Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik

A Change-Oriented Architecture for Mathematical Authoring Assistance

Marc Wagner

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken, 2010

Dekan:	Prof. Dr. Holger Hermanns
Vorsitzender:	Prof. Dr. Gert Smolka
Gutachter:	Prof. Dr.-Ing. Jörg H. Siekmann
	Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm
	Prof. Dr. Fairouz Kamareddine
Beisitzer:	Priv.-Doz. Dr. Helmut Horacek
Kolloquium:	15.11.2010

Geleitwort

The monograph is the dissertation of Marc Wagner. Motivated by the problem to support the writing of mathematical documents by mathematical assistance systems, it focuses on the architecture of distributed systems in general with the application to bridge the gap between the mathematical language used in documents and the formal logic representations of mathematical assistance systems.

In distributed software systems the representations of the internal states of each subsystem are diverse and need to be synchronized. State of the art is that each subsystem synchronizes with the connected subsystems by pushing the updates of its internal state or pulling the partners' internal state. This design of interfaces has the drawback that each subsystem needs to know the signature of interface functions of other subsystems and how to map its own data representation to the remote data representation and back. Consequently, any change in either the interface functions or the data representations requires adaptations in all connected subsystems. The solution proposed by Marc Wagner is the document-centric *change-oriented architecture* (COA) design pattern for distributed software systems, where the internal state of each software module is viewed as a document and the communication between software modules is exclusively based on exchanging updates on the documents. Mediators responsible to bidirectionally transform the different document types form the glue layer between two software modules. The mediators are based on an invertible grammar formalism to describe the document transformations and provide efficient support to map changes in one document format to another.

The design pattern is applied to interface a text-editor used to author mathematical documents with a proof assistance system. The major problem in interfacing these systems is to mediate between the texts in the editor written in natural language mixed with mathematical notations for concepts, formulas and proofs, and the logic based representation in the proof assistance system. Following a controlled natural language approach for writing the mathematical documents, Marc Wagner demonstrates how the invertible grammar formalism in the mediators of the *change-oriented architecture* can be used to propagate changes made by the author from the text-editor to the mathematical assistance system and back, for instance, to generate proof descriptions in natural language. Furthermore, he demonstrates how the problem to dynamically extend grammars to accommodate new concepts and notations defined in the documents can be solved using the change-oriented architecture design pattern.

Agility of software development processes is a major issue in order to efficiently cope with the evolution of software to adapt it to new requirements. Keeping as much as possible the impacts of changes of individual modules local to these modules is essential for an agile software development process. The change-oriented architecture proposed in this thesis and its separation of logic and data at the interfaces between distributed software components provides an interesting design pattern for distributed software systems that addresses issues from software evolution already in the software architecture.

Dr. Serge Autexier, DFKI GmbH (Deutsches Forschungszentrum für Künstliche Intelligenz), Sichere Kognitive Systeme, Bremen, Januar 2011

Kurzzusammenfassung

Das computergestützte Verfassen mathematischer Dokumente mit einem wissenschaftlichen Text-Editor erfordert neue Techniken des mathematischen Wissensmanagements und der Wissenstransformation, um die Arbeitsweise eines Assistenzsystems, wie die des ΩMEGA Systems, zu organisieren. Die Herausforderung besteht darin, dass im gesamten System verteilt verschiedene Arten von vorgegebenem und abgeleitetem Wissen existieren, die in unterschiedlichen Formen und Abhängigkeiten auftreten. Wenn Änderungen in diesen Wissenseinheiten auftreten, müssen diese effektiv propagiert werden.

In dieser Arbeit wird eine *Änderungs-Orientierte Architektur* für das computergestützte Verfassen mathematischer Dokumente vorgestellt. Dabei werden Dokumente als Schnittstelle verwendet und die Komponenten der Architektur interagieren, indem sie aktiv die Schnittstellendokumente ändern und auf Änderungen reagieren. Um diese Art der Interaktion zu optimieren, werden in dieser Arbeit zwei wesentliche Methoden vorgestellt. Zum einen wird eine effiziente Methode zur Berechnung von gewichteten semantischen Änderungen zwischen zwei Versionen eines Dokumentes entwickelt. Zum anderen wird ein umkehrbarer Grammatikformalismus vorgestellt, der zur automatisierten bidirektionalen Transformation von Schnittstellendokumenten dient.

Die vorgestellte Architektur ist eine vollwertige Grundlage zum computergestützten Verfassen mathematischer Dokumente mit semantischen Annotationen und einer kontrollierten mathematischen Sprache.

Abstract

The computer-assisted authoring of mathematical documents using a scientific text-editor requires new mathematical knowledge management and transformation techniques to organize the overall workflow of an assistance system like the ΩMEGA system. The challenge is that, throughout the system, various kinds of given and derived knowledge units occur in different formats and with different dependencies. If changes occur in these pieces of knowledge, they need to be effectively propagated.

We present a *Change-Oriented Architecture* for mathematical authoring assistance. Thereby, documents are used as interfaces and the components of the architecture interact by actively changing the interface documents and by reacting on changes. In order to optimize this style of interaction, we present two essential methods in this thesis. First, we develop an efficient method for the computation of weighted semantic changes between two versions of a document. Second, we present an invertible grammar formalism for the automated bidirectional transformation between interface documents.

The presented architecture provides an adequate basis for the computer-assisted authoring of mathematical documents with semantic annotations and a controlled mathematical language.

Zusammenfassung

Das computergestützte Verfassen mathematischer Dokumente mit einem wissenschaftlichen Text-Editor erfordert neue Techniken des mathematischen Wissensmanagements und der Wissenstransformation, um die Arbeitsweise eines mathematischen Assistenzsystems, wie die des ΩMEGA Systems, zu organisieren. Die Herausforderung besteht darin, dass im gesamten System verteilt verschiedene Arten von vorgegebenem und abgeleitetem Wissen existieren, die in unterschiedlichen Formaten und Abhängigkeiten auftreten. Diese Wissenseinheiten und deren Abhängigkeiten müssen gepflegt werden und, wenn Änderungen in einer Komponente auftreten, müssen diese effektiv propagiert werden.

In dieser Arbeit wird eine *Änderungs-Orientierte Architektur* für das computergestützte Verfassen mathematischer Dokumente vorgestellt. Dabei werden Dokumente als Schnittstelle verwendet und die Komponenten der Architektur interagieren, indem sie aktiv die Schnittstellendokumente ändern und auf Änderungen reagieren. Um diese Art der Interaktion zu optimieren, werden in dieser Arbeit zwei wesentliche Methoden vorgestellt.

Zum einen wird eine effiziente Methode zur Berechnung von gewichteten semantischen Änderungen zwischen zwei Versionen eines Dokumentes entwickelt. Eine *Ähnlichkeitsspezifikation* wird eingeführt, die es erlaubt zu definieren, wann zwei Teile eines Dokumentes als ähnlich oder gleich zu betrachten sind. Da die Elemente eines Schnittstellendokumentes Objekte in der entsprechenden Komponente repräsentieren, wird es der Komponente ermöglicht, die geschätzten Auswirkungen für die Änderung oder Löschung dieser Objekte in einer *Änderungsspezifikation* zu gewichten. Das Problem der gewichteten Baum-zu-Baum Korrektur wird hierbei auf ein Suchproblem reduziert.

Zum anderen wird ein umkehrbarer Grammatikformalismus vorgestellt, der zur automatisierten bidirektionalen Transformation von Schnittstellendokumenten verschiedener Komponenten dient. Zu den Regeln der Grammatik können Unifikationsbedingungen formuliert werden, die durch den konstruierten Syntaxbaum erfüllt werden müssen. Um nicht-transformierte Teile eines Dokumentes bei der Rücktransformation generieren zu können, wird der Ablauf der Transformation gespeichert. Die Verarbeitungsreihenfolge einer Grammatikregel kann deklarativ angegeben werden, womit vorhandenes Wissen über den Informationsfluss in einem Dokument in die Grammatikregeln eingebettet werden kann.

Die vorgestellte Architektur ist eine vollwertige Grundlage für das computergestützte Verfassen mathematischer Dokumente mit semantischen Annotationen und einer kontrollierten mathematischen Sprache. Zur Entwicklung eines Prototyps wurde das ΩMEGA System mit den Text-Editoren TEXMACS und MS WORD integriert.

Anhand eines Vorlesungs-Szenarios beschreiben wir die Einsatzmöglichkeiten und Vorteile der *Änderungs-Orientierten Architektur*. Zu diesem Zweck haben wir in einer explorativen Studie mit einem Erstsemesterkurs der Fachrichtung Mathematik an der Universität des Saarlandes die Voraussetzungen für mathematische Autorenunterstützung analysiert. Das Änderungsverhalten und ausgewählte linguistische Aspekte wurden für unser Szenario ausgewertet.

Um die Einführung von neuen Konzepten in Vorlesungsskripten zu unterstützen, wird eine neue Methode entwickelt, die zu den in einem Dokument eingeführten Notationen dynamisch entsprechende umkehrbare Grammatikregeln erzeugt. Dies ermöglicht selbsterweiterbare Dokumente und die Verwendung der vom Autor angegebenen Notation für die inkrementelle Analyse und Erweiterung des Dokumentes.

Zur interaktiven Bearbeitung von Übungsaufgaben werden inkrementelle Verfahren zur Erzeugung von Beweisverpflichtungen aus Beweisskizzen entwickelt, die in einer kontrollierten mathematischen Sprache oder in natürlicher Sprache mit semantischen Annotationen verfasst sind. Die Unterstützung des mathematischen Assistenzsystems ΩMEGA kann über eine Interaktionssprache direkt innerhalb des Dokumentes angefordert werden. Feedback wie beispielsweise der Verifikationsstatus von Beweisschritten wird transparent in das Dokument integriert.

Die *Änderungs-Orientierte Architektur* trägt der nicht-monotonen Entwicklung mathematischen Wissens Rechnung, indem sie eine einheitliche adaptive Schnittstelle zwischen den Komponenten des ΩMEGA Systems und zur Einbindung externer Systeme zur Verfügung stellt.

Extended Abstract

The computer-assisted authoring of mathematical documents using a scientific text-editor requires new mathematical knowledge management and transformation techniques to organize the overall workflow of a mathematical assistance system like the ΩMEGA system. The challenge is that, throughout the system, various kinds of given and derived knowledge occur in different formats and with different dependencies. These pieces of knowledge and their dependencies need to be maintained and, if changes occur in any component, they need to be effectively propagated.

We present a Change-Oriented Architecture for the mathematical authoring assistance with the ΩMEGA system. Thereby, documents are used as interfaces and the components of the architecture interact by actively changing the interface documents and by reacting on changes. In order to optimize this style of interaction, we present two essential methods in this thesis.

First, we develop an efficient method for the computation of weighted semantic changes between two versions of a document. We introduce a *similarity specification* that allows for defining when two parts of a document are to be considered similar or equal. Since elements of the interface document represent objects in the interfaced component, we allow for specifying the estimated weights for changing or deleting these objects in an *edit specification*. We reduce the weighted tree-to-tree correction problem to a search problem.

Second, we present an invertible grammar formalism for the automated bidirectional transformation between interface documents of different components. Unification constraints can be attached to the grammar rules that have to be satisfied by the constructed parse tree. In order to be able to generate non-translated parts of the document in the inverse translation, we store the trace of the transformation. The recursive processing order for a grammar rule can be declaratively specified, thus the knowledge about the information flow in the document can be embedded into the grammar rules.

The presented architecture provides an adequate basis for the computer-assisted authoring of mathematical documents with semantic annotations and a controlled mathematical language. For this purpose, we have developed a prototype system for mathematical authoring assistance by integrating the ΩMEGA system with the text-editors TEXMACS and MS WORD.

Using a course scenario we describe the capabilities and benefits of a *Change-Oriented Architecture*. Therefore, we have analyzed the requirements for mathematical authoring assistance in an exploratory study with a first year mathematics course at Saarland University. The authoring behavior and selected linguistic aspects were evaluated for our scenario.

In order to support the introduction of new concepts in lecture notes, we develop a new method which dynamically synthesizes invertible grammar rules for the notation introduced in a document. This allows for self-extensible documents and for the exploitation of the notation defined by an author for the incremental document analysis and generation.

For the interactive authoring of exercise solutions we develop incremental methods to generate proof obligations from the proof sketches written in a controlled mathematical language or in full natural language with semantic annotations. Thereby, the support of the mathematical assistance system ΩMEGA can be request inside of the document using an appropriate interaction language. Feedback, like for example the verification state of proof steps, is transparently integrated with the document.

The *Change-Oriented Architecture* accommodates the non-monotonic evolution of mathematical knowledge by establishing a uniform and adaptive interface between the components of the ΩMEGA system as well as external systems.

Acknowledgments

Als erstes möchte ich mich bei meinem Betreuer Serge Autexier bedanken, der mich über so viele Jahre hinweg unterstützt hat. Seine gleichermaßen begeisternde wie kritische Art hat mir immer geholfen, meine Arbeit weiter zu verbessern. Meinem Doktorvater Jörg Siekmann danke ich für die beständige Förderung, das große Vertrauen und die Möglichkeit zur Promotion in seiner Arbeitsgruppe. Die Zusammenarbeit mit einer so großen Gruppe von exzellenten Wissenschaftlern war eine Bereicherung für meine Arbeit.

Ich möchte Fairouz Kamareddine und Reinhard Wilhelm für die Begutachtung meiner Arbeit danken. Darüber hinaus waren die interessanten Diskussionen und Ratschläge sehr wertvoll für die Entwicklung meiner Dissertation.

Mein besonderer Dank gilt Stephan Busemann, der mein Verständnis der Computerlinguistik geschärft hat und mir immer mit Rat und Tat zur Seite stand. Ebenso danke ich Claus-Peter Wirth für viele inspirierende Diskussionen, und für seine guten Ratschläge gerade auch in schwierigen Phasen der Arbeit. Selbstverständlich möchte ich mich bei meinem langjährigen Büropartner Marvin Schiller für die hervorragende Arbeitsatmosphäre und die Unterstützung und Motivation bedanken.

Meinen Studenten Oliver Bender und Thorsten Hey danke ich für ihre ausgezeichnete Mitarbeit. Mit ihren erfolgreichen Abschlussarbeiten unterstützen sie die Ergebnisse meiner Arbeit.

Während meiner Dissertation konnte ich mit vielen hervorragenden Kollegen interessante Projekte bearbeiten. Für die produktive Zusammenarbeit möchte ich allen meinen Co-Autoren herzlich danken: David Aspinall, Christoph Benzmüller, Dominik Dietrich, Armin Fiedler, Henri Lesourd, Christoph Lüth, Christine Müller, Normen Müller, Thomas Neumann, Ewaryst Schulz.

Für viele konstruktive Diskussionen, wertvolles Feedback und schöne Kaffeepausen bedanke ich mich bei allen meinen Kollegen der ΩMEGA Gruppe (Jörg Siekmann), der SKS Gruppe des DFKI Bremen (Bernd Krieg-Brückner) und der ACTIVEMATH Gruppe des DFKI Saarbrücken (Erica Melis). Insbesondere möchte ich hier Andreas Franke, Frank Theiss, Helmut Horacek, Martin Pollet, Jürgen Zimmer, Quoc Bao Vo, Chad Brown, Gueorguie Dobrev, Christoph Lüth, Till Mossakowski, Dieter Hutter, Lutz Schröder, Christian Maeder, Mattias Werner, George Goguadze, Eric Andres, Michael Dietrich, Ahmad Salim Doost, Paul Libbrecht, Bruce McLaren, Oliver Scheuer, Dimitra Tsovaltzi, Stefan Winterstein, Sergey Sosnovsky und Martin Homik danken.

In meiner Arbeit konnte ich von kurzen Forschungsaufenthalten an der Heriot-Watt University, der University of Edinburgh und der International University Bremen profitieren. Ich möchte mich für diese Möglichkeiten bei Fairouz Kamareddine, Alan Bundy und Michael Kohlhase bedanken. Für den familiären Empfang und die schöne Zeit danke ich Manuel Maarek, Krzystof Retel, Robert Lamar, Joe Wells, David Aspinall, Lucas Dixon, Roy McCasland, Fiona McNeill, Moa Johansson, Alan Smaill, Graham Steel, Normen Müller, Christine Müller, Christoph Lange, Achim Mahnke, Fulya Horozal und Florian Rabe.

Für die Möglichkeit zur Durchführung einer explorativen Studie mit einem Erstsemesterkurs Mathematik möchte ich mich bei Volker John und Michael Roland von der Fachrichtung Mathematik der Universität des Saarlandes bedanken.

Der grundlegende Aufbau dieser Dissertation beruht auf interessanten Gesprächen mit Geoff Sutcliffe, in denen ich seinen professionellen Pragmatismus zu schätzen gelernt habe. Vielen Dank für die wertvollen Tipps und Ratschläge.

