But for all the good ingredients, no one has baked a complete self-contained MoC cake.

As a partial validation of our management of change methodology, we presented the *locutor* system which supports both the change-and-fix and the Methodology for Software Evolution (MSE) processes [115] of change propagation adopted to heterogenous collections of semi-structured documents. While program code browsers [24, 116] deal with dependencies in software, they leave the interrelated documents to the user. Our change management system differs from browsers in the fact that it maintains information about both dependencies and inconsistencies in collection of documents, and provide a specialized but important kind of query: find all marked entities within the entire collection which have to be changed in order to make documents consistent. These kinds of tools help the author/programmer to be organized during the process of maintenance. The author believes that they may play an important role in the future and will keep on going his development on the *locutor* system to finally provide an an all-encompassing generic change management tool for semi-structured documents.

Chapter 11

Future Work

In this chapter we present the secondary points of this work. We divide them into two categories: the conceptual part and the part of implementations.

The Conceptual Level

We consider a complexity analysis, a termination analysis and a proof of soundness extremely important for our *sdiff* differencing algorithm. A complexity analysis, according to [30], explores an algorithm to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Is this true for *sdiff*? Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity). Further research has to determine the complexity function of *sdiff*.

A termination analysis, according to [137], attempts to determine whether the evaluation of a given program will definitely terminate. It is a form of program analysis that is related to the halting problem. Further research has to evaluate whether or not *sdiff* has a halting problem.

Regarding soundness, operating with edit scripts is complicated, because proof of validity of a given edit script will require considering the whole sequence of applications of its edit operations. Instead, we suggest to introduce normalized edit scripts and to proof that any valid edit script can be transformed into a Normal Form. These three points are extremely important because our *sdiff* algorithm, the middle layer of our MoC cake, keeps this together. This means our change impact analysis as well as our adjustment is based on a robust, efficient and accurate *sdiff* algorithm.

Moreover, regarding equivalence systems parametrized to our *sdiff* algorithm, we consider a concrete specification of the sub-classes covered by our EQ syntax as well as a proof that these are decidable to be mandatory. At the current state of this work only empirical results have shown that with our declarative EQ syntax one is able to described the substantial equivalences for change management but a formal investigation is lacking for a founded manifestation.

Another high-impact issue with respect to our change impact analysis and adjustment approach is an automated termination analysis on the respective graph transformations. Even if we assume individual graph rewriting rule systems do terminate, we are only sure that the combined abstraction and projections terminate as well. However, the combined propagation rule systems may well be non-terminating due to some ping-pong ripple effects. For this it would be highly desirable to have some automated termination analysis, which is only known for very restricted graph rewriting classes.

Further issues are version control commands on XML documents and canonical XML. According to Disclaimer 3.1, we have to fix/improve the definition of version control commands on XML documents. As a starting point we suggest to utilize the *locutor* XML attributes presented in Sec. 3.2.3. Further, we have to analyze the usage of canonical XML further: think of a document D_1 with no explicit attribute but using the DTD defaults. In a subsequent version, D_2, the attributes are explicitly added and modified. Further research has to tackle the following questions: How should the edit script look like? What are the benefits of canonicalization?

The Implementation Level

We consider dynamic switching of equivalence systems within our *sdiff* algorithm as extremely important. For dynamic switching of utilized equivalence system depending on the current document type we suggest to replace the signature of the *sdiff* algorithm by **def** *sdiff*($E:\mathscr{D}\to\mathbb{Q},\mathscr{L}:$ List[$\mathbb{FS}^+$], $\mathscr{R}:$ List[$\mathbb{FS}^+$]). This also enables us to switch equivalence systems within a document utilizing namespaces. This has particular implications for hybrid XML documents. Semi-structured documents composed of several different grammars, such as, for example, a DocBook XML document with mixed in UML XMI [109] and MathML [21] fragments can be investigated more fine-grained in contrast to a differential analysis, accepting only one equivalence system, for example, an equivalent systems for DocBook only.

Furthermore, we consider the versioning of SDI graphs as an important next step in our development in order to improve the tracing of change histories and thus to improve the overall visibility into and communication of changes. Therefore, we suggest to extend Subversion's `.svn` administrative directory structure by `.svn/.locutor` sub-directories storing the respective sub-graphs. In the same context the $\delta\mu\tilde{\mu}$-framework has to be fully integrated in order to gain all of the benefits of our *sdiff* algorithm. Only with the help of this framework the communication between the traditional Subversion server, based on ordinary edit scripts, and the improved Subversion command line client *locutor* is possible. As an alternative we propose to replace the conventional Subversion server by the improved one, TNTBase [162], so that *locutor* directly communicates with the backend via treeish edit scripts.

Regarding the EQ syntax we are currently working on the (1) handling of namespaces, (2) disambiguation of primary keys, and (3) the specification of equivalence criteria via simple typed Lambda expressions. For the former we suggest to extend ⟨*body*⟩ by an optional `scope ::= URI` declaration resulting in utilizing extended QNames. With respect to the second issue imagine to append a second e-mail address to a person record. The open question is "Which one of the piggy banks would you like?". Regarding the latter we suggest to extend ⟨*body*⟩ by an optional ⟨*funs*⟩ production rule in order to be able to specify even more complex equivalence constraints. Within a ⟨*funs*⟩ production rule one should be able to describe equivalence criteria via simple typed Lambda expressions.

A "nice-to-have" feature within our *locutor* command line client is previewing implicit changes within a document. With respect to the *diff-patch* axiom $patch(diff(\mathscr{T}_1,\mathscr{T}_2),\mathscr{T}_1) = \mathscr{T}_2$ we cannot change this statement to $patch(diff(\mathscr{T}_1,\mathscr{T}_2),\mathscr{T}_1) \sim_{\mathscr{E}} \mathscr{T}_2$. But we can utilize $\mathscr{E}$ for previewing potential ripple effects within a document. Let us assume the following dependency graph: $A \longleftarrow: \underline{\mathbb{t}}_1 - B -: \underline{\mathbb{t}}_2 \longrightarrow B'$. The document fragment A depends on the document fragment B via a relation of type $\underline{\mathbb{t}}_1$. Then document fragment B is modified to B' via change type $\underline{\mathbb{t}}_2$. If $B \sim_{\mathscr{E}} B'$ and $\underline{\mathbb{t}}_1$ does not correlate with $\underline{\mathbb{t}}_2$, i.e. no propagation along the relation between A and B, then $A \longleftarrow: \underline{\mathbb{t}}_1 - B'$ holds. Otherwise A is affected by the change on B. A common scenario for such a preview is in OWL world. Think of B and B' being to semantically equivalent OWL representations, then a OWL fragment A depending on the OWL fragment B is not affected by its syntactic change to another but equivalent OWL representation.

Eventually, to evaluate our MoC cake in total, the full integration of the *locutor* core library [99] into the *locutor* command line client [100] is highly important and already work in progress [101].

Bibliography

[1] Gartner Group for AIIM International Staff, C.: State of the Document Technologies Industry, 1997-2003. Association for Information and Image Management, Silver Spring, MD, USA (1999)

[2] Anderson, K., Sherba, S., Lepthien, W.: Towards Large-Scale Information Integration. In: Proceedings of the 24th International Conference on Software Engineering, pp. 524–534 (2002)

[3] Arnold, R.S., Bohner, S.A.: Impact Analysis - Towards a Framework for Comparison. In: ICSM '93: Proceedings of the Conference on Software Maintenance, pp. 292–301. IEEE Computer Society, Washington, DC, USA (1993)

[4] Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and hardness of approximation problems. In: In Proc. 33rd Ann. IEEE Symp. on Found. of Comp. Sci, pp. 14–23 (1992)

[5] Autexier, S.: DocTIP: Document and Tool Integration Platform (2010). `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/DocTIP`

[6] Autexier, S.: The GMoC Tool for Generic Management of Change (2010). `http://www.informatik.uni-bremen.de/dfki-sks/omoc/gmoc.html`