Ein besonderer Dank gilt der Studienstiftung des deutschen Volkes, die meine Arbeit ideell und finanziell gefördert hat. Die Teilnahme an Akademien und der Gedankenaustausch mit Kollegen auf Doktorandenforen waren wertvolle Erfahrungen. Insbesondere möchte ich Matthias Frenz, Hans-Ottmar Weyand und Uwe Hartmann für interessante Gespräche und das entgegengebrachte Vertrauen danken. Ebenso danke ich der Graduiertenschule der Saarbrücker Informatik, insbesondere Michelle Carnell, für die anschließende Abschlussförderung meiner Arbeit.

Diese Arbeit wäre nicht möglich gewesen ohne die kontinuierliche Motivation und Unterstützung durch meine liebe Frau Esther, meine Eltern und meine Schwester. Herzlichen Dank an meine Familie.

Contents

I
Introduction

1 Motivation

The computer-assisted authoring of mathematical documents using a scientific text-editor requires new mathematical knowledge management and transformation techniques to organize the overall workflow of a mathematical assistance system like the ΩMEGA system. The challenge is that, throughout the system, various kinds of given and derived knowledge occur in different formats and with different dependencies. These pieces of knowledge and their dependencies need to be maintained and, if changes occur in any component, they need to be effectively propagated. The motivation of this thesis is to solve the following two mediation problems.

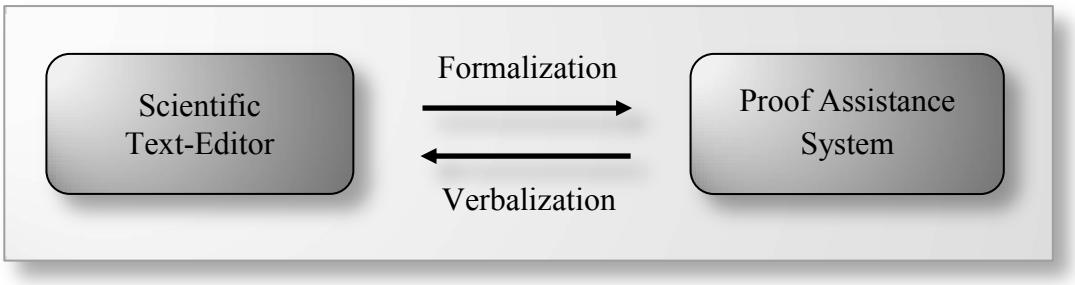
First, the *mediation problem in-the-small* considers the mediation between the mathematical documents which are written in a text-editor using a controlled mathematical language or full natural language with semantic annotations, and their formalization for the proof assistance system ΩMEGA. Efficient propagation of changes in mathematical knowledge is essential. By transforming the whole document, we would overwrite the whole content of the document in the text-editor. Consequently, we would lose large parts of the natural language text written by the user. In the other direction, we would lose the verification previously performed by the proof assistance system. In order to support the self-extensible nature of mathematical documents, the mediation has to exploit the notation defined by an author for the incremental document analysis and generation.

Second, the *mediation problem in-the-large* considers the mediation of knowledge between the different components of the ΩMEGA system. Large mathematical assistance systems are usually developed by several people in parallel, each of them working on different aspects and adding new functionalities. The challenge is to keep the system maintainable and at the same time ready for fast and simple inclusions of new functionalities provided by third-party systems. Furthermore, the evolutionary development of a knowledge-based system usually results in frequent changes of the component interfaces, for example the addition of new methods that allow for more fine-grained modifications of knowledge. There is a need for a uniform component interface that reduces the amount of change management needed when the components evolve in parallel.

The goal of this thesis is the development of a *Change-Oriented Architecture* for *Mathematical Authoring Assistance* with the ΩMEGA system, providing an automated uniform solution for these two mediation problems.

1.1 Mathematical Authoring

The general motivation for this thesis is the current discrepancy between a mathematical document prepared for publishing and its formalization for the verification by a proof assistance system. On the one hand the author is writing her document in a standard text-editor using a domain-specific subset of natural language interleaved with formulas in a well-defined notation, on the other hand proof assistance systems require a formalization written in a formal (logic based) specification language.



There is a huge gap between these different levels of formality. Table 1 shows an example from a project on formalizing the Fundamental Theorem of Algebra [Geuvers *et al*, 2000]. On the left, the published document is shown, and on the right, its formalization for the proof assistance system Coq [Coquand & Huet, 1988].

... **Proposition 7.1 (Kneser Lemma).** For every $n \in \mathbb{N}$, $n \geq 2$ there exists a $q \in \mathbb{R}$, $0 < q < 1$ such that for every polynomial over $\mathbb{C}$ with leading coefficient 1 $f(x) = x^n + b_{n-1}x^{n-1} + \cdots + b_1 x + b_0$ one has $\forall c > \lvert b_0 \rvert . \exists z \in \mathbb{C} . \left(\lvert z \rvert < c^{\frac{1}{n}} \wedge \lvert f(z) \rvert < qc \right)$	... Lemma Kneser : (n:nat)(lt (0) n) -> (EX q | (Zero [:<] q) /\ (q [:<] One) /\ (p:(cpoly CC))(Monic n p) -> (c:IR)((AbsCC (p!Zero)) [:<] c) -> (EX z | ((AbsCC z)[^]n [:<] c) /\ ((AbsCC (p!z)) [:<] q[*]c))).

Table 1. The Kneser Lemma and its formalization for Coq (reproduced from [Geuvers et al*, 2000])*

If we take a closer look at this example, we notice that the difference goes beyond a simple syntax translation. Indeed, the formalization of the polynomial $f(x)$ as a variable `p` of type `(cpoly CC)` hides the structure of the polynomial which is shown in the published document. The reason is that it is not clear how to directly represent the part "..." of the polynomial in the formal specification language of a proof assistance system. Therefore, such parts often have a different formalization.

It is not the goal of this thesis to bridge this gap completely but to develop the required methods to automatically mediate as much as possible between the authoring for the formalization of a mathematical document and the authoring for publishing. This requires the integration of a proof assistance system with a scientific text-editor. Since proof assistance systems operate in general either in batch processing style or by command line interaction, we need to develop a new interface paradigm for proof assistance systems that reacts on arbitrary changes in a document. Furthermore, this style of interaction needs to be adapted to the entire architecture of the proof assistance system in order to support a flexible, independent and evolutionary development of system components. Thus, we first need an efficient method to compute the changes between documents.

Semantic Change Computation. The document in the text-editor, its formalized counterpart, interface documents of the components of the proof assistance systems, all these documents represent various kinds of (semi-)structured knowledge. To compute the changes of two documents we need to know how the *similarity* and *semantic equality* of elements can be determined. For example, there are elements for which the order of their children is not relevant. Furthermore, elements in a document represent objects in the components of the proof assistance system. For example, changing an axiom may result in the obligation to verify the proofs of lots of theorems again. We need a method to take these hidden *edit weights* into account when computing the optimal semantic change between documents. Finally, since it is not reasonable to develop methods for reacting on arbitrary small knowledge changes, the change computation needs to be limited to specific document levels.

Invertible Grammar Formalism. Different kinds of knowledge are represented by the components in their interface documents in different formats, in particular the document in the text-editor and its formalized counterpart for the proof assistance system. Thus, we need a robust method to translate between these two different knowledge formats in both directions. Hence in general, we have to develop a new formalism for transformation grammars with a special focus on its automatic inversion. Thereby, the similarity of documents should be natively taken into account to reduce the amount of grammar rules, for example by covering all ordering variants with one single grammar rule pattern. Furthermore, knowledge is sometimes split or reordered in an interface document. We need means to embed the knowledge about information flow into the grammar rules to effectively prune alternatives as early as possible during document translation.

Having developed these key components of a *mediation module*, we need to develop a new *Change-Oriented Architecture* for the proof assistance system ΩMEGA.

Mathematical Authoring Assistance. We will illustrate the usage of the Change-Oriented Architecture with a course scenario. Therefore, we will develop methods to assist the authoring of lecture notes, exercises and their solutions. For example, we need an incremental bidirectional transformation pipeline between the document in the text-editor, written in a controlled mathematical language combined with semantically annotated full natural language, and the formalization for the proof assistance system ΩMEGA with its proof obligations.

1.2 Overview of this Thesis

The thesis is organized into four parts: the first and introductory part ends with Chapter 2 by recapitulating the state of the art. The main contributions of the thesis are presented in the Parts II and III.

Part II presents the formal theory for the *Change-Oriented Architecture*. In Chapter 3 we introduce the document model, the change model and the general principles of this architecture. Chapter 4 introduces specifications for the similarity and semantic equality of documents and the edit costs and granularity of changes. In Chapter 5 we develop a solution to the weighted semantic tree-to-tree correction problem by reducing it to a search problem. Chapter 6 presents an invertible grammar formalism for the automated bidirectional transformation of documents.

Part III presents the application of the *Change-Oriented Architecture* to *Mathematical Authoring Assistance* in a course scenario with the ΩMEGA system. In Chapter 7 we report about the results of an exploratory study to analyze the requirements of mathematical authoring assistance. Chapter 8 presents a method for exploiting the notation defined by the author in lecture notes in order to automate the formalization and rendering of formulas. The process of creating proof obligations from exercise solutions written in controlled mathematical language is described in Chapter 9, together with the document-centric integration of ΩMEGA with the text-editors TEXMACS and MS WORD.

In Part IV we summarize the results of this work in Chapter 10 and we present an outlook for future research in Chapter 11.

2 State of the Art

The first computer generated proof of a mathematical theorem was generated in 1954 by a program of Martin Davis, which implemented a subset of first order predicate calculus, called Presburger Arithmetic. The Dartmouth Conference in 1956, widely considered as the birth place of artificial intelligence, saw the first automated theorem proving systems and these spawned the general vision that one day, all formal documents and routine mathematical theorems could in fact be shown or at least checked automatically by a machine. After the early enthusiasm had declined, the actual difficulties involved in automating everyday problems have been more and more realized. In the following, we will present the most prominent approaches in the history of formalized mathematics, starting with the formalization of foundations, the resolution principle and the first proof checkers, going to proof assistance systems and libraries of formalized mathematics, ending with the integration of proof assistance systems with text-editors to form mathematical assistance systems.

2.1 Historical Overview

The dream to mechanize formal reasoning dates back at least to Gottfried Wilhelm Leibniz in the 18th century. He formulated the touching vision that two philosophers engaged in a dispute could simply code their arguments into an appropriate formalism and then calculate (*Calculemus!*) who is right. With Frege's Begriffsschrift this dream came closer to reality and modern mathematical logic was born by the end of the 19th century. Other important milestones in the formalization of mathematics were Hilbert's programme and his "*Grundlagen der Mathematik*" ([Hilbert & Bernays, 1934], [Hilbert & Bernays, 1939], [Gabbay *et al*, 2010]), and Russell and Whiteheads formalization in "*Principia Mathematica*" [Whitehead & Russell, 1910-1913] and more recently the 20th century Bourbakism.

Since the early systems in 1956 there have been substantial improvements in the area of automated deduction with respect to system performance and usability, most of them using the resolution principle [Robinson, 1965]. The early enthusiasm of the sixties also led to the pioneering development of a proof checker, the AUTOMATH system [de Bruijn, 1970] by N. G. de Bruijn. The system checks the logical correctness of mathematical documents that are written in a formal language called *mathematical vernacular* [de Bruijn, 1994].

Since the beginning of the Seventies the MIZAR project [Rudnicki & Trybulec, 1999] addresses the demand for a more readable input language. The project supports mathematicians in publishing their work and has grown to one of the largest libraries of formalized mathematics. At the heart of the system there is a formal language whose logical structure is based on Jaskowski-style natural deduction [Jaskowski, 1934] (in contrast to Gentzen-style natural deduction [Gentzen, 1934]). This formal language allows to verify the logical consistency of the content of an article as well as to verify cross references to other articles written in the same language. The relationship of the MIZAR language to the AUTOMATH language is comparable to the relationship of a high-level programming language to an assembly language.

Besides AUTOMATH and MIZAR, further formal languages have been developed in parallel for the proof assistants COQ [Coquand & Huet, 1988], ISABELLE [Wenzel *et al*, 2008], MATITA [Asperti *et al*, 2006], ΩMEGA [Autexier *et al*, 2009], and some others including the *Logical Frameworks* [Pfenning, 1999]. These support the declaration of the notation for symbols as prefix, infix, postfix and mixfix, and further definitions of complex notation by rewriting mechanisms that are called *abstraction* and *rendering* parsers [Padovani & Zacchiroli, 2006]. Almost all proof assistance systems reject ambiguities that cannot be resolved by explicit precedence or associativity declarations, except the MATITA system which uses sophisticated disambiguation heuristics [Sacerdoti Coen & Zacchiroli, 2008] in combination with user interaction.

All mentioned projects target a representation of mathematical content that supports both machine processing as well as human authoring. The major problem of these approaches is that they do not sufficiently succeed in combining the widely diverging representational requirements. As a consequence, the proof assistance systems were extended by techniques that generate from a proof in machine-oriented representation an output proof which gets close to natural language for enhanced readability. The first reconstructive approach was presumably presented by Xiarong Huang [Huang, 1994]. The system PCOQ [Amerkad *et al*, 2001] for example uses a schematic approach to represent its output in quasi-natural language. The systems NUPRL [Holland-Minkley *et al*, 1999], CLAM [Alexoudi *et al*, 2004] and P.REX [Fiedler, 2001] go further and use natural language processing techniques to generate true natural language output. Despite the almost textbook quality of the output, the legibility of the input remained archaic.

A new representative of distributed systems for the publication of machine checked mathematics is LOGIWEB [Grue, 2007]. It allows the authoring of articles in a customizable language with the usual LaTeX workflow.

Besides the separation of informal and formal documents, one can also go for the mixed approach of the THEOREMA system [Buchberger *et al*, 1997] that allows separated informal and formal parts intertwined in the user created document. The informal parts can be authored without any restrictions, but these parts cannot be used within the formal parts, whereas the formal parts have to be written in the input language of the computer algebra system MATHEMATICA. The THEOREMA system has been developed as an extension of MATHEMATICA, thus it inherits all document structuring possibilities. Automatically generated proofs are translated into a graphically enriched quasi-natural language by a schema based approach that uses arbitrary new *logicographic symbols* [Nakagawa & Buchberger, 2001] for mathematical functions and predicates. However, the consistency between informal and formal parts has to be maintained manually by the author.

Many proof assistance systems use the generic PROOFGENERAL [Aspinall, 2000] system as a user interface. PROOFGENERAL allows the user to edit the central document in the native proof assistant format in an ASCII editing environment like EMACS. From there the document can be evaluated by various tools, such as a proof assistance system which checks whether the document contains valid proofs. A custom interaction protocol called PGIP [Aspinall *et al*, 2005] is used that provides a state model for the proof assistant and locks the fragment of the document that is being processed by the proof assistant. There are also alternative protocols proposed like IAPP [Gast, 2008] that transfer the ownership of proof commands between the interface and the proof assistant. Furthermore, the protocol of TMEGG [Mamane & Geuvers, 2006] allows for reordering the commands sent to the prover. All these variants have in common that they operate line based with a single focus point of interaction and that the communication protocol use the *locking technique*, where a lock prevents parts of a document from being modified.

Since the input language of all presented systems is still far away from what is actually written in textbooks, the problem is addressed recently by approaches that reverse the usual development perspective: Starting from the traditional authoring application of the end-user and her plain document, the research problem is how to generate the input representation for the formal system. The two driving forces of this new perspective are semantic annotation languages and controlled mathematical languages.

2.2 Semantic Annotation Languages

A formal language for mathematics called *Weak Type Theory* (WTT) is introduced in [Kamareddine & Nederpelt, 2004] as an intermediary between the natural language of the mathematician and the formal language of the logician. Grown out of de Bruijn's *Mathematical Vernacular*, WTT extends it by assigning a unique atomic weak type to each text element and by introducing a meta-theory for describing aspects of WTT documents. Based on WTT the MATHLANG language [Kamareddine *et al*, 2004] has been developed which is a semantic annotation language that can be used to mark different aspects of a mathematical document from the overall theory structure down to single variables in formulas. After the process of manually annotating the document the benefits are for example integrity checks of rhetorical aspects.

Another work is the natural language analysis of mathematical proofs by Claus Zinn [Zinn, 2004]. The focus of his work is the generation and automatic verification of a formal proof representation from the typical informal representation (in natural language) of a standard mathematical textbook. In his approach, the user also has to annotate the document by hand before the automatic analysis starts, in order to make the logical structure explicit. His system was applied to a corpus of proofs from a mathematical textbook for first-year students.