[7] Autexier, S., Hutter, D.: Mind the Gap - Maintaining Formal Developments in MAYA. In: Festschrift in Honor of J.H. Siekmann. Springer-Verlag, LNCS 2605 (2005)

[8] Autexier, S., Hutter, D., Mossakowski, T., Schairer, A.: The Development Graph Manager MAYA (system description). In: H. Kirchner (ed.) Proceedings of 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02), no. 2422 in LNCS. Springer Verlag (2002)

[9] Autexier, S., Müller, N.: Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents. In: Proceedings of the 10th ACM Symposium on Document Engineering (2010). To be published

[10] Barnard, D.T., Clarke, G., Duncan, N.: Tree-to-tree Correction for Document Trees. Tech. Rep. 95-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, Canada (1995)

[11] Bashir, M., Qadir, M.: Traceability Techniques: A Critical Study. In: Multitopic Conference, 2006. INMIC '06. IEEE, pp. 265–268 (2006). DOI 10.1109/INMIC.2006.358175

[12] Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. Tech. rep., Universität Karlsruhe (TH), Institut für Programmstrukturen und Datenorganisation (2010)

[13] Bohner, S.A.: A Graph Traceability Approach for Software Change Impact Analysis. Ph.D. thesis, George Mason University, Fairfax, VA, USA (1995)

[14] Bohner, S.A., Arnold, R.S.: Software Change Impact Analysis. John Wiley & Sons (1996)

[15] Bolwidt, E.: Jaxup - A Java XML Update engine (2010). Available at `http://klomp.org/jaxup/`

[16] Borg (Star Trek) (2009). `http://en.wikipedia.org/wiki/Borgs`

[17] Boris, B.: Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA (1990)

[18] Boyer, J.: Canonical XML Version 1.0. W3C recommendation, The World Wide Web Consortium (2001). URL `http://www.w3.org/TR/xml-c14n`

[19] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowan, J.: Extensible Markup Language (XML) 1.1 (Second Edition). W3C recommendation, The World Wide Web Consortium (2006). URL `http://www.w3.org/TR/xml11/`

[20] Bruce W. Speck Teresa R. Johnson, C.P.D., Heaton, L.B. (eds.): Collaborative Writing: An Annotated Bibliography. Greenwood Press, Westport, CT (1999)

[21] Carlisle, D., Ion, P., Miner, R., Poppelier, N.: Mathematical Markup Language (MathML) Version 2.0. W3C recommendation, The World Wide Web Consortium (2010). URL `http://www.w3.org/TR/MathML2/`

[22] Chawathe, S.S.: Comparing Hierarchical Data in External Memory. In: VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases, pp. 90–101. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)

[23] Chawathe, S.S., Rajaraman, A., Garcia-molina, H., Widom, J.: Change Detection in Hierarchically Structured Information. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 493–504 (1996)

[24] Chen, Y.F., Nishimoto, M.Y., Ramamoorthy, C.V.: The C Information Abstraction System. IEEE Trans. Softw. Eng. **16**(3), 325–334 (1990)

[25] Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C recommendation, The World Wide Web Consortium (2007). URL `http://www.w3.org/TR/xpath/`

[26] Cobena, G., Abiteboul, S., Marian, A.: Detecting Changes in XML Documents. In: ICDE, pp. 41–52. IEEE Computer Society (2002)

[27] Cole, R., Hariharan, R., Indyk, P.: Tree pattern matching and subset matching in deterministic $O(nlog^3n)$-time. In: SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, pp. 245–254. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1999)

[28] Collins-Sussman, B., Fitzpatrick, B.W., Pilato, M.: Version Control With Subversion. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2004)

[29] Collofello, J.S., Orn, M.: A practical software maintenance environment. In: Software Maintenance, pp. 45–51. Scottsdale, AZ, USA (1988)

[30] Analysis of algorithms (2009). `http://en.wikipedia.org/wiki/Complexity_analysis`

[31] Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., Smith, S.: Implementing Mathematics with the Nuprl Development System. Prentice-Hall (1986)

[32] Curbera, F., Epstein, D.: Fast difference and update of XML documents. In: XTech. San Jose (1999)

[33] darcs (2009). Available at `http://darcs.net`

[34] DCMI Home: Dublin Core Metadata Initiative (DCMI) (2010). Available at `http://dublincore.org`

[35] Dommitt Diff Tool (2009). Available at `http://www.dommitt.com`

[36] Dekkers, M., Weibel, S.: State of the Dublin Core Metadata Initiative. D-Lib Magazine **9**(4) (2003). URL `http://www.dlib.org/dlib/april03/weibel/04weibel.html`

[37] DeltaXML by Monsell EDM Ltd (2010). Available at `http://www.deltaxml.com`

[38] DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering **17**(9), 900–910 (1991). DOI http://dx.doi.org/10.1109/32.92910

[39] DFKI Bremen (2010). `http://www.dfki.de/web`

[40] DiffMK by Norman Walsh at Sun Microsystems (2009). Available at `http://www.sun.com/xml/developers/diffmk`

[41] Document Object Model (2006). URL `http://www.w3.org/DOM`. Seen June

[42] EDS: System level automation tool for enterprises (slate). `http://www.sdrc.com/slate`

[43] Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999)

[44] Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.): Handbook of graph grammars and computing by graph transformation: vol. 3: concurrency, parallelism, and distribution. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999)

[45] at EPFL, M.O.: The Scala Programming Language (2010). `http://www.scala-lang.org`

[46] Espinoza, A., Alarcon, P.P., Garbajosa, J.: Analyzing and Systematizing Current Traceability Schemas. In: Software Engineering Workshop, 2006. SEW '06. 30th Annual IEEE/NASA, pp. 21–32 (2006). DOI 10.1109/SEW.2006.12

[47] A Framework of Guidance for Building Good Digital Collections (2010). Available at http://framework.niso.org/

[48] Fogel, K.F.: Open Source Development with CVS. Coriolis Group Books, Scottsdale, AZ, USA (1999)

[49] Fontaine, R.L.: A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML (2001). Available at http://www.gca.org/papers/xmleurope2001/papers/pdf/s29-2.pdf

[50] Git - Fast Version Control System (2009). Available at http://git.or.cz

[51] Goldfarb, C.F.: The roots of sgml – a personal recollection (1996). Availabel at http://www.sgmlsource.com/history/roots.htm

[52] Goldfarb, C.F., Prescod, P.: XML Handbook. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)

[53] Goos, G.: GrGen.NET — A Generative Programming System for Graph Rewriting (2009). http://www.grgen.net

[54] Goos, G.: yComp — Das Anzeigesystem fÃijr Programmgraphen in VCG-Format (2009). http://www.info.uni-karlsruhe.de/software.php/id=6

[55] Gordian Consulting GmbH (2010). http://www.gordian-consulting.de

[56] Gotel, O., Finkelstein, C.: An Analysis of the Requirements Traceability Problem. In: Proceedings of the First International Conference on Requirements Engineering, pp. 94–101 (1994). DOI 10.1109/ICRE.1994.292398

[57] working group, I.R.: http://www.incose.org

[58] Gutwin, C., Greenberg, S., Roseman, M.: Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation. In: HCI '96: Proceedings of HCI on People and Computers XI, pp. 281–298. Springer-Verlag, London, UK (1996)

[59] Harel, D., Tarjan, R.E.: Fast Algorithms for Finding Nearest Common Ancestors. SIAM Journal on Computing **13**(2), 338–355 (1984). DOI http://dx.doi.org/10.1137/0213024

[60] Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, pp. 154–163. ACM, New York, NY, USA (1994). DOI http://doi.acm.org/10.1145/193173.195402

[61] Haynes, D.: Metadata: For Information Management and Retrieval. Facet Publishing (2004)

[62] Hets: The Heterogeneous Tool Set. Web site at http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/