The long-term goal of the ΩMEGA project [Siekmann & Autexier, 2007] is the development of a large, integrated assistance system supporting different mathematical tasks and a wide range of typical research, publication and knowledge management activities. The most important achievements in this project have been: (1) the development of the concepts of knowledge-based proof planning [Melis & Siekmann, 1999] and proof planning with multiple strategies [Melis & Meier, 2000], (2) a mathematical software bus MATHWEB [Franke & Kohlhase, 1999] for distributed automated theorem proving, later extended to semantic reasoning web services [Zimmer & Autexier, 2006], (3) a three dimensional data structure for proof plans ([Cheikhrouhou & Sorge, 2000], [Autexier *et al*, 2005]), and a system for the management of changes in structured theories [Hutter, 2000], (4) the natural language explanation of proofs [Fiedler, 2001] and the proof of the irrationality of $\sqrt{2}$ [Siekmann *et al*, 2002], (5) the CORE calculus [Autexier, 2005] with deep inference and an assertion-level proof representation with under-specification [Autexier *et al*, 2004], (6) the integration with the scientific text-editor TEXMACS [van der Hoeven, 2001], called PLATΩ [Wagner *et al*, 2006], using a semantic annotation language based on a subset of concepts found in OMDOC [Kohlhase, 2000] and OPENMATH [Davenport, 2000], combined with a new annotation language for proof steps.

2.3 Controlled Mathematical Language

Traditionally, controlled natural languages fall into two major categories: those that improve readability for human readers, and those that enable reliable automatic semantic analysis of the language. The first type of languages, for example *Caterpillar Technical English* CTE [Kamprath *et al*, 1998] or IBM*'s EasyEnglish* [Bernth, 1997], is used in industry to increase the quality of technical documentations. The second type of language has a formal logic base with a formal syntax and semantics and can be mapped onto an existing formal language. Examples are *Attempto Controlled English* ACE [Fuchs *et al*, 2008] or *Boeing's Computer Processable Language* CPL [Clark *et al*, 2005]. In the mathematical domain, the following two controlled mathematical languages for the authoring of documents are in use.

The *System for Automated Deduction* SAD [Lyaletski *et al*, 2006] has been developed for the automated processing of mathematical texts in a batch-style workflow comparable to MIZAR. The SAD system uses an extremely tight connection between linguistic processing and logical inference which is unique in the field. Although the examples of mathematical proof come close to textbook quality, the language is not used by the community mainly because it is not reusable due to its tight connection with the SAD system.

The most recent approach is the NAPROCHE project, which is an acronym for NATURAL LANGUAGE PROOF CHECKING, a joint initiative of computational linguists and mathematicians [Koepke & Schröder, 2003]. This approach analyzes the interplay between the natural mathematical language as it is used in mathematical textbooks, and formal mathematics from the mathematician's point of view. At the heart of this approach is the NAPROCHE language, a controlled natural language designed by mathematicians for mathematicians. The linguistic aspect of the NAPROCHE system is based on an extension of the *Discourse Representation Theory* [Kamp & Reyle, 1993].

All systems that support a subset of natural language have completely different input and output representations. A solution proposed for unifying the requirements of parsing and rendering are invertible grammar formalisms with various linguistic features. A prominent representative is the *grammatical framework* GF [Ranta, 2004], a λ-calculus based formalism to define grammars consisting of an abstract and a concrete syntax. However, our experience with the development of the ΩMEGA system is that an all-encompassing representation of all kinds of knowledge, which would be needed for the abstract syntax of GF, is a rather complex solution that hampers the natural evolution of the system.

2.4 Discussion

The two modern driving forces for improving the authoring of formalizable mathematics are *semantic annotation languages* and *controlled mathematical languages*. The advantage of semantic annotations is the great flexibility of using full natural language to write the document, the drawback is the tedious process of adding the annotations to the document. The advantage of a controlled mathematical language is that there is no authoring overhead like providing annotations, the drawback is that this language is restricted. Since there is no clear winner, we develop in this thesis a framework that supports both approaches in parallel and in combination. From a practical point of view, the combination of semantic annotations with a controlled mathematical language increases the robustness of the system.

The workflow of existing systems can be roughly classified into batch processing systems and interactive command-line systems. The result of integrating these systems with a scientific WYSIWYG text-editor is either a push-button verification service or an interactive script-style dialogue. In the latter case, the result of a command sent to the proof assistance system is displayed in the text-editor. The approach we present in this thesis goes far beyond that. We consider the document in the text-editor as both the input and output document of the proof assistance system. To the best of our knowledge, no other document-centric approach has been presented that is fully based on changes of the document in the text-editor.

With respect to the architecture of a knowledge-based system, one usually chooses a Blackboard architecture if the kinds of knowledge that need to be dealt with are unknown or if the best processing order of the knowledge sources or the tasks of the knowledge sources are unknown. When there is a stable solution, the Blackboard architecture is replaced by a concrete pipeline or broker architecture, and the interfaces of the components have to be carved in stone. With the *Change-Oriented Architecture* we propose an intermediate step in this evolution of the architecture. When the processing order and the tasks of the specialist knowledge sources are roughly known, they can be connected by *mediation modules*. Instead of defining concrete interface methods to manipulate knowledge at a specific level, a mediation module translates between the interface documents of two components. The components interact by changing their interface document and reacting on changes applied by other components. Thereby, the changes are computed by the mediation module according to the specification given by a component. This allows for the independent improvement of the components until their interface methods are stable.

II
Change-Oriented Architecture

In this part, we describe the concepts and methods of the *Change-Oriented Architecture*. The principle idea of this architecture is to use *documents as interfaces* of the components and to communicate their changes instead of calling specific interface methods. In this setting, the two fundamental problems that we have to solve are the following.

1) How can we compute the optimal changes between two documents?
2) How can we automate the bidirectional translation between two documents?

Before we start to address these problems, we have to introduce the basic notions and notations. Therefore, we introduce in Chapter 3 the document model and the supported change operations. We will analyze the problem of change management for Service-Oriented Architectures and Blackboard Architectures. As a solution to this problem we will introduce the Change-Oriented Architecture as an extension to the Service-Oriented Architecture. We give an overview of the architecture model and discuss the application of different design patterns.

In order to address the first problem of computing optimal changes, we need to define the notion of optimality for changes. The different aspects that have to be taken into account are discussed in Chapter 4. On the one hand, the content that is represented by the document has a specific semantics. Important questions to ask are: How do we identify corresponding elements in a document? Is the order of particular elements relevant or not? On the other hand, the changes are computed to be handled by a specific component. What are the implicit costs of changing particular elements? What is the granularity of changes the component is able to deal with? In Chapter 4, we introduce specifications to declaratively answer these questions and thus define a component- and document-specific notion of optimal changes.

The problem of computing the optimal changes between two documents is then reduced in Chapter 5 to a shortest-path-to-goal problem. We present an algorithm that is based on Dijkstra's shortest path algorithm but dynamically expands the search graph as the algorithm traverses the graph. The correctness of this algorithm is proved, and the complexity is analyzed and compared to the state-of-the-art algorithm.

Finally, we address in Chapter 6 the problem of automating the bidirectional translation between two documents. We present a pattern-based formalism for transformation grammars whose rules are designed to be automatically invertible. The formalism supports rule attributes as known from attribute grammars, with additional integration of unification and type checking. Furthermore, we develop a technique for the incremental processing of a document. This is important because we want to keep as much of the document unchanged as possible.

3 Foundations

The paradigm of the *Change-Oriented Architecture* is to use *documents as interfaces* of the components and to communicate their changes. In the following we are going to set the stage for this architecture by first introducing the notions and notations for *sets*, *relations*, *functions*, *labeled trees* and other basic concepts. For example, a tree where every node and edge has a label and the layers of the tree and every subtree are totally ordered will be called a *labeled tree*.

The most important concept, to be defined first, is the document model. If we look at the content exchange formats used by web services or the document formats used by modern text-editors, we see a representation that is essentially an instance of *labeled trees*. In this context, XML [W3C, 2008] is notably the most widely used representation format. Therefore, we will illustrate our document model by describing the instantiation of *labeled trees* for the XML format.

After that, we will define the operations that can be used to change a document. The model for the *edit operations* is a fundamental design decision. First, we will define the set of *edit operations* that are supported: *insert*, *delete*, *replace* and *append*. This decision requires justification. Why not support the *relabel* operation? We would benefit from shorter change descriptions, but since subtrees in an interface document generally correspond to complex objects in a component, this operation is not valid. Why not support the *move* operation? Detecting a move operation is not always a decidable problem. In case of a false positive, this change may propagate through the system and result in a non-intended state. Second, the semantics of the *edit operations* have to be defined. This decision considers mainly the *delete* operation. Do we delete the whole selected subtree or do we replace the subtree by its children? In conformance with the vast majority of systems for change computation we decide to delete whole subtrees.

Finally, we will give an overview of the architecture model of the *Change-Oriented Architecture* and discuss the application of different design patterns to the mediation module which forms the interface between two components. In particular, we will compare this architecture model to the Blackboard architecture and the Service-Oriented Architecture. Since a component may act as a *façade* to a collection of components, we will discuss how changes to the interface document of this *façade* component can be treated as a transaction. All mediation modules are transaction safe, thus they guarantee that every submitted set of changes for the interface document is treated atomically, consistently, isolated and durable. The information flow will be illustrated on a concrete example.

3.1 Notions and Notation

We introduce the basic notions and notations for *sets*, *relations*, *functions*, *orders*, *sequences*, *graphs*, *trees*, *labeled trees* and related concepts in the following.

Definition 3.1.1 (Sets): A *set* is a collection of objects, called *elements*, in which no order exists and where every element occurs only once. The *empty set*, that is, the set which contains no elements, is denoted by $\emptyset$. A set B is called a *subset* of a set A, and conversely A is called a *superset* of B, written $B \subseteq A$, if every element of B is also an element of A. We say that B is a *proper subset* of A, denoted by $B \subsetneq A$, if $B \subseteq A$ and $B \neq A$.

The *intersection* of the sets A and B, denoted by $A \cap B$, is the set of elements which belong to both A and B. For the sets $A_1, \dots, A_n$ we denote their intersection as $\bigcap_{i=1}^{n} A_i$. If the intersection of two sets A and B is the empty set, the sets are called *disjoint*. The sets $A_1, \dots, A_n$ are called *pairwise disjoint* if $\forall i, k. \big((i \neq k) \Rightarrow (A_i \cap A_k = \emptyset)\big)$.

The *union* of the sets A and B, denoted by $A \cup B$, is the set that contains the elements of both sets A and B. The union of the sets $A_1, \dots, A_n$ is the set denoted by $\bigcup_{i=1}^{n} A_i$. The union of two disjoint sets A and B is denoted by $A \uplus B$.

The *difference* $A \backslash B$ of the sets A and B is the set of those elements of A which do not belong to B. For a finite set A we denote its *cardinality* by $|A|$, that is the (cardinal) number of elements in the set A. We denote a finite set containing the elements $x_1, \dots, x_n$ by $\{x_1, \dots, x_n\}$. The notation $\{x \in A | P(x)\}$ describes the set of all elements of the set A for which the property P holds.

The *powerset* of a set A is the set of all subsets of A defined by $\mathcal{P}(A) \coloneqq \{B | B \subseteq A\}$. The *cartesian product* of two sets A and B is the set of all ordered pairs of one element of A and one element of B, defined by $A \times B \coloneqq \{(x, y) | x \in A \wedge y \in B\}$.

Definition 3.1.2 (Relations and Functions): Let A and B be two sets. A *relation* R between A and B is a set of ordered pairs (a, b) such that $a \in A$ and $b \in B$. For $(a, b) \in R$ we also write aRb. A *(total) function* f from A to B is a relation between A and B such that for each $a \in A$ there is one and only one associated $b \in B$. For the key-value-pair $(a, b) \in f$ we use the notation $f(a) = b$.

The set A is called the *domain* of f, denoted by $dom(f)$, and the set B is called the *codomain*, denoted by $codom(f)$. The set $\{b \in B | \exists (a, b) \in f\}$ is called the *range* of f, denoted by $ran(f)$. Furthermore, the function f is called *injective* if it holds that $\forall a, b \in A. \Big(\big(f(a) = f(b)\big) \Rightarrow (a = b)\Big)$. It is called *surjective* if $\forall b \in B. \exists a \in A. f(a) = b$. The function f is *bijective* if it is both injective and surjective. If there is a bijective function from A to B, then it holds that $|A| = |B|$.

The *set of all bijective functions* from A to B is denoted by $\mathfrak{F}_{A\leftrightarrow B}^{\tau}$. The notation $\mathfrak{F}_{A\leftrightarrow B}^{\tau}$ shows an additional context parameter $\tau = (\bot, \emptyset)$. We use the first parameter $\bot$ to indicate the unordered case, in contrast to the ordered case of bijective functions between sequences. The reason for the second parameter $\emptyset$ is that we also consider the case of sets A and B with different cardinalities $|A| \leq |B|$. A *multi-function* f from A to B with $C \subseteq A$ is a relation between A and B such that for each $a \in A \backslash C$ there is one and only one associated $b \in B$, and for each $a \in C$ there is at least one associated $b \in B$. The set of all bijective multi-functions from A to B with $C \subseteq A$ is denoted by the context $\tau' = (\bot, C)$. Note that we omit the context parameter whenever the intended meaning can be inferred.

A (partial) *mapping* from A to B is a bijective function from $A' \subseteq A$ to $B' \subseteq B$ with $|A'| = |B'|$. Thus, the semantics of mappings are those of partial bijective functions between A and B. The *set of all mappings* from A to B is denoted by $\mathfrak{M}_{A\leftrightarrow B}^{\tau}$ with $\tau = (\bot, \emptyset)$. The parameter $\bot$ indicates again the unordered case, and the parameter $\emptyset$ that we have sets with equal cardinalities. A *multi-mapping* from A to B with $C \subseteq A$ is a bijective multi-function from $A' \subseteq A$ to $B' \subseteq B$ with $|A'| \leq |B'|$. The set of all bijective multi-mappings from A to B with $C \subseteq A$ is denoted by the context $\tau' = (\bot, C)$. The parameter $\bot$ in the context $\tau' = (\bot, C)$ indicates that the elements of A and B are not ordered. Furthermore, in the case that the set A contains less elements than the set B, the parameter C specifies the subset of elements of A that have at least one assigned element of B. Note that in the special case of $C = \emptyset$ both sets A and B must have equal cardinalities.

The *inverse function* f^{-1} from B to A of a bijective function f from A to B is defined by $f^{-1} \coloneqq \{(y, x) | (x, y) \in f\}$. The *composition* $f \circ g$ of a function f from A to B and a function g from B to D is a function h from A to D defined by $h \coloneqq \{(x, z) | (x, y) \in f \wedge (y, z) \in g\}$.

Definition 3.1.3 (Orders): A *binary relation R on a set A* is a set of ordered pairs of elements from A. A binary relation $\leq$ on a set A is called a *partial order* on A if it is reflexive $\big(\forall a \in A.\, (a \leq a)\big)$, anti-symmetric $\Big(\forall a, b \in A.\, \big((a \leq b \wedge b \leq a) \Rightarrow (a = b)\big)\Big)$ and transitive $\Big(\forall a, b, c \in A.\, \big((a \leq b \wedge b \leq c) \Rightarrow (a \leq c)\big)\Big)$.

A binary relation R on a set A is a *total order on A* if it holds that $\forall a, b \in A.\, (aRb \vee bRa \vee a = b)$. A set A is totally ordered by a binary relation R if $R \cap (A \times A)$ is a total order on the set A. A binary relation R on a set A is called an *equivalence relation* on A if it is reflexive $\big(\forall a \in A.\, (aRa)\big)$, transitive $\Big(\forall a, b, c \in A.\, \big((aRb \wedge bRc) \Rightarrow (aRc)\big)\Big)$ and symmetric $\big(\forall a, b \in A.\, (aRb \Rightarrow bRa)\big)$.

Definition 3.1.4 (Sequences): A *sequence* with elements of a set S is a total function from $\{1, \dots, n\}$ to S for some $n \in \mathbb{N}, n \geq 0$. Hence the order of a sequence matters and the same element can appear several times at different positions in a sequence. A finite sequence containing the key-value-pairs $(1, s_1), \dots, (n, s_n)$ is denoted by $[s_1, \dots, s_n]$. We say that s_i *precedes* s_m respectively s_m *follows after* s_i in a sequence $S = [s_1, \dots, s_n]$, if $i < m$. Furthermore, we write $(i, s) \in S$ or $S[i] = s$ to denote that s is the i-th element in the sequence S. We omit the position if it can be derived from the context. For a finite sequence A we denote its *cardinality* by $|A|$, that is its (cardinal) number of elements.

A sequence S' with $|S'| = n'$ is called a subsequence of S with $|S| = n$, denoted by $S' \sqsubseteq S$, if there is a total function $f: \{1, \dots, n'\} \to \{1, \dots, n\}$ from the positions in S' to the positions in S such that $\forall i \in \{1, \dots, n'\}. \big((i, s) \in S' \Rightarrow (f(i), s) \in S\big)$ and $\forall i, j \in \{1, \dots, n'\}. \big(i < j \Rightarrow f(i) < f(j)\big)$. A subsequence S' of S is called a contiguous subsequence of S, denoted by $S' \Subset S$, if $\forall i \in \{1, \dots, n' - 1\}. f(i + 1) = f(i) + 1$ holds.