[63] Hoffmann, C.M., O'Donnell, M.J.: Pattern Matching in Trees. Journal of the ACM **29**(1), 68–95 (1982)

[64] Horwitz, S., Thomas, R., Binkley, D.: Interprocedural slicing using dependence graphs. SIGPLAN Not. **39**(4), 229–243 (2004). DOI http://doi.acm.org/10.1145/989393.989419

[65] Hunt, J.W., McIlroy, M.D.: An Algorithm for Differential File Comparison. Tech. Rep. CSTR 41, Bell Laboratories, Murray Hill, NJ (1976)

[66] Hutter, D.: Management of Change in Verification Systems. In: Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000, pp. 23–34. IEEE Computer Society (2000)

[67] Hutter, D., Schairer, A.: Towards an Evolutionary Formal Software Development. In: Proceedings 16th IEEE International Conference on Automated Software Engineering, ASE-2001. IEEE Computer Society, San Diego, USA (2001)

[68] Hwang, Y.F.: Detecting faults in chained-inference rules in information distribution systems. Ph.D. thesis, George Mason University, Fairfax, VA, USA (1998)

[69] Ionescu, M.D., Ionescu, M.D.: xProxy: A Transparent Caching and Delta Transfer System for Web Objects (2000). Unpublished manuscript

[70] IT Infrastructure Library (2009). URL `http://www.itil-itsm-world.com/index.htm`

[71] Jacobs University Bremen (2010). `http://www.jacobs-university.de`

[72] Kaiser, G.E., Feiler, P.H., Popovich, S.S.: Intelligent assistance for software development and maintenance. IEEE Softw. **5**(3), 40–49 (1988). DOI http://dx.doi.org/10.1109/52.2023

[73] Karger, D.R., Jones, W.: Data unification in personal information management. Commun. ACM **49**(1), 77–82 (2006). DOI http://doi.acm.org/10.1145/1107458.1107496

[74] Keables, J., Roberson, K., von Mayrhauser, A.: Data flow analysis and its application to software maintenance. In: Proceedings of the Conference on Software Maintenance, pp. 335–347. IEEE CS Press, Los Alamitos, CA. (1988)

[75] Kent, W.: A simple Guide to Five Normal Forms in Relational Database Theory. Commun. ACM **26**(2), 120–125 (1983). DOI http://doi.acm.org/10.1145/358024.358054

[76] Korel, B., Laski, J.: Dynamic slicing of computer programs. The Journal of Systems and Software **13**(3), 187–195 (1990). DOI http://dx.doi.org/10.1016/0164-1212(90)90094-3

[77] Krieg-Brückner, B., Lindow, A., Lüth, C., Mahnke, A., Russell, G.: Semantic Interrelation of Documents via an Ontology. In: e-Learning Fachtagung Informatik, 6.-8. September 2004, pp. 271–282. Springer-Verlag (2004)

[78] Krieg-Brückner, B., Mahnke, A.: Semantic Interrelation and Change Management. In: OMDoc – An open markup format for mathematical documents [Version 1.2], no. 4180 in LNAI, chap. 26.6, pp. 274–277. Springer Verlag (2006). URL `http://omdoc.org/omdoc1.2.pdf`

[79] Laux, A., Martin, L.: XUpdate - XML Update Language. Available at http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html

[80] Loyall, J.P., Mathisen, S.A.: Using dependence analysis to support the software maintenance process. In: ICSM '93: Proceedings of the Conference on Software Maintenance, pp. 282–291. IEEE Computer Society, Washington, DC, USA (1993)

[81] Lyle, J.R., Wallance, D.R., Graham, J.R., Gallagher, K.B., Poole, J.P., Binkley, D.W.: Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 1: Requirements and Design. National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899 (1990)

[82] MacKenzie, D., Eggert, P., Stallman, R.: Comparing and Merging Files with GNU diff and patch. Network Theory Ltd. (2003)

[83] Madhavji, N.H.: Environment Evolution: The Prism Model of Changes. IEEE Trans. Software Eng. **18**(5), 380–392 (1992)

[84] Mahnke, A., Scheffczyk, J.: Engineering Mathematical Knowledge. In: M. Kohlhase (ed.) Mathematical Knowledge Management, MKM'05, no. 3863 in LNAI. Springer Verlag (2005)

[85] Malhotra, A., Melton, J., Walsh, N.: XQuery 1.0 and XPath 2.0 Functions and Operators. W3C recommendation, The World Wide Web Consortium (2007). URL http://www.w3.org/TR/xpath-functions/

[86] MARC Standards (2010). Available at http://www.loc.gov/marc

[87] Marsh, J., Orchard, D., Veillard, D.: XML Inclusions (XInclude) Version 1.0 (Second Edition). W3C recommendation, The World Wide Web Consortium (2006). URL http://www.w3.org/TR/xinclude/

[88] Mason, P., Cosh, K., Vihakapirom, P.: On Structuring Formal, Semi-Formal and Informal Data to Support Traceability in Systems Engineering Environments. In: CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management, pp. 642–651. ACM, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/1031171.1031288

[89] McCabe & Associates, I.: Battlemap Analysis Tool Reference Manual. McCabe & Associates, Inc., Twin Knolls Professional Park, 5501 Twin Knolls Road, Columbia (1992)

[90] microtec consulting GmbH (2010). http://microtec-consulting.de

[91] Monotone: reliable, distributed Version Control (2009). URL http://www.monotone.ca

[92] Moreton, R.: A Process Model for Software Maintenance. Journal Information Technology **5**, 100–104 (1990)

[93] Moser, L.E.: Data dependency graphs for ada programs. IEEE Trans. Softw. Eng. **16**(5), 498–509 (1990). DOI http://dx.doi.org/10.1109/32.52773

[94] Mossakowski, T.: Heterogeneous Specification and the Heterogeneous Tool Set. Habilitation, Universität Bremen (2005)

[95] Mossakowski, T., Autexier, S., Hutter, D.: Extending development graphs with hiding. In: H. Hußmann (ed.) Fundamental Approaches to Software Engineering (FASE 2001), no. 2029 in LNCS, pp. 269–284. Springer Verlag (2001)

[96] Mossakowski, T., Autexier, S., Hutter, D.: Development Graphs – Proof Management for Structured Specifications. Journal of Logic and Algebraic Programming **67**(1–2), 114–145 (2006)

[97] Mossakowski, T., Hoffman, P., Autexier, S., Hutter, D.: Part iv: CASL logic. In: B. Krieg-Brückner, P. Mosses (eds.) The CASL Reference Manual. Springer-Verlag, LNCS 2960 (2004)

[98] Müller, N.: An Ontology-Driven Management of Change. In: K.D. Althoff, M. Schaaf (eds.) LWA, *Hildesheimer Informatik-Berichte*, vol. 1/2006, pp. 186–193. University of Hildesheim, Institute of Computer Science (2006). URL http://kwarc.info/nmueller/papers/lwa06.pdf

[99] Müller, N.: *locutor* - An Ontology-driven Management of Change System (2009). http://code.google.com/p/locutor

[100] Müller, N.: *locutor* - An Ontology-driven Management of Change System (Command Line Client) (2009). https://svn.kwarc.info/repos/locutor/src/svnkitx

[101] Müller, N.: A Command Line Client for Management of Change on Semi-structured Documents (2010). http://code.google.com/p/deltaq

[102] Müller, N.: SCALAXX - Scala XML Extensions (2010). http://code.google.com/p/scalaxx

[103] Müller, N.: SCAUP - SCAla UP! (2010). http://code.google.com/p/scaup

[104] Müller, N., Kohlhase, M.: Fine-Granular Version Control & Redundancy Resolution. In: J. Baumeister, M. Atzmüller (eds.) LWA, pp. 1–8. Universität Würzburg (2008). URL http://www.kwarc.info/nmueller/papers/lwa08-fst.pdf