The notation $[x \in S | P(x)]$ describes a subsequence of S whose elements have the property P. The concatenation of two sequences $S = [s_1, \dots, s_k]$ and $S' = [s_{k+1}, \dots, s_n]$ is defined as $S :: S' \coloneqq [s_1, \dots, s_n]$. The empty sequence, that is the sequence which contains no elements, is denoted by $[\,]$. An element s can be added in front of or at the end of a sequence S using the operator $\star$. We have: $[s_1, \dots, s_k] = s_1 \star [s_2, \dots, s_k] = [s_1, \dots, s_{k-1}] \star s_k$. The *powerset* of a sequence A is the set of all subsequences of A defined by $\mathcal{P}(A) \coloneqq \{B | B \sqsubseteq A\}$.

Let A and B be two sequences. The *set of all bijective functions* from A to B, that preserve the order of A and B, is denoted by $\mathfrak{F}^{\tau}_{A \leftrightarrow B}$ with $\tau = (\top, \emptyset)$, hence it holds that $\forall f \in \mathfrak{F}^{\tau}_{A \leftrightarrow B}. \forall \big((i, a), (j, b)\big) \in f. i = j$. Clearly, this set always contains exactly one function. The first context parameter $\top$ indicates that the elements of A and B are ordered. Similarly to the unordered case with sets, we define the *set of all bijective multi-functions* from A to B with $C \subseteq A$, that preserve the order of A and B, and we denote this set by the context $\tau' = (\top, C)$. Note that the special case $C = \emptyset$ implies again $|A| = |B|$.

A (partial) *mapping* from A to B is a bijective function from $A' \sqsubseteq A$ to $B' \sqsubseteq B$. Thus, the semantics of mappings are those of partial bijective functions between A and B. The *set of all mappings* from A to B is denoted by $\mathfrak{M}^{\tau}_{A \leftrightarrow B}$ with $\tau = (\top, \emptyset)$ and preserves the order of A and B, hence $\forall m \in \mathfrak{M}^{\tau}_{A \leftrightarrow B}. m \in \mathfrak{F}^{\tau}_{dom(f) \leftrightarrow ran(f)}$. The parameter $\top$ of $\tau = (\top, \emptyset)$ indicates that the elements of A and B are ordered. A *multi-mapping* from A to B with $C \subseteq A$ is a bijective multi-function from $A' \sqsubseteq A$ to $B' \sqsubseteq B$ with $C \sqsubseteq A$. The *set of all multi-mappings* from A to B is $\mathfrak{M}^{\tau'}_{A \leftrightarrow B}$ with $\tau' = (\top, C)$ and preserves the order of A and B. Furthermore, in the case that the set A contains less elements than the set B, the parameter C specifies the elements of A that have at least one assigned element of B.

Definition 3.1.5 (Graphs): A *directed graph* is a pair $G = (V, A)$, where V is a finite set of nodes and A is a finite set of ordered pairs of elements from V. The elements of V are called the *nodes* or *vertices* of G, the elements of A are its *edges* or *arcs*. We assume that A contains no edges of the form (v, v) where $v \in V$. The (cardinal) number of nodes in G is the *size* of G and denoted by $|G|$. An edge $e = (u, v)$ is said to *connect* the nodes u and v. We also say that e is *incoming* for v and *outgoing* for u.

A (directed) *walk* in G is a sequence $[v_1, \dots, v_k]$ of nodes of G such that G contains edges (v_i, v_{i+1}) for all $i = 1, \dots, k-1$. The length of the walk $[v_1, \dots, v_k]$ is k. A walk is called a (directed) *path* if all $v_1, \dots, v_k$ are pairwise distinct. It is called a (directed) *circuit* if $v_1 = v_k$. A node v is called reachable from a node u if there is a path from u to v. The set of nodes reachable from a node v is denoted by $reach(v)$.

An *(undirected) graph* is a pair $U = (V, E)$, where V is a finite set and E is a finite set of sets each containing two elements from V. The terminology is defined similarly to the directed graph. Additionally, two nodes u and v are called *adjacent* if E contains an edge $\{u, v\}$. Two edges e_1 and e_2 are called *incident* if they have a common node. A graph $U = (V, E)$ is called *connected* if for all nodes $u, v \in V$ there is a path connecting u and v. For each directed graph $G = (V, A)$ there is an *underlying undirected graph* $U = (V, E)$ which is obtained by ignoring the order of the edges in A, given by $E := \{\{v, w\} | (v, w) \in A\}$.

Definition 3.1.6 (Trees): A directed graph is called a (directed rooted) *tree* if the following conditions hold:

1) it is connected and contains no circuits,
2) it has exactly one *root*, which is a node without incoming edges,
3) every non-root node is reachable from the root.

The properties 2 and 3 imply a natural orientation of the edges, directed from the root towards the *leaves*, which are nodes without outgoing edges. If r is the root of a tree T, denoted by $r = root(T)$, we say that T is a *tree rooted at* r. Let $T = (V, A)$ be a tree rooted at r. All nodes $v \in V$ with $v \neq r$ have exactly one incoming edge. If $(u, v) \in A$, then u is called a *parent* of v and v is a *child* of u. Thus, each node $v \in V$ with $v \neq root(T)$ has exactly one parent.

For a node v we denote its parent by $parent(v)$ and the set of its children by $children(v)$. Nodes without children are the *leaves* of a tree. A node u is called an *ancestor* of a node v, if there is a directed path from u to v and $u \neq v$. In this case, v is called a *descendant* of u. Two nodes u and v are called *siblings* if they have the same parent. The *depth of the node* v in a tree is the length of the path from the root of the tree to the node v. The *depth of the tree* is the maximum of the depth of all nodes in the tree.

A tree T_2 is a *direct subtree* of a tree T_1 if $root(T_2)$ is a child of $root(T_1)$. The *set of all direct subtrees* of a tree T is denoted by $\mathcal{C}(T)$. A tree T_2 is a *subtree* of a tree T_1 if there is a directed path from $root(T_1)$ to $root(T_2)$ in T_1. The *set of all subtrees* of a tree T including the tree T itself is denoted by $\mathcal{S}(T)$. A tree T_2 is the *parent tree* of a tree T_1 if $T_1 \in \mathcal{C}(T_2)$. A tree T_2 is an *ancestor tree* of a tree T_1 if $T_1 \in \mathcal{S}(T_2)$ and $T_1 \neq T_2$. The *set of all ancestor trees* of a tree T_1 in a tree T is denoted by $\mathcal{A}_T(T_1)$.

Definition 3.1.7 (Labeled Trees): Let $T = (V, A)$ be a tree. We define the *node labeling function* L_V to be a total function from the finite set of nodes V to the finite universal set of node labels $\mathcal{L}_V$. Additionally, we define the *edge labeling function* L_A to be a total function from the finite set of edges A to the finite universal set of edge labels $\mathcal{L}_A$. The finite set of labels $\mathcal{L}$ is a superset of both sets $\mathcal{L}_V$ and $\mathcal{L}_A$ with $\mathcal{L}_V \cap \mathcal{L}_A = \emptyset$. Thus, a label is assigned to every node and every edge in the tree T. The label of the root node of a tree T is called the *tree label* of T and it is denoted by $L(T)$.

The *direct subtree layer* $\mathcal{C}_l(T)$ of a labeled tree T and any label $l \in \mathcal{L}_A$ is the subset of all direct subtrees $\mathcal{C}(T)$ that contains all direct subtrees whose root node is connected to the root node of T by an edge with the label l. Thus, the set $\mathcal{C}(T)$ of a given tree T is partitioned by the label of the connecting edges. Let the binary relation R on V be a partial order on V that totally orders the root nodes of the labeled trees in $\mathcal{C}_l(t)$ for all $t \in \mathcal{S}(T)$ and all $l \in \mathcal{L}_A$. Then the (*direct subtree) layer* of a labeled tree t and a label l can be written as a finite sequence $\mathcal{C}_l(t) = [T_1, \dots, T_k]$ ordered by R where T_1 is the *left-most child* and T_k the *right-most child* of the tree t in this layer. We say that the layer $\mathcal{C}_l(t)$ is labeled by l. A layer $\mathcal{C}_l(t)$ can be *restricted* by a set of node labels $K \subseteq \mathcal{L}_V$ as defined by $\mathcal{C}_l(t)|_K := [t' \in \mathcal{C}_l(t) | L(t') \in K]$. Let $Y = [T_1, \dots, T_m]$ be a (partial) layer of a tree, the *size* of Y is defined by $|Y| := |T_1| + \cdots + |T_m|$. Furthermore, we extend the definition of the set of all subtrees to (partial) layers of a tree by $\mathcal{S}(Y) := \mathcal{S}(T_1) \cup \dots \cup \mathcal{S}(T_m)$.

A tree where every node and edge has a label and the layers of the tree and every subtree are totally ordered is called a *labeled tree* $D = (V, A, L_V, L_A, R)$. Two labeled trees D_1 and D_2 are called *equally labeled*, denoted by $D_1 \approx D_2$, if and only if $L(D_1) = L(D_2)$. Let D be a labeled tree, we define the sibling functions S_L and S_R as total functions from the set of trees $\mathcal{S}(D)$ to the powerset of the set of trees $\mathcal{S}(D)$ as follows: Let $\mathcal{C}_l(t) = [D_1, \dots, D_k]$ be the layer of a labeled tree $t \in \mathcal{S}(D)$ and a label $l \in \mathcal{L}_A$. We define $S_L(D_i) := [D_1, \dots, D_{i-1}]$ to return the *left sibling labeled trees* of D_i in D, and $S_R(D_i) := [D_{i+1}, \dots, D_k]$ to return its *right sibling labeled trees* in D, for $i = 1, \dots, k$. The node D_{i-1} is called the *direct left sibling labeled tree* of D_i, while D_{i+1} is the *direct right sibling labeled tree* for D_i, with $i = 2, \dots, k-1$. The set of all labeled trees is denoted by $\mathfrak{D}$.

Example. Figure 1 shows an example of a labeled tree. The nodes are represented by circles containing the node label. The node with the label “A” is the root of this example tree. Nodes are connected by labeled edges. The box label D_i above a node denotes the subtree rooted at that node.

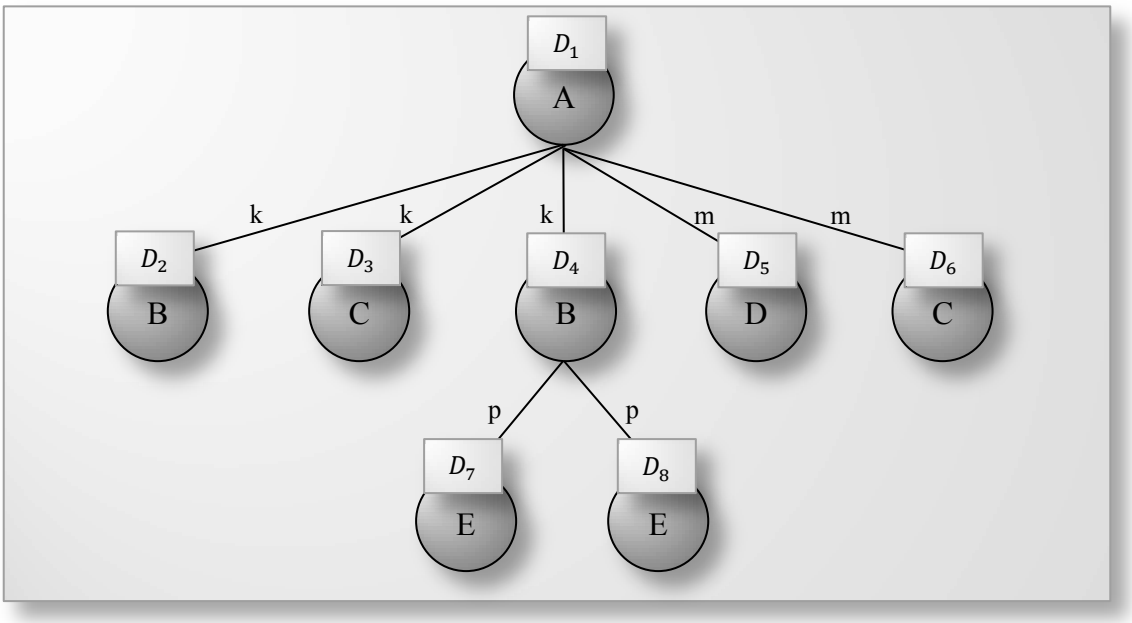


Figure 1. Example of a labeled tree

By using this example tree we will discuss the introduced notions. The direct subtree layer $\mathcal{C}_{"k"}(D_1)$ of the tree D_1 and the layer "k" is given by $\mathcal{C}_{"k"}(D_1) = [D_2, D_3, D_4]$. Further examples of direct subtree layers are $\mathcal{C}_{"m"}(D_1) = [D_5, D_6]$, $\mathcal{C}_{"p"}(D_1) = [\,]$ and $\mathcal{C}_{"p"}(D_4) = [D_7, D_8]$. With $K = \{"C"\}$ we can restrict the layer $\mathcal{C}_{"k"}(D_1)$ to $\mathcal{C}_{"k"}(D_1)|_K = [D_3]$.

The subtrees D_2 and D_4 are equally labeled, thus we have $L(D_2) = L(D_4)$ and $D_2 \approx D_4$. The set of all subtrees of D_1 is $\mathcal{S}(D_1) = \{D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8\}$. Further examples of sets of subtrees are $\mathcal{S}(D_4) = \{D_4, D_7, D_8\}$, $\mathcal{S}([D_4, D_6]) = \{D_4, D_6, D_7, D_8\}$ and $\mathcal{S}([D_2, D_3, D_4]) = \{D_2, D_3, D_4, D_7, D_8\}$. The size of the labeled tree D_1 is $|D_1| = 8$. The size of the partial layer $Y := [D_3, D_4, D_5]$ is $|Y| = 5$. Other examples are $|D_4| = 3$ and $|D_2| = |D_3| = |D_7| = 1$.

Finally, the left sibling labeled trees of D_4 are $S_L(D_4) = [D_2, D_3]$ and the direct left sibling of D_4 is D_3. Furthermore, the right sibling labeled trees of D_4 are $S_R(D_4) = [\,]$ because there are not right siblings of D_4 in the layer k.

3.2 Document Model

Having introduced the basic notions and notation, we define now our *document model* based on the notion of *labeled trees*. To illustrate the model we instantiate it for the document format XML [W3C, 2008], a format that is widely used for the representation of arbitrary data structures, for instance in web services. Furthermore, XML-based formats have become the industry standard for most modern text-editors, including MS WORD (OPEN XML [ISO/IEC, 2008]) and OPENOFFICE (OPENDOCUMENT [ISO/IEC, 2006]).

Definition 3.2.1 (Document): A *document* is a labeled tree D with a set of edge labels $\mathcal{L}_A$ and a set of node labels $\mathcal{L}_V$ which consists of the following pairwise disjoint sets:

- the set of element node labels $\mathcal{L}_{VE}$,
- the set of attribute node labels $\mathcal{L}_{VA}$,
- the set of text node labels $\mathcal{L}_{VT}$, and
- the set of comment node labels $\mathcal{L}_{VC}$.

It holds that $\mathcal{L}_V = \mathcal{L}_{VE} \uplus \mathcal{L}_{VA} \uplus \mathcal{L}_{VT} \uplus \mathcal{L}_{VC}$. The root of the labeled tree is the root node of the document.

Definition 3.2.2 (Document Subtrees): An *element* D_E in a document D is a subtree of D that has a label $L(D_E) \in \mathcal{L}_{VE}$. Its children can be *attributes* that are connected by edges with the *attribute label* $a \in \mathcal{L}_A$. Furthermore, its children can also be *elements*, *texts* or *comments* that are connected by edges with the *content label* $c \in \mathcal{L}_A$. Thus, the element D_E may have at most two layers, one for the attributes and one for the content children. We use the following serialization schema: $\backslash L(D_E)\{\ \mathcal{C}_a(D_E)\ \}\ [\ \mathcal{C}_c(D_E)\ ]$. An illustrating example will be given on the next page.

An *attribute* D_A in a document D is a subtree of D that has a label $L(D_A) \in \mathcal{L}_{VA}$. Its child is always a *text* D_T that is connected by an edge with the *value label* $v \in \mathcal{L}_A$. Thus, the attribute D_A always has exactly one layer, and it is serialized as $L(D_A)$ = "$L(D_T)$".

A *text* D_T in a document D is a subtree of D that has a label $L(D_T) \in \mathcal{L}_{VT}$. A *text* has no children and we serialize it by its label as "$L(D_T)$".

A *comment* D_C in a document D is a subtree of D that has a label $L(D_C) \in \mathcal{L}_{VC}$. The comment has no children and we serialize it by its label as $\#L(D_C)\#$.

Throughout this thesis we always assume *well-formed documents*, that is, we presuppose that all documents conform to our document model. This model is not intended to represent information like the *document type definition* or *processing instructions* which are parts of the XML format.

```
<w:body xmlns:w="http://schemas.openxmlformats.org/word...">
  <w:p>
    <w:r>
      <w:format><w:color val="00FF00"/></w:format>
      <w:t>First paragraph of a text document ...</w:t>
      <!-- TODO -->
    </w:r>
  </w:p>
</w:body>
```

Table 2. Example Document in OpenXML Format

In order to deal with *namespaces* occurring in an XML document, we set up a global namespace store and define the node labels for elements $\mathcal{L}_{VE}$ and for attributes $\mathcal{L}_{VA}$ to be pairs (l_{ns}, l_{name}) serialised as l_{ns} ":" l_{name} , where

- l_{ns} refers to a namespace instance in the global namespace store, and
- l_{name} is the name of the element or attribute.

The example document in Table 2 shows the body of a simple MS Word document in OpenXML format that is represented in our document model as shown in Figure 2 and that is serialized as shown in Table 3, where `w` refers to the namespace **`"http://schemas.openxmlformats.org/word..."`** in the global namespace store.

```
\w:body{}[
 \w:p{}[
  \w:r{}[
   \w:format{}[ \w:color{ val="00FF00" }[ ]],
   \w:t{}[ "First paragraph of a text document ..." ],
   #TODO# ]]]
```

Table 3. Example Document in Labeled Tree Format

We have shown how to represent an XML document in our document model as an instance of labeled trees. Thereby, the name and namespace of an XML element or attribute are encoded in the label of that subtree. The partition between attributes and content is realized using the layers of labeled trees, that is the children are connected by an edge with either an attribute or a content label.

The advantage of using labeled trees instead of the XML document model is that we are not restricted to two layers. Additionally, the values of attributes are not restricted to text values. Indeed, we can use arbitrary many layers containing arbitrary labeled (sub)trees. This allows us for example for easily representing the document format of the text-editor TEXMACS, which uses natively the tree-like Scheme s-expression format.

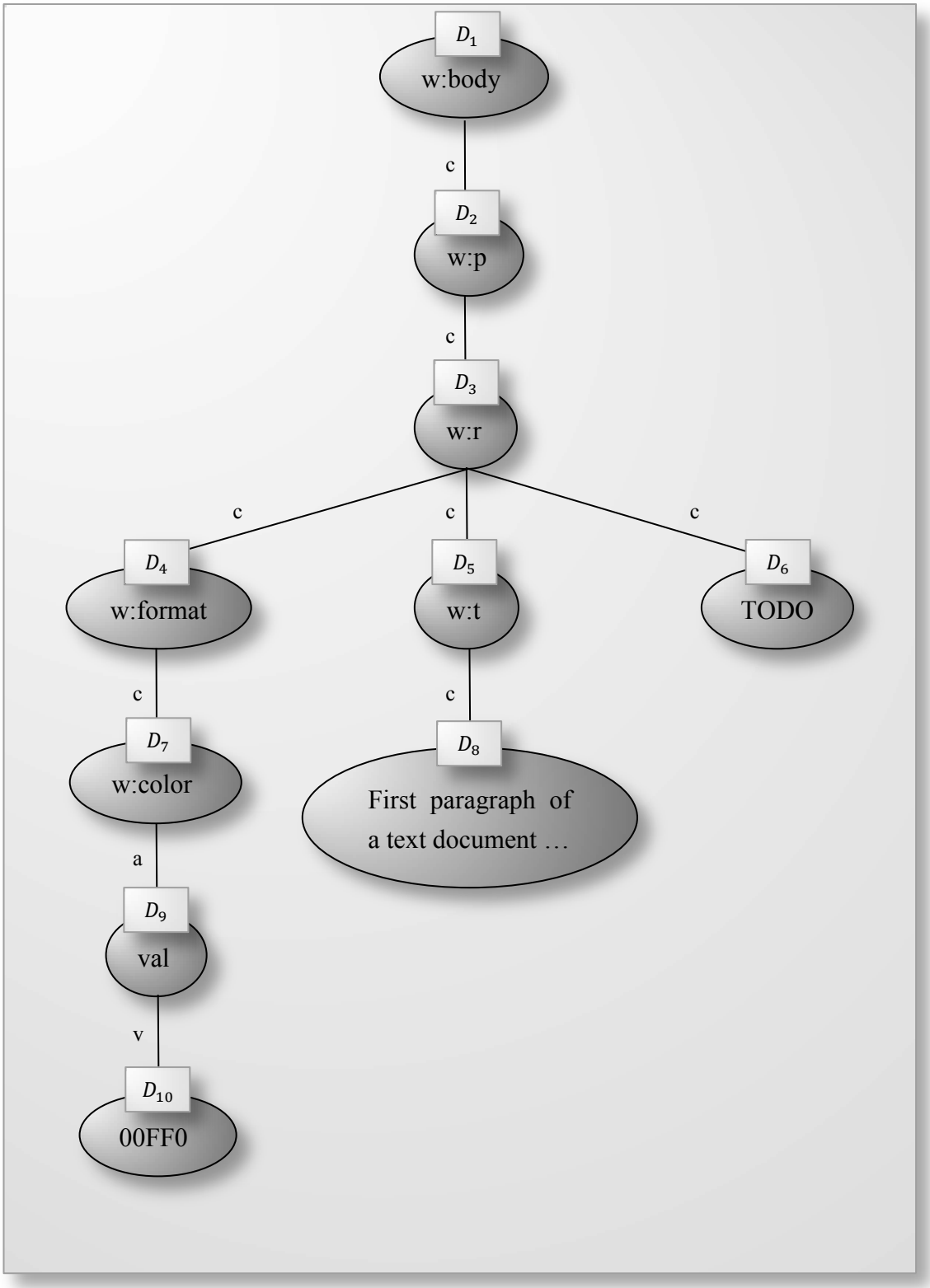


Figure 2. Example Document as a labeled tree

3.3 Change Model

The problem of finding a transformation between two given trees, in our case labeled trees, is known as the *tree-to-tree correction problem* [Selkow, 1977], which we formulate in our setting as follows: Given two labeled trees, find a valid *edit script* for transforming the first tree into the second one with minimal *edit costs*.

The problem statement requires the formal definitions of the notions of *edit script* and *edit costs*. The usual approach is to define *edit operations* such that any given tree can be transformed into another one by applying a sequence of those operations, called *edit script*. The edit costs are then usually defined as a static function on such a sequence. This state-of-the-art method has the drawbacks that neither the semantics of the trees are taken into account for defining the edit costs, nor the needs of the consumer of the edit script. In the following, we will focus on the definition of the *edit operations*, the notion of *edit costs* and their optimality will be discussed in Chapter 4.

The problem of comparing trees has to be addressed in several research areas such as computational biology [Lin *et al*, 2001], structured databases [Claypool & Rundensteiner, 2004], compiler optimization [Wilhelm, 1981] and others. The comparative studies of change models and algorithms in [Bille, 2005] and [Cobéna *et al*, 2002] show that the existing change models differ in the set of supported *edit operations* and their semantics. The edit operations identified by these surveys are *relabel*, *insert, delete, replace*, *append* and *move*. The semantics of these operations can be distinguished whether they are defined on nodes of a tree or on whole subtrees. Based on these edit operations, the following three classes of tree-to-tree correction problems were defined:

Tree edit distance. This is the problem of computing the optimal edit script between two trees, which is defined as a sequential edit script with minimal costs.

Tree alignment distance. An alignment between two trees is obtained by first inserting nodes with empty labels into both trees such that the trees become isomorphic when ignoring the labels. The tree alignment is then an overlay of both trees.

Tree Inclusion. As the name of the problem indicates, the tree inclusion problem determines whether a tree can be obtained from another tree only by deleting nodes.

In addition to these problem classes, one differentiates between comparing ordered and unordered labeled trees. Therefore, we will introduce in Chapter 4 a specification to define the semantics of the tree comparison between specific interface documents.

We base our document change model on the edit operations *insert, delete, replace* and *append*. This decision clearly needs to be justified. Why do we not support the *relabel* operation? Let us discuss this important question with the example shown in Table 4.

\theory{}[\theorem{}[\name{}[**"XY"**], \conj{}[**"#1"**]]]	\theory{}[\axiom{}[\name{}[**"XY"**], \conj{}[**"#1"**]]]

Table 4. Comparing two documents D_1 (on the left) and D_2 (on the right)

The document D_1 can be modified to become equal to the document D_2 by relabeling the root node of the theorem subtree into an axiom. Clearly, this is the most concise change description one can image. But in our context, this document represents the state of knowledge of an interfaced service component. For example let adding and deleting theorems as well as adding and deleting axioms be the only interface methods provided by the service component. Thus, relabeling a theorem into an axiom cannot be mapped directly to a valid interface method. We would have to delete the theorem and insert the axiom.

In this example, we relabeled a "*theorem*" into the ontologically similar concept "*axiom*". This ontological relationship is definitely not the general case for the relabel operation. Thus, if we would support the relabel operation, the interfaced service components would have to provide interface methods for converting between arbitrary objects. Since this is not a reasonable requirement, we do not support the relabel operation.

Why do we not support the *move* operation? Detecting a move operation is not always a decidable problem. Consider for example the sequences ABC and $BACB$. Is the first B the moved one or the second one? In case of a false positive, this change may propagate throughout the components of the system and result in a non-intended state. There are algorithms [Chawathe *et al*, 1996] that use the move operation to compute more compact edit scripts, but they only compute approximate solutions. Additionally, the move operation is not restricted to the same tree layer but may move a subtree to an arbitrary position, which increases the complexity of detecting a move operation. For all these reasons, we decided not to use the move operation for our system interaction purposes.

Regarding the semantics of the edit operations, there exist two prominent models: *Kuo-Chung Tai's model* [Tai, 1979] and *Selkow's model* [Selkow, 1977]. In Tai's model, deleting a node means making its children become children of the node's parent. Most tools use Selkow's model where the entire subtree rooted at the node is deleted. In conformance with the semantics used by the *XUpdate specification* [XML:DB, 2000], a specification for XML edit scripts, we decided to follow the semantics of Selkow's model.

Definition 3.3.1 (Valid Edit Operations): We define the following *valid edit operations* for a labeled tree D and any subtree D_x of D, referenced by $\overrightarrow{D_x}$, to modify D according to the following semantics:

- a *delete* operation $\delta_E(\overrightarrow{D_x})$ with $D_x \neq D$ which removes D_x.
- a *replace* operation $\delta_R(\overrightarrow{D_x}, D_1)$ which replaces D_x by D_1.
- an *insert* operation $\delta_I(\overrightarrow{D_x}, [D_1, \ldots, D_n])$ with $\overrightarrow{D_x} \neq \vec{D}$ and $n \geq 1$ which adds the labeled trees $D_1, \ldots, D_n$ to D as follows: The ordered sequence of labeled trees $[D_1, \ldots, D_n]$ are inserted as direct left siblings of D_x such that $S_L(D_x) \coloneqq S_L(D_x) :: [D_1, \ldots, D_n]$.
- an *append* operation $\delta_A(\overrightarrow{D_x}, l, [D_1, \ldots, D_n])$ with $n \geq 1$ which adds the labeled trees $D_1, \ldots, D_n$ to D as follows: The ordered sequence of labeled trees $[D_1, \ldots, D_n]$ are appended to the layer $\mathcal{C}_l(D_x)$. If D_y is the last node in $\mathcal{C}_l(D_x)$, then we have $S_R(D_y) \coloneqq [D_1, \ldots, D_n]$.

We say that these edit operations *target* the tree D_x, written as $target(\delta) = \overrightarrow{D_x}$ where δ is an edit operation. Furthermore, we say that δ_E, δ_R and δ_A *modify* the target tree D_x and δ_I *modifies* the parent of the target tree. This set of edit operations allows to manipulate an arbitrary subtree D_k of D and its children to any other subtree D'_k. Thus, a labeled tree D can be modified by the edit operations at an arbitrary level of granularity.

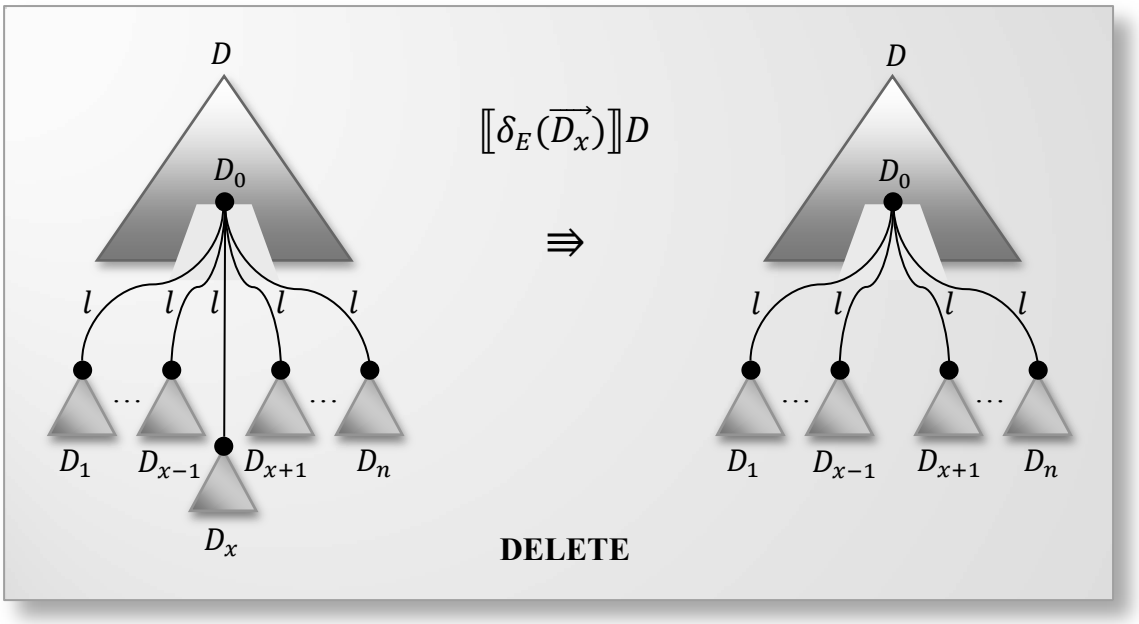
For the serialization of an edit operation we mainly need a unique representation for the subtree reference $\overrightarrow{D_x}$. This is usually realized by a path schema which uniquely identifies a subtree. In conformance with existing standards we use the XPATH specification [W3C, 1999] to represent a reference to a subtree in an XML instance of labeled trees. Thereby, the subtrees in an attribute layer are distinguished from the subtrees in a content layer by a prefixed "@". For example, the path `/w:body[1]/w:p[1]/w:r[1]` refers to the `w:r` subtree in our example shown in Table 3. An XPATH expression may in general evaluate to arbitrary many subtrees, however, in our context we allow only XPATH expressions that evaluate to exactly one subtree.

Let p be a valid path referring to a subtree D_x in the labeled tree D, and let $D_1, \ldots, D_n$ be serialized labeled trees, then we serialize the edit operations as labeled trees:

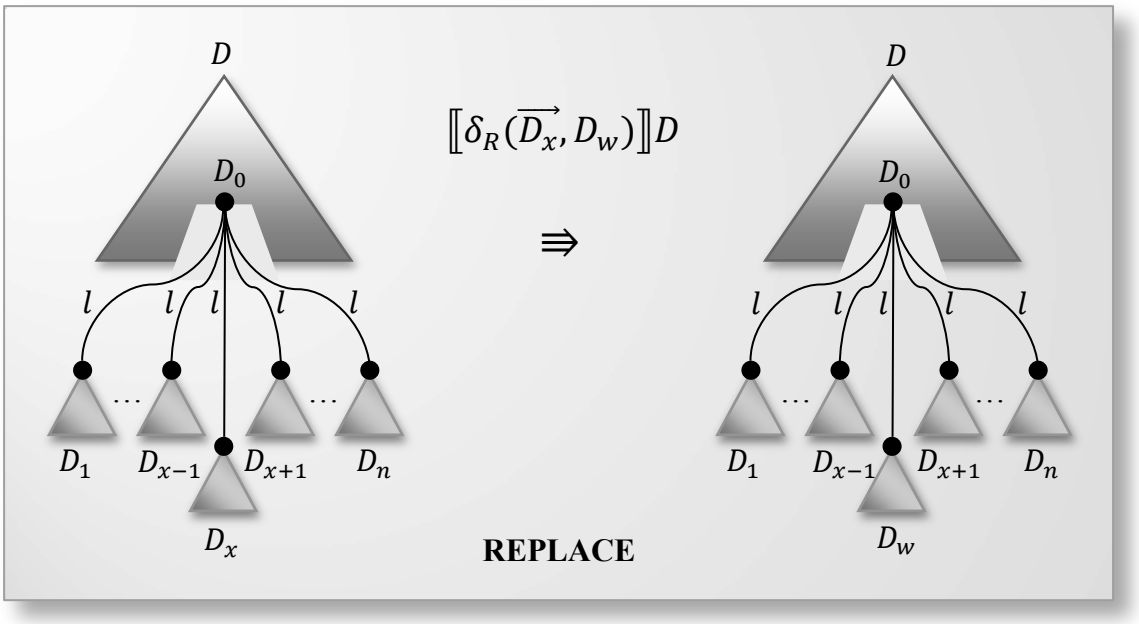
- a *delete* operation: `\delete{ target=`p `}[]`
- a *replace* operation: `\replace{ target=`p `}[` D_1 `]`
- an *insert* operation: `\insert{ target=`p `}[` $D_1, \ldots, D_n$ `]`
- an *append* operation: `\append{ target=`p`, layer=`l `}[` $D_1, \ldots, D_n$ `]`

In the following we will illustrate the introduced edit operations. The triangles represent labeled (sub)trees and the big dots represent nodes in the tree. The edges are ordered from left to right. The application of an edit operation δ on a tree D is denoted by $[\![\delta]\!]D$.

The *delete* operation $\delta_E(\overrightarrow{D_x})$ removes the whole subtree D_x from the labeled tree D. Thereby the former direct right sibling of D_x (if one exists) becomes the direct right sibling of the former direct left sibling of D_x (if one exists).

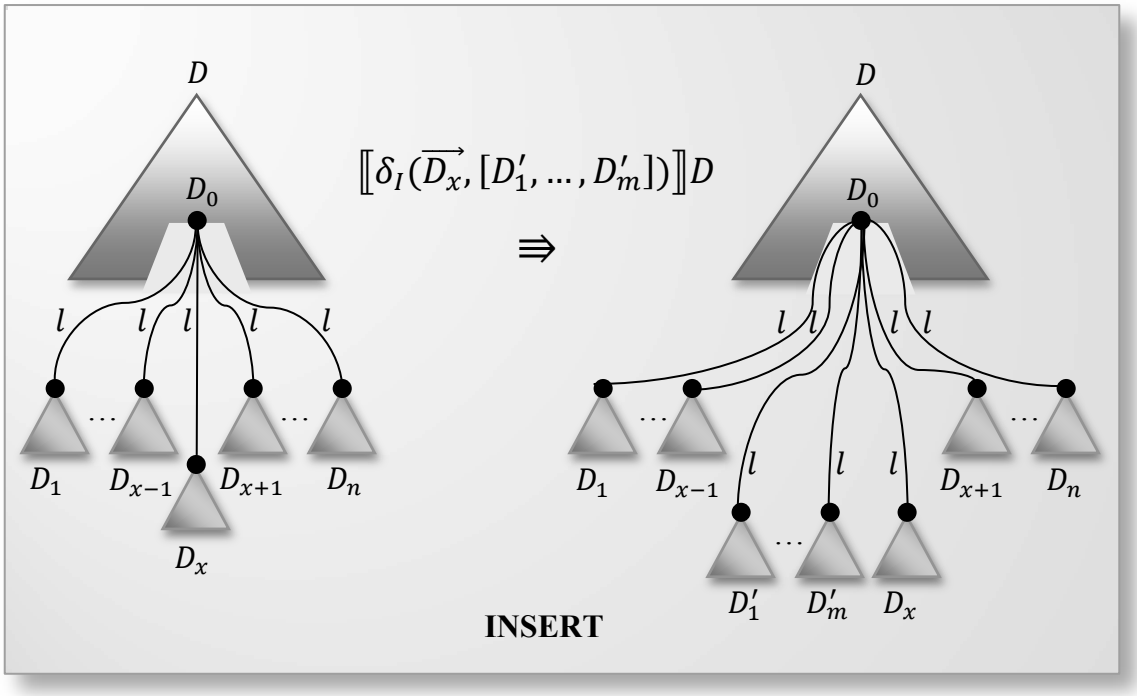


The *replace* operation $\delta_R(\overrightarrow{D_x}, D_w)$ replaces the whole subtree D_x by the labeled tree D_w. Thereby the former direct right sibling of D_x (if one exists) becomes the direct right sibling of the root of D_w. The former direct left sibling of D_x (if one exists) becomes the direct left sibling of the root of D_w.

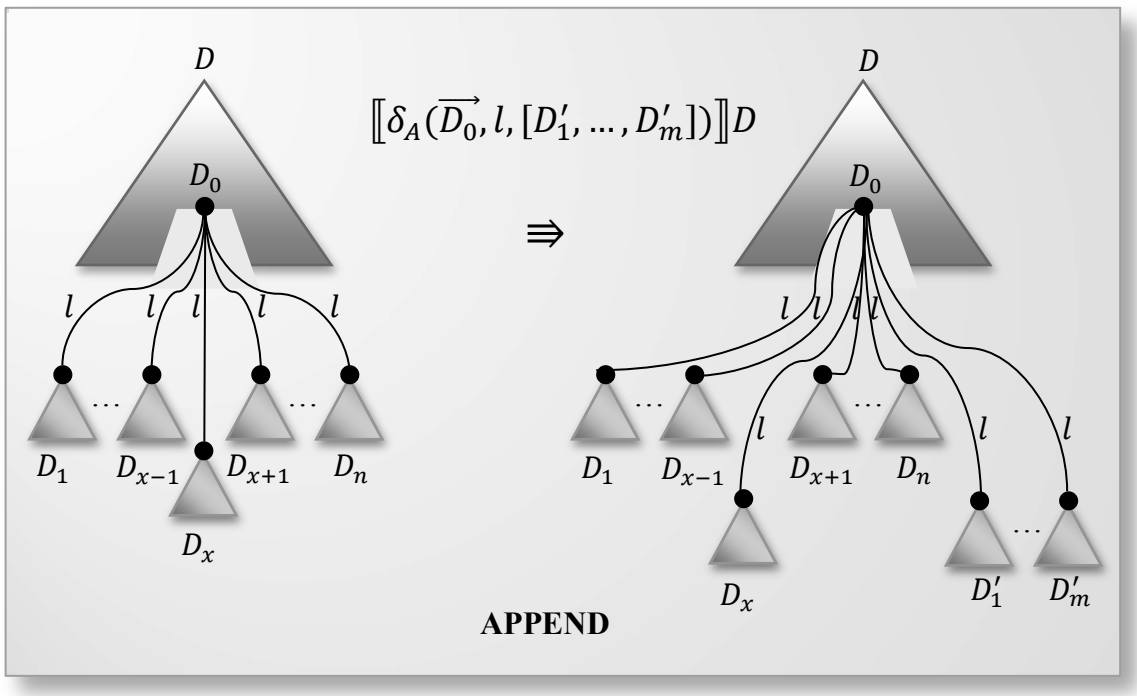


Note that all nodes and edges in the subtree D_i no longer exist after applying a delete or replace operation on the target D_i.

The *insert* operation $\delta_I(\overrightarrow{D_x}, [D'_1, \ldots, D'_m])$ adds the trees $D'_1, \ldots, D'_m$ to the tree D such that they become left siblings of D_x. Thereby the former direct left sibling of D_x (if it exists) becomes the direct left sibling of the inserted tree D'_1.



The append operation $\delta_A(\overrightarrow{D_0}, l, [D'_1, \ldots, D'_m])$ adds the trees $D'_1, \ldots, D'_m$ to the tree D such that they become the right siblings of the last child of D_0 in the layer labeled by l. If D_0 does not have any child in this layer, the trees $D'_1, \ldots, D'_m$ become the new children of D_0 in this layer.



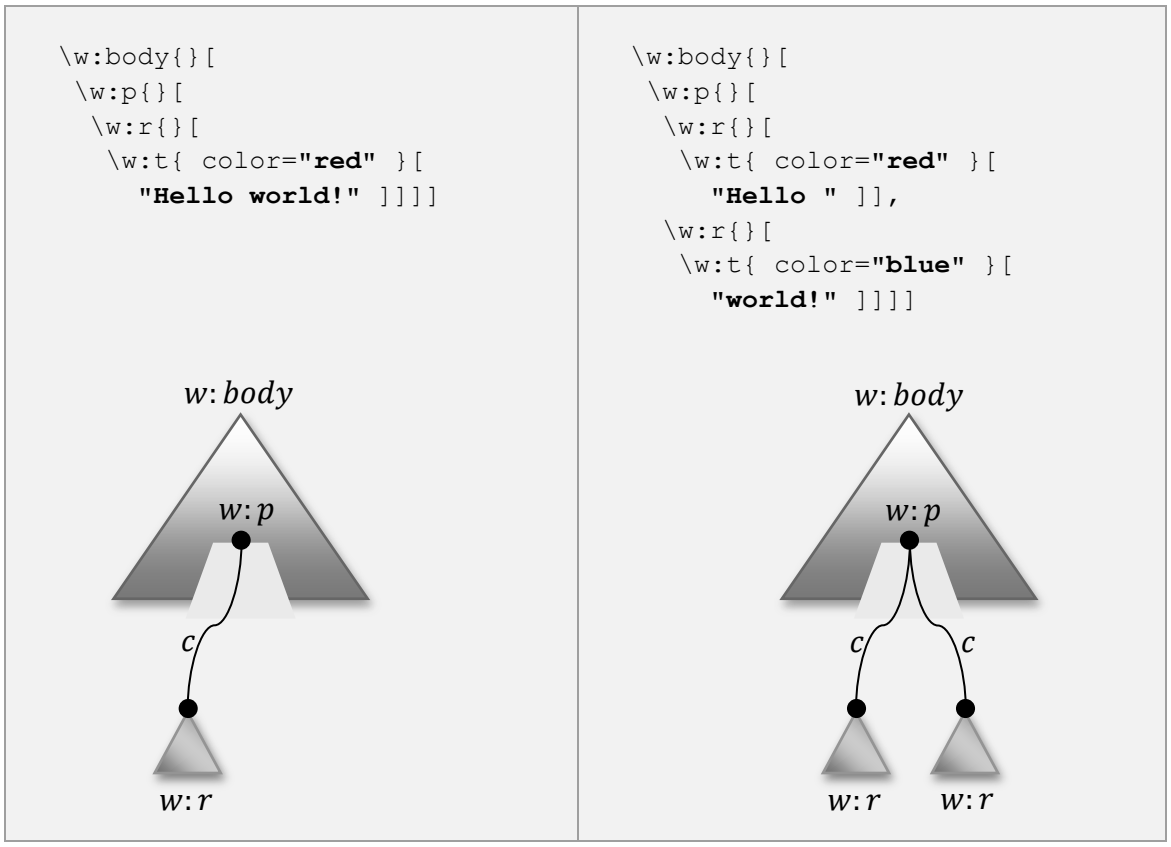


Table 5. Comparing two informal documents A (on the left) and A' (on the right)

Let us consider the example in Table 5 to get familiar with the change model. Intuitively, one would summarize the changes from document A to document A' as follows: The text `"Hello world!"` has been replaced by `"Hello "`, and a new element `\w:r{}[...]` has been appended after the last child of the tree `\w:p{}[...]`, namely the element tree `\w:r{}[...]`. The changes can be represented by the following valid edit operations:

```
\append{ target=/w:body[1]/w:p[1], layer="content" }
    [ \w:r{}[ \w:t{ color="blue" }[ "world! " ]]]
\replace{ target=/w:body[1]/w:p[1]/w:r[1]/w:t[1]/text[1] }
    [ "Hello " ]
```

Note that this is not the only valid set of edit operations that patches the document A to the document A'. There are several parameters that may influence the computation of the set of edit operations, the so-called *edit script*:

- First, the differencing mechanism has to identify corresponding subtrees. When should two subtrees be considered similar? Let us define for our example that two elements `\w:t{...}[...]` are similar if all attributes and the whole content are similar. Then we would have removed that subtree and inserted a new one instead of replacing it because we only want to modify the content of similar trees deeply.

- Second, some subtrees in the document might be related to external resources. Deleting these subtrees should be considered more expensive than deleting other subtrees, because of the additional costs for the internal management of change.
- Third, we might only be interested in change descriptions up to a specific depth, for example because we post-process the change set. The computation of too fine-grained modifications would be a waste of resources. In our example, we could have for example replaced the element `\w:p{...}[...]` by its new version.

We will discuss these semantic aspects of changes in more detail in Chapter 4.

Having defined the basic edit operations, we need means to apply these operations to a document and to group edit operations for representing the changes between documents.

Definition 3.3.2 (Application of an Edit Operation): Let δ be a valid edit operation for a labeled tree D. The *application of this edit operation* is defined by $[\![\delta]\!]D := D'$ where D' denotes the tree resulting from applying the edit operation according to *Definition 3.3.1*. We also say that the labeled tree D is *patched* by the edit operation δ. For an invalid edit operation δ_x we define its application by $[\![\delta_x]\!]D := D$. Thus the tree is not modified in this case.

The edit operations are the atomic change operations for labeled trees. They are the building blocks to transform one labeled tree into another. In order to describe a complete transformation, we introduce the notion of an edit script as a sequence of valid edit operations.

Definition 3.3.3 (Edit Script): An *edit script* Δ for a labeled tree D is a sequence of edit operations that are all valid for D. Additionally, we define the notation for the following subsequences of Δ that filter the edit script by a particular type of edit operation:

1) Filter by type of edit operations
   a) Delete operations $\Delta_E := [\delta_E \in \Delta]$
   b) Insert operations $\Delta_I := [\delta_I \in \Delta]$
   c) Replace operations $\Delta_R := [\delta_R \in \Delta]$
   d) Append operations $\Delta_A := [\delta_A \in \Delta]$

2) *Filter by target subtree $D_k \in \mathcal{S}(D)$*
   a) Operations that target a tree $D' \in \mathcal{S}(D_k)$ except the tree D_k itself
   $\Delta^{(\overrightarrow{D_k})} := [\delta \in \Delta | target(\delta) \in \mathcal{S}(\overrightarrow{D_k}) \wedge target(\delta) \neq \overrightarrow{D_k}]$

b) Operations that target the tree D_k (without its subtrees)
$\Delta|\overrightarrow{D_k} \coloneqq [\delta \in \Delta | target(\delta) = \overrightarrow{D_k}]$

c) Append operations that target the layer $l \in \mathcal{L}_A$ of the tree D_k
$\Delta|(\overrightarrow{D_k}, l) \coloneqq [\delta \in \Delta | \delta = \delta_A(\overrightarrow{D_k}, l, \dots)]$

Edit operations may insert new trees into a labeled tree or delete existing ones. We can identify these trees by the following notational convention.

Notation 3.3.4 (Trees Inserted by an Edit Operation): Let δ be a valid edit operation for the labeled tree D, and let D_i be labeled trees for all $i \in \{1, \dots, n\}$, and let $D_k \in \mathcal{S}(D)$. The set of *trees inserted by the edit operation* δ, denoted by Ψ_+^δ , is the following set depending on the type of the edit operation δ.

$$\Psi_+^\delta \coloneqq \begin{cases} \emptyset & if\ \delta = \delta_E(\overrightarrow{D_k}) \\ \{D_1\} & if\ \delta = \delta_R(\overrightarrow{D_k}, D_1) \\ \{D_1, \dots, D_n\} & if\ \delta = \delta_I(\overrightarrow{D_k}, [D_1, \dots, D_n]) \\ \{D_1, \dots, D_n\} & if\ \delta = \delta_A(\overrightarrow{D_k}, l, [D_1, \dots, D_n]) \end{cases}$$

Notation 3.3.5 (Trees Deleted by an Edit Operation): Let δ be a valid edit operation for the labeled tree D, and let D_i be labeled trees for all $i \in \{1, \dots, n\}$, and let $D_k \in \mathcal{S}(D)$. The set of *trees deleted by the edit operation* δ, denoted by Ψ_-^δ , is the following set depending on the type of edit operation δ.

$$\Psi_-^\delta \coloneqq \begin{cases} \{D_k\} & if\ \delta = \delta_E(\overrightarrow{D_k}) \\ \{D_k\} & if\ \delta = \delta_R(\overrightarrow{D_k}, D_1) \\ \emptyset & if\ \delta = \delta_I(\overrightarrow{D_k}, [D_1, \dots, D_n]) \\ \emptyset & if\ \delta = \delta_A(\overrightarrow{D_k}, l, [D_1, \dots, D_n]) \end{cases}$$

We have defined an edit script to contain edit operations that are all valid for the same labeled tree D. Thus, the operations of an edit script have to be applied simultaneously. This raises the problem how to prevent conflicts between multiple operations. Two or more valid edit operations, for example a delete and a replace operation both targeting the same subtree, may conflict with each other, and thus invalidate an edit script.

In order to prevent these conflicts, we define the following properties that guarantee the validity of an edit script and additionally imply a normal form for edit scripts.

Definition 3.3.6 (Valid Edit Script): An edit script Δ for a labeled tree D is called a *valid edit script* if the following properties hold for any $D_k \in \mathcal{S}(D)$ and any $l \in \mathcal{L}_A$:

1) *Operational Consistency of Edit Operations:*
   a) $|(\Delta_{\mathrm{E}}|\overrightarrow{D_k})| \leq 1$
   b) $|(\Delta_{\mathrm{I}}|\overrightarrow{D_k})| \leq 1$
   c) $|(\Delta_{\mathrm{R}}|\overrightarrow{D_k})| \leq 1$
   d) $\left|\left(\Delta_{\mathrm{A}}|(\overrightarrow{D_k}, l)\right)\right| \leq 1$
2) *Structural Consistency of Edit Operations:*
   $$\left((\Delta_{\mathrm{E}}|\overrightarrow{D_k}) \cup (\Delta_{\mathrm{R}}|\overrightarrow{D_k}) \neq \emptyset\right) \Rightarrow \left((\Delta_{\mathrm{A}}|\overrightarrow{D_k}) = \left(\Delta^{(\overrightarrow{D_k})}\right) = \emptyset\right)$$
3) *Logical Consistency of Edit Operations:*
   $$\left|(\Delta_{\mathrm{E}}|\overrightarrow{D_k})\right| + \left|(\Delta_{\mathrm{R}}|\overrightarrow{D_k})\right| \leq 1$$
4) *Partial Order of Edit Operations:*
   $$\Delta = \Delta_I :: \Delta_A :: \Delta_R :: \Delta_E$$

The set of all valid edit scripts for a labeled tree D is denoted by D^{Δ}.

The *Properties 1a-1c* guarantee that there is at most one delete, insert and replace operation per subtree. For the operations delete and replace this restriction is intuitively clear, for the insert operation it means that all trees, that have to be inserted before a subtree, have to be inserted using a single operation instead of several ones. *Property 1d* guarantees that the same restriction holds for the append operation and specific tree layers.

Property 2 ensures that there is no modification in a subtree allowed if the subtree itself is deleted or replaced. *Property 3* guarantees that a subtree is not both deleted and replaced. Finally, *Property 4* imposes a partial order based on the types of the edit operations. Replace and delete operations are the last edit operations in an edit script because their application might remove subtrees targeted by other operations.

In the following we will discuss whether the union or difference of valid edit scripts produces in turn a valid edit script. Furthermore, we will prove that an edit script, which satisfies the above properties, can be applied sequentially although the semantics of the edit script requires a parallel application of the contained edit operations. Additionally we will analyze the confluence of valid edit scripts. In particular we will show that the application of a valid edit script or one of its order-variants, which preserves the partial order property, results in equal labeled trees.

The trees inserted or deleted by a valid edit script are clearly the trees inserted respectively deleted by the contained valid edit operations.

Definition 3.3.7 (Trees Inserted by an Edit Script): Let Δ be a valid edit script for the labeled tree D. The set of *trees inserted by the edit script* Δ are defined as follows:

$$\Psi_+^\Delta := \begin{cases} \Psi_+^\delta \cup \Psi_+^{\Delta'} & if\ \Delta = \delta * \Delta' \\ \emptyset & if\ \Delta = [\,] \end{cases}$$

Definition 3.3.8 (Trees Deleted by an Edit Script): Let Δ be a valid edit script for the labeled tree D. The set of *trees deleted by the edit script* Δ are defined as follows:

$$\Psi_-^\Delta := \begin{cases} \Psi_-^\delta \cup \Psi_-^{\Delta'} & if\ \Delta = \delta * \Delta' \\ \emptyset & if\ \Delta = [\,] \end{cases}$$

We overloaded the symbols Ψ_+ and Ψ_- because the intended meaning can always be inferred from the arguments.

So far, we have defined the notions of valid edit operations and valid edit scripts. In order to manipulate and manage valid edit scripts, the question comes up whether and how the union and difference of two valid edit scripts are defined.

Definition 3.3.9 (Union of Valid Edit Scripts): Let Δ_1 and Δ_2 be valid edit scripts for a labeled tree D. The union of both edit scripts is defined by:

$$\Delta_1 \boxplus \Delta_2 := \Delta_{1,I} :: \Delta_{2,I} :: \Delta_{1,A} :: \Delta_{2,A} :: \Delta_{1,R} :: \Delta_{2,R} :: \Delta_{1,E} :: \Delta_{2,E}$$

By construction it satisfies the *Property 4* of valid edit scripts. If and only if the *Properties 1, 2 and 3* are additionally satisfied by $\Delta_1 \boxplus \Delta_2$, it is a valid edit script for D.

Hence, the union of valid edit scripts is in general not a valid edit script, but for valid edit scripts targeting different tree layers we can prove that their union is a valid edit script.

Lemma 3.3.10 (Union of Valid Edit Scripts for Tree Layers): Let D be a labeled tree, $l \in \mathcal{L}_A$ be an edge label, and let Δ_l be a valid edit script for D such that all $\delta \in \Delta_l$ target a tree in the direct subtree layer $\mathcal{C}_l(D)$ or one of its subtrees. Then $\Delta := \boxplus_{l \in \mathcal{L}_A} \Delta_l$ is a valid edit script for D.

Proof: We have to show that Δ satisfies the properties of valid edit scripts. *Property 1* requires that there is at most one edit operation for a specific subtree and tree layer. The target subtrees of all edit operations in pairwise different Δ_l are not overlapping because they target trees in different layers of the tree D or subtrees of them. Therefore, the *Property 1* is satisfied. *Property 2* requires that there is no modification in a subtree allowed if the subtree itself is deleted or replaced. Following the previous argumentation, there cannot occur such an inconsistent situation. Therefore, the *Property 2* is satisfied. *Property 3* requires that a subtree is not both deleted and replaced. Analogously, this holds because the target subtrees of edit operations in pairwise different Δ_l are not overlapping. Finally, the partial order of edit operations required by *Property 4* is satisfied by the construction of the union of valid edit scripts. We conclude that Δ is a valid edit script for D.

■

In contrast to that, the difference of valid edit scripts is always a valid edit script.

Definition 3.3.11 (Difference of Valid Edit Scripts): Let Δ_1 and Δ_2 be valid edit scripts for a labeled tree D. The difference of both edit scripts is defined as:

$$\Delta_1 \boxminus \Delta_2 \coloneqq (\Delta_{1,I} \backslash \Delta_{2,I}) :: (\Delta_{1,I} \backslash \Delta_{2,I}) :: (\Delta_{1,I} \backslash \Delta_{2,I}) :: (\Delta_{1,I} \backslash \Delta_{2,I})$$

By definition it satisfies the *Property 4* of valid edit scripts. Since the *Properties 1, 2 and 3* are preserved by the subsequence $\Delta_1 \boxminus \Delta_2$ of Δ_1, it is a valid edit script for D.

Ideally, we would like to apply the edit operations of a valid edit script sequentially instead of simultaneously. We will show now, that the properties of valid edit script indeed guarantee a conflict-free sequential application of their contained edit operations.

Lemma 3.3.12 (Sequentiality of Valid Edit Scripts): Let $\Delta = \delta \star \Delta'$ be a non-empty valid edit script for a labeled tree D. Then Δ' is a valid edit script for the tree $[\![\delta]\!]D$ which is the tree D patched by the valid edit operation δ.

Proof: First, we have to show that Δ' is an edit script for $[\![\delta]\!]D$. The type of edit operations δ_R and δ_E are the only ones that remove subtrees from the labeled tree D and thus potentially invalidate edit operations in Δ' by removing their target subtree. Thus we have to consider the two cases (1) $\delta = \delta_R$ and (2) $\delta = \delta_E$ where δ targets a subtree $D_k \in \mathcal{S}(D)$. Because of *Property 4*, the sequence Δ' contains in both cases only δ_R or δ_E operations. Thus, we have to show that none of these operations target a subtree $D' \in \mathcal{S}(D_k)$.

Property 3 prevents a δ_R operation and a δ_E operation from targeting the same subtree. The *Properties 1a-1d* prevent two δ_R operations (δ_E respectively) from targeting the same subtree. Thus there is no edit operation in Δ' that targets D_k. Finally, *Property 2* prevents all edit operations in Δ' from targeting a tree in $\mathcal{S}(D_k)\backslash\{D_k\}$. Hence, the application of the edit operation δ to the tree D does not invalidate any edit operation in the sequence Δ'. Therefore, Δ' is an edit script for $[\![\delta]\!]D$. Since the *Properties 1, 2 and 3* are clearly preserved by all subsequences of Δ, and since removing the first edit operation preserves the order required by *Property 4*, the edit script Δ' is valid for $[\![\delta]\!]D$.

■

This lemma allows us to define the application of a valid edit script to a labeled tree as a sequential operation.

Definition 3.3.13 (Application of a Valid Edit Script): Let D_1 and D_2 be labeled trees, and let Δ be a valid edit script for D_1. The judgment of deriving the labeled tree D_2 from D_1 by applying the edit script Δ is denoted by $(D_1, \Delta) \hookrightarrow_{\mathrm{PATCH}} \mathrm{D}_2$. We also say that the labeled tree D_1 is *patched* by the edit script Δ to the labeled tree D_2. The operational semantics of $\hookrightarrow_{PATCH}$ is defined by the inference rules in Table 6.

$$\frac{([\![\delta]\!]D_1, \Delta) \hookrightarrow D_2}{(D_1, \delta \star \Delta) \hookrightarrow D_2} \qquad \frac{}{(D_1, [\,]) \hookrightarrow D_1}$$

Table 6. Algorithm PATCH

The properties of a valid edit script do not impose a unique normal form, because we only require a partial ordering of the edit operations. Therefore we want to show that the application of all ordering variants of a valid edit script result in the same modified tree. The variants of a valid edit script contain the same edit operations but in a different order. Nevertheless, a variant has to respect the properties of valid edit scripts, in particular the partial order required by *Property 4*. Thus, the reordering possibilities are reduced to the subsequences containing edit operations of the same type.

Definition 3.3.14 (Variants of a Valid Edit Script): Let Δ_1 and Δ_2 be valid edit scripts for a labeled tree D. The edit script Δ_2 is called a *variant* of the edit script Δ_1 if $|\Delta_1| = |\Delta_2| = n$ and if there is a total function $f:\{1,\dots,n\} \to \{1,\dots,n\}$ such that $\forall i \in \{1,\dots,n\}.\big((i,\delta) \in \Delta_1 \Rightarrow (f(i),\delta) \in \Delta_2\big)$. The notion of variants is clearly reflexive, symmetric and transitive, thus satisfying the properties of an equivalence relation.

We show now that a labeled tree patched by different variants of a valid edit script results in the same labeled tree.

Lemma 3.3.15 (Confluence of Variants of a Valid Edit Script): Let Δ_1 be a valid edit script for a labeled tree D and Δ_2 be a variant of Δ_1. Then we have $(D, \Delta_1) \hookrightarrow_{\text{PATCH}} D_1$ and $(D, \Delta_2) \hookrightarrow_{\text{PATCH}} D_2$, and it holds that $D_1 = D_2$.

Proof: The edit operations in the variant Δ_2 of Δ_1 are ordered differently but with respect to the partial order property of valid edit scripts. Thus, only the subsequences containing edit operations of the same type can be ordered differently. In order to prove that the application of Δ_2 results in an equal labeled tree as the application of Δ_1, we will show that all edit operations of the same type in a valid edit script are non-interfering.

The *replace* and *delete* operations modify the tree only at the position of their target subtrees. By the *Properties 1a, 1c and 3*, it follows that the target subtrees of all edit operations of these types contained in Δ_2 are pairwise different target subtrees. Thus, changing the order of these edit operations has no effect on the resulting labeled tree. The in*sert* and *append* operations add new subtrees to the tree. In a fixed but arbitrary layer of a subtree of D, the *insert* operation inserts new subtrees before the position of the target subtree and the *append* operation after the position of the last child. Hence, *insert* and *append* operations are non-interfering in the same tree layer. *Property 1b* guarantees furthermore that the target subtrees of all *insert* operations are different. Analogously this holds for all *append* operations by *Property 1d*. Thus, changing the order of these edit operations has also no effect on the resulting labeled tree. Hence we have $D_1 = D_2$.

■

This lemma shows indirectly that the variants of a valid edit script are actually an *equivalence class for a valid edit script*. They are equivalent in the sense that the result of their application is an equal labeled tree. Thus, whenever we construct valid edit scripts in the following, we do not need to take care about the order of adding valid edit operations to a valid edit script. When we write $\delta \star \Delta$, the edit operation δ is added to the sequence Δ such that the resulting sequence satisfies the partial order required by *Property 4*.

The question arises whether two valid edit scripts for a labeled tree that patch that tree to an equal tree are always variants of each other? The answer is no. Think about a labeled tree D with two children D_1 and D_2 on the same layer. We want to swap the children. Our options are: (1) insert D_2 before D_1 and delete the last D_2, (2) append D_1 to the same layer of D and delete the first D_1, (3) replace D completely by the desired new tree. These edit scripts are all valid for D but they are no variants of each other.

3.4 Architecture Model

Having defined the document model and the change model, we now give an overview of the *Change-Oriented Architecture* (COA), its principles and its design patterns. We begin with a description of the *Service-Oriented Architecture* (SOA) and illustrate the problem of *change management*.

SOA is a recent technique to integrate distributed applications on the web. SOA does not define an API, but an interface model to the services in terms of protocols and functionality. A concrete service interface is typically described using the *Web Services Description Language* (WSDL). This description defines the signature of the service in terms of its methods, argument and result types. Due to the processing overhead of the SOAP protocol, high-performance applications often use the REST protocol instead.

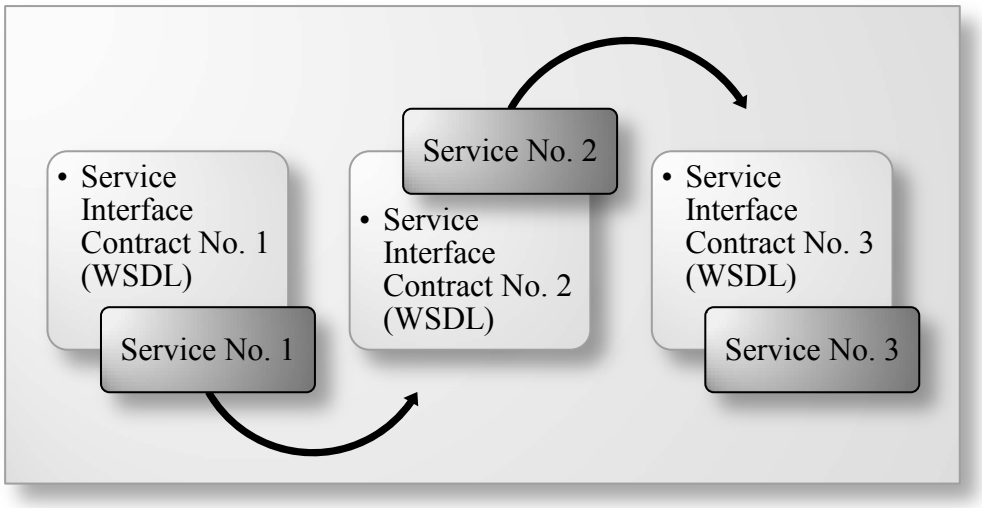


Figure 3. Example of a Service-Oriented Architecture

A simple example of a Service-Oriented Architecture is shown in Figure 3. Note that the first service uses the methods of the second service defined in its interface contract, and that the second service uses the contracted methods of the third service. Now, the notion of change in the context of change management describes the change of the interface contract of a service resulting from a natural evolution of that service.

For example, let the third service be in charge of storing and retrieving mathematical theorems by a uniform theorem identification number (TIN). The interface contract of this service allows for adding, deleting and retrieving theorems by a TIN. When theorems need to be changed, for example because of a correction or generalization, the interface requires deleting the old theorem and adding the new one. The deletion may trigger unintended side-effects and thus result in unnecessary processing.

A natural evolution of the third service would be the addition of interface methods that support the update of premises and conclusions of a theorem. This would result in a new service interface contract for the third service that is in this case fortunately backward compatible. The problem of *change management* is that the system does not automatically profit from the improvement of the third service. Indeed, we have to adapt the implementation of the second service to take advantage of the new methods offered by the third service. This may in turn require modifications of the interface of the second service, hence resulting in a chain of changes to service implementations and interface contracts.

Originally designed as a way to handle complex, ill-defined problems, where the solution is the sum of its parts, we will discuss whether a *Blackboard Architecture* could solve our problem of *change management*. This architecture consists (i) of the *Blackboard*, a common place to store the existing knowledge in the system, (ii) of several *knowledge sources*, workers that update the blackboard with a partial solution when its internal constraints match the blackboard state, and (iii) of a *controlling shell* which controls the flow of problem-solving activity and organizes the knowledge sources.

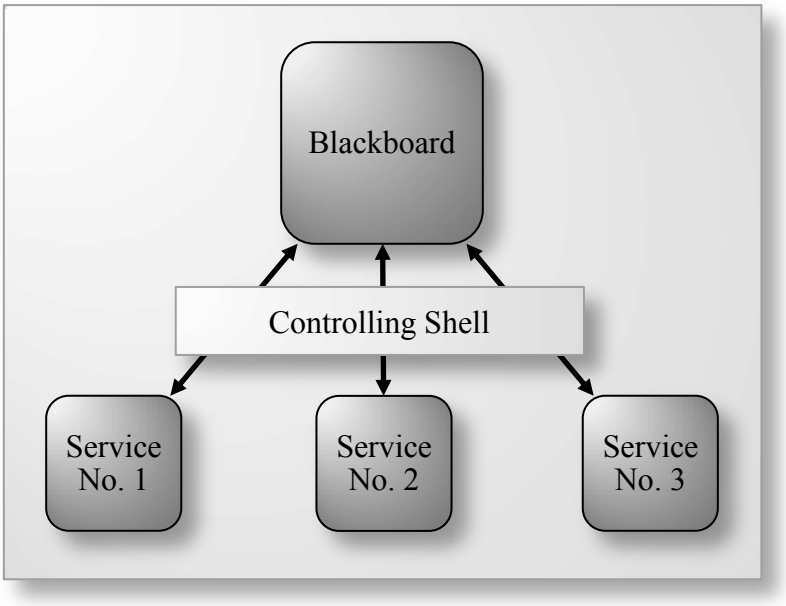


Figure 4. Example of a Blackboard Architecture

Figure 4 shows our scenario in a Blackboard model, where the SOA services form the knowledge sources. The explicit service composition of the SOA model is embedded in the controlling shell of the Blackboard model, resulting in general in a processing overhead for the service organization in large systems.

Now with respect to the problem of change management we notice that we can easily exchange the knowledge source of the third service by a more evolved variant of that service. The main reason is that the services do not depend anymore on specific interfaces of other services. Indeed, they only rely on a uniform knowledge representation on the blackboard. In this architecture, the services react to changes of the knowledge on the blackboard instead of interface method calls.

When we try to apply the Blackboard model as a global solution to SOA we face the following problems:

1) Designing a uniform knowledge representation, which is supported by all services, is a difficult task and may lead to unexpected correlations.
2) Service composition is encapsulated and embedded in the controlling shell of the Blackboard. This design results in a single point of failure.
3) A monolithic design may result in frequent changes of the uniform knowledge representation or the controlling shell, which in turn requires frequent adaptation of the services.

Because of these problems with using a global Blackboard, we propose to use a separate Blackboard between every pair of connected services. For this purpose, we introduce the notion of a *Change-Oriented Architecture* (COA) as an extension of the *Service-Oriented Architecture* (SOA). Instead of composing services by binding one service to the interface contract of another service, the COA uses a *mediation module* to loosely couple two components. Figure 5 shows our scenario in a *Change-Oriented Architecture*.

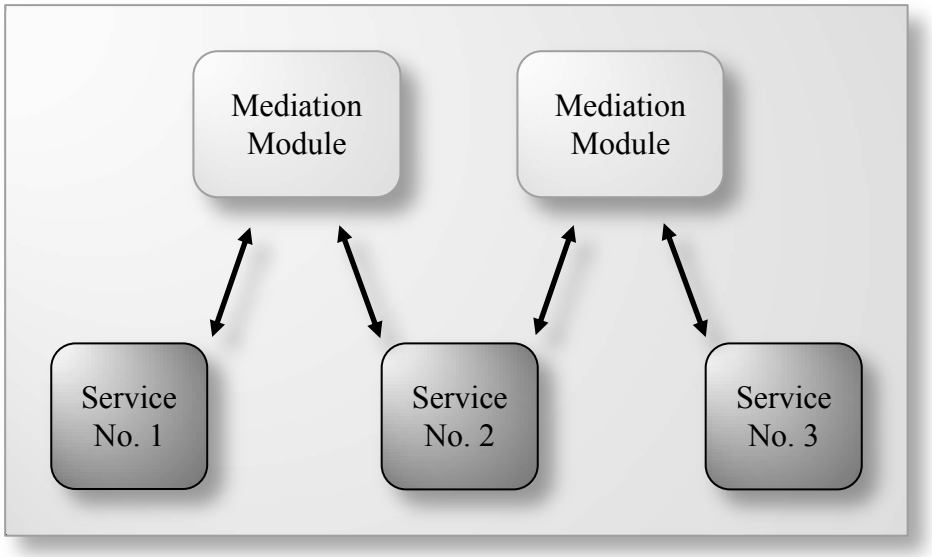


Figure 5. Example of a Change-Oriented Architecture

Before we will introduce the *mediation module* as a special form of a Blackboard, we will discuss how the connection between two services can be established via a Blackboard:

1) Define a uniform knowledge representation (document) for both services.
2) Map knowledge changes (document changes) to existing interface methods.

In comparison to the global Blackboard approach, the COA approach solves the second problem because existing service compositions are no longer hidden but replaced by a *mediation module*. Thus we do not introduce a single point of failure like a single Blackboard. Furthermore, the COA solves the third problem because it replaces the tight connection between two components by a mediation module. A service component no longer directly depends on specific interface methods of a connected component. If the granularity of the interface of a component changes in a COA, a connected component does not need to be adapted in order to profit from the improved interface. This is automatically achieved by computing component-specific change descriptions.

Although this is already an acceptable solution, it does not solve the first problem because it requires a uniform knowledge representation. Indeed, when connecting service components one usually faces in practice the problem that they employ different representations and that they only need to exchange a subset of their information. Forcing a uniform knowledge representation may propagate through all service components and finally lead to a complex bloated pseudo-standard which continuously increases the effort needed to connect new service components, to say the least.

In order to solve this problem in a COA, we do not require a uniform knowledge representation for all services, indeed this representation can be defined differently for each service, even differently for each pair of connected services. For the purpose of translating between different representations back and forth, we will introduce a formalism for automatically invertible grammars in Chapter 6. Thus, if the representation of a service component changes, we only have to adapt the bidirectional transformation grammar in the *mediation module*. A new service component is integrated by defining an invertible transformation between its own knowledge representation and the representation of the connected existing service component. Altogether, the connection between two components can be established by a *mediation module* as follows:

1) Specify a knowledge representation (document) for each service.
2) Define a bidirectional transformation between both knowledge representations.
3) Map knowledge changes (document changes) to existing interface methods.

By allowing multiple interface documents, a classic SOA request can be modeled as a supplementary interface document and a response as a change to this interface document. Thus, a service component may offer both interface models in parallel, and a classic SOA can be iteratively transformed to a COA while remaining backward compatible.

A *mediation module* uses essentially the Blackboard design pattern extended by methods for the computation of semantic document changes and the bidirectional transformation of documents. The main differences to the classic Blackboard design pattern are:

- An interface document exists in two representations, one for each connected service component. We call these representations the *types* of a document. The mediation module has both document types in parallel under revision control. Therefore, the *location* of a document, its URI in the mediation module's Blackboard, needs first to be setup by a bidirectional transformation grammar, which is then used to transform one type of a document into the other. The revision control is additionally helpful for integration testing but if space is critical, one can employ a method that "forgets" old inactive versions.
- When a new version of a document is sent to the mediation module, either as a valid edit script or as a whole document, then the integrity of the document is checked, the document is transformed into the other type, and again the integrity is checked. Only if all these steps succeed, the new version is accepted by the Blackboard and committed to the revision control.
- In contrast to revision control systems like CVS, Subversion or Git, the mediation module returns upon an update request the optimal changes instead of the complete modified document. To compute the optimal changes we take structural properties like the order relevance of subtrees into account, as well as information provided by a service component about weights for removing or adding subtrees and the granularity of changes supported by that component.
- The task of the controlling shell is to govern the described behavior of the mediation module and to guarantee a read-write lock for the Blackboard such that all write operations are handled as transactions: atomic, consistent, isolated and durable.

The interface methods of a *mediation module* are shown in Table 7. In a corresponding strongly typed WSDL description for a general interface to a *mediation module*, we use the base type *string* for documents and change descriptions. We will illustrate the interface methods and the behavior of the *mediation module* in the following use case.

Method	Arguments	Results
Setup	[uri, grammar, specs]	= [types]
Init	[uri, type, doc]	= [version]
Info	[uri]	= [kind, version]
List	[uri]	= [uris]
Checkout	[uri, type, version]	= [doc]
Commit	[uri, type, version, doc]	= [version]
Patch	[uri, type, version, diff]	= [version]
Update	[uri, type, version, spec]	= [diff, version]
Subscribe	[uri, type, call-back]	= [success]
Remove	[uri]	= [success]
State	[uri]	= [grammar, specs]
Change	[uri, grammar, specs, type]	= [success]

Table 7. Interface Protocol of the Mediation Module

Use Case. First of all, the location for a new interface document is setup in the mediation module using the **Setup** method by providing the *invertible grammar* for the bidirectional transformation between the two document types, and the *change specification* for each document type, which guides the change computation and defines integrity constraints. The method returns the two document types that just have been setup.

Then, the first service component initializes the document by calling the **Init** method with the location, the type and the content of the document. The mediation module checks the integrity of the document and transforms the document to the second type using the invertible grammar. The integrity is checked and the method returns the initial version.

The second service component can browse through the repository using the **List** method and it can obtain information about a location using the **Info** method, which returns the kind (document/folder) and the current version number. The second service component retrieves the document by calling the **Checkout** method with the location, the desired type, and the desired initial version number. The method returns the initial document content in the document type of the second service component.

Since this service component wants to be informed about changes instead of regularly checking for updates, the component calls the **Subscribe** method with the location, the type and a call-back. The mediation module registers the call-back. Whenever a document of the given type at the given location or its sub-locations is modified, removed or initialized, this component is notified by the mediation module using the call-back.

Now, the first service component commits a modified version of the document by calling the **Commit** method with the location, the type and the modified content of the document. The mediation module checks the integrity of the document and transforms the document to the second type using the bidirectional transformation grammar. The method returns the new version of the document.

Since the document of the second type changed, the mediation module notifies the second service component using the call-back. This component retrieves the changes of the document by calling the **`Update`** method with the location, the desired type, its current local version of the document and a specification which includes information to guide the change computation. This specification adds additional weights to parts of the document indicating internal dependencies. Furthermore, the specification defines the granularity of changes that this service component is able to process. The mediation module returns a valid edit script that is optimal with respect to the given specification, as well as the current version of the document. The second service component then merges the returned edit script with its local changes and processes the remaining edit script, for example by calling its corresponding SOA interface methods.

Now we assume the second service component performs some background processing and computes the result as an edit script to be applied to the interface document. The component calls the **`Patch`** method with the location, the type, the current version and the valid edit script. The mediation module checks the validity of the edit script, applies the script and checks the integrity of the document. Finally, the new document is transformed to the first document type using the bidirectional transformation grammar and checked for integrity. The method returns the new version of the document.

By calling **`Update`**, the first service component is in turn able to retrieve the changes as a valid edit script which is optimal with respect to the specification provided by this component. And the cycle continues. As the system evolves, there may be the need to adapt the grammar and change specification. For this purpose, the method **`State`** allows to retrieve the current settings and **`Change`** allows to define a new invertible grammar, new change specifications and the type of the source interface document from which the corresponding target interface document is then retranslated automatically.

The *mediation module* relies on the following concepts, methods and techniques:

1) A notion of optimality for edit scripts that respects the semantics of the document structure as well as the change weights and granularity provided by the consumer of the edit script, a service component. (see Chapter 4)
2) An efficient method to compute an optimal edit script with respect to the given change specification. (see Chapter 5)
3) A formalism for bidirectional transformation grammars which are automatically invertible and support incremental processing. (see Chapter 6)

The *mediation module* is realized using standard software design patterns and can in turn be used to implement design patterns. In the following, we will give an overview of relevant *structural*, *behavioral* and *concurrency patterns*. The *italic* definitions of these design patterns are reproduced from [Gamma *et al*, 1994].

Adapter Pattern. *An adapter converts the interface of a service component into an interface that clients expect.* In the mediation module, we use the bidirectional transformation grammar as an adapter between the different document types required by the components. Thus, the mediation module automatically translates between both document types.

Bridge Pattern. *A bridge decouples an abstraction from its implementation so that the two can vary independently.* In the mediation module, the optimal edit scripts obtained by an `Update` method are the abstract interface of a component. The edit operations are then converted by the component to calls of specific implemented methods. Since the granularity of the edit operations can be specified by the component, an independent improvement of the implementation is always possible.

Decorator Pattern. *A decorator attaches additional responsibilities to an object dynamically although keeping the same interface.* So far we only defined the document model of the mediation module, but we have not restricted the kind of knowledge that can be represented in an interface document. This allows for example to represent method execution requests as annotations to parts of the document, or to represent results or feedback in the interface document.

Façade Pattern. *A façade provides a uniform interface to a set of interfaces in a subsystem.* This pattern can be realized with the COA as shown for example in Figure 6.

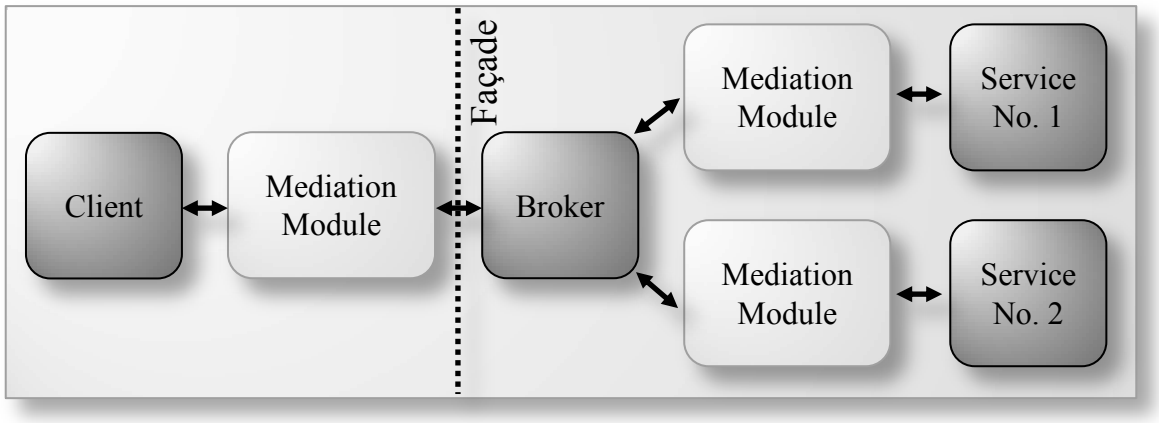


Figure 6. Façade Pattern with Mediation Modules

Chain of Responsibility Pattern. *A chain of responsibility avoids coupling the sender of a request to its receiver by giving more than one component a chance to handle the request.* This pattern can be realized with the COA as shown for example in Figure 7.

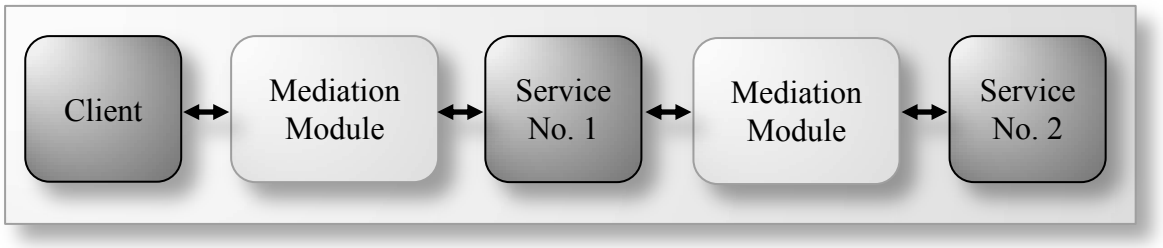


Figure 7. Chain of Responsibility Pattern with Mediation Modules

Command Pattern. *A command encapsulates a request as an object.* A method execution request can also be a first-class citizen of the interface document, for example by designing a controlled language for writing such requests.

Mediator Pattern. *A mediator is a component that encapsulates how a set of components interact by promoting loose coupling and keeping components from referring to each other explicitly.* As the name already indicates, the mediation module acts in fact as a mediator between the components of a COA.

Memento Pattern. *A memento captures and externalizes the internal state of a component such that the component can be restored to this state later.* In the mediation module, the interface document types can be used as a memento for the corresponding component. Whether the state of a component can completely be restored or not depends on how much information about the internal state is actually being represented in the interface document.

Observer Pattern. *An observer defines a one-to-many dependency between objects so that when one object changes its state, all dependent objects are notified automatically.* In the mediation module, a component is allowed to subscribe to locations in the shared repository. This