[105] Müller, N., Wagner, M.: Towards Improving Interactive Mathematical Authoring by Ontology-driven Management of Change. In: A. Hinneburg (ed.) LWA, pp. 289–295. Martin-Luther-University Halle-Wittenberg (2007)

[106] Myers, E.W.: An O(ND) Difference Algorithm and Its Variations. Algorithmica **1**, 251–266 (1986)

[107] NIAGARA Query Engine (2009). Available at http://www.cs.wisc.edu/niagara

[108] Open Archives Initiative (2010). Available at http://www.openarchives.org

[109] Object Management Group, I.: XML Metadata Interchange (2010). http://www.omg.org/technology/documents/formal/xmi.htm

[110] Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide, 1st edn. Artima Inc (2008)

[111] Offutt, A.J.: An integrated automatic test data generation system. Journal of Systems Integration **1**(3-4), 391–409 (1991)

[112] Offutt, A.J., Irvine, A.: Testing object-oriented software using the category-partition method. In: Seventeenth International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS USA '95), pp. 293–304. Santa Barbara, CA (1995)

[113] Pfleeger, S.L., Bohner, S.A.: A framework for software maintenance metrics. In: Proceedings of the International Conference on Software Maintenance (1990)

[114] Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)

[115] Rajlich, V.: MSE: a methodology for software evolution. Journal of Software Maintenance **9**(2), 103–124 (1997)

[116] Rajlich, V., Damaskinos, N., Linos, P., Khorshid, W.: Vifor: a tool for software maintenance. Softw. Pract. Exper. **20**(1), 67–77 (1990). DOI http://dx.doi.org/10.1002/spe.4380200108

[117] Ramesh, B., Jarke, M.: Towards reference models for requirements traceability. IEEE Transactions on Software Engineering **27**(1) (2001)

[118] Reiser4. `http://en.wikipedia.org/wiki/Reiser4` (2010). URL `http://en.wikipedia.org/wiki/Reiser4`

[119] Rochkind, M.J.: The Source Code Control System. IEEE Trans. Software Eng. **1**(4), 364–370 (1975)

[120] Rombach, H., Ulery, B.: Improving software maintenance through measurement. Proceedings of the IEEE **17**(4), 581–595 (1989)

[121] Rozenberg, G. (ed.): Handbook of graph grammars and computing by graph transformation: volume I. foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)

[122] RTCA Inc.: Do-178b, software considerations in airborne systems and equipment certification. `http://www.rtca.org` (1992)

[123] Russell, B.: Principles of Mathematics. HarperCollins Publishers Ltd; 2nd edition (1937)

[124] Schairer, A., Hutter, D.: Proof Transformations for Evolutionary Formal Software Development. In: Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002. Springer-Verlag, LNCS 2422 (2002)

[125] S.Chawathe, S., Garcia-Molina, H.: Meaningful change detection in structured data. In: SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, pp. 26–37. ACM, New York, NY, USA (1997). DOI http://doi.acm.org/10.1145/253260.253266

[126] Scheffczyk, J., Borghoff, U.M., Rödig, P., Schmitz, L.: A Comprehensive Description of Consistent Document Engineering. Report, University of the Federal Armed Forces Munich (2003)

[127] Schröder, B.: Ordered Sets, first edn. Birkhäuser Boston, Lousianna Tech University, Rustion, LA 71272, USA (2002)

[128] Selkow, S.M.: The Tree-to-Tree Editing Problem. Information Processing Letters **6**(6), 184–186 (1977)

[129] Sutton, M.J.D.: Document Management for the Enterprise: Principles, Techniques, and Applications. Wiley (1996). 400 pages

[130] The SVK Version Control System (2009). Available at `http://svk.elixus.org/view/HomePage`

[131] SVNKit - The only pure Java Subversion library in the world! (2009). Available at `http://svnkit.com`

[132] Documentation on SVNKit API (2009). Available at `http://svnkit.com/javadoc/index.html`

[133] SVNKit System Architecture (2009). Available at `https://wiki.svnkit.com/SVNKit_Architecture`

[134] SvnX is an open source GUI for most features of the svn client binary (2008). Available at `http://www.lachoseinteractive.net`

[135] Tai, K.C.: The Tree-to-Tree Correction Problem. Journal of the ACM **26**(3), 422–433 (1979)

[136] Telelogic: Doors XT. `http://www.telelogic.com`

[137] Termination analysis (2009). `http://en.wikipedia.org/wiki/Termination_analysis`

[138] Tichy, W.F.: RCS - A System for Version Control. Software - Practice and Experience **15**, 637–654 (1985)

[139] Universität des Saarlandes (2010). `http://www.uni-saarland.de`

[140] van der Vlist, E.: RELAX NG. O'Reilly Media (2003)

[141] VM Tools (2009). Available at `http://www.vmsystems.net/vmtools`

[142] XML Schema (2006). URL `http://www.w3.org/XML/Schema`. Seen June

[143] Wagner, M.: Change-Oriented Architecture for Mathematical Authoring Assistance. Ph.D. thesis, FR 6.2 Informatik, Universität des Saarlandes (2010). Forthcoming

[144] Wagner, R.A., Fischer, M.J.: The String-to-String Correction Problem. Journal of the ACM **21**(1), 168–173 (1974). DOI http://doi.acm.org/10.1145/321796.321811

[145] Wang, Y., DeWitt, D.J., yi Cai, J.: X-Diff: An Effective Change Detection Algorithm for XML Documents. In: U. Dayal, K. Ramamritham, T.M. Vijayaraman (eds.) ICDE, pp. 519–530. IEEE Computer Society (2003)

[146] Weiser, M.: Program slicing. In: ICSE '81: Proceedings of the 5th international conference on Software engineering, pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)

[147] White, L.J.: A firewall concept for both control-flow and data-flow in regression integration testing. IEEE Transactions on Software Engineering pp. 171–262 (1992)

[148] Wieringa, R.: Traceability and Modularity in Software Design. In: Proceedings of the 9th International Workshop on Software Specification and Design, pp. 87–95 (1998). DOI 10.1109/IWSSD.1998.667923

[149] X-Diff – Detecting Changes in XML Documents (2009). Available at http://pages.cs.wisc.edu/~yuanwang/xdiff.html

[150] Xerces - C++ XML Parser by Apache (2010). Available at http://xerces.apache.org/xerces-c/

[151] xmldiff by LogiLab (2010). Available at http://www.logilab.org/xmldiff

[152] XML Diff and Merge Tool by IBM (2010). Available at http://alphaworks.ibm.com/tech/xmldiffmerge

[153] XML TreeDiff by IBM (2010). Available at http://alphaworks.ibm.com/tech/xmltreediff

[154] XSH - XML Editing Shell (2010). Available at http://xsh.sourceforge.net/

[155] XyDiff Tools: Detecting changes in XML Documents (2010). Available at http://leo.saclay.inria.fr//software/XyDiff/cdrom/www/xydiff/index-eng.htm

[156] Xyleme - Learning Content Management System (LCMS) (2010). Available at http://www.xyleme.com/

[157] Yau, S.S., Collofello, J., MacGregor, T.: Ripple Effect Analysis of Software Maintenance. Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International pp. 60–65 (1978)

[158] Z39.50: A Primer on the Protocol. http://www.niso.org/publications/press/Z3950_primer.pdf (2002). URL http://www.niso.org/publications/press/Z3950_primer.pdf

[159] Zhang, K., Shasha, D.: Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. SIAM Journal on Computing **18**(6), 1245–1262 (1989)

[160] Zhang, K., Statman, R., Shasha, D.: On the Editing Distance Between Unordered Labeled Trees. Information Processing Letters **42**(3), 133–139 (1992)

[161] Zhang, K., Statman, R., Shasha, D.: On the Editing Distance between Unordered Labeled Trees. Information Processing Letters **42**(3), 133–139 (1992). DOI http://dx.doi.org/10.1016/0020-0190(92)90136-J

[162] Zholudev, V.: TNTBase - Versioned Storage for XML (2010). http://tntbase.org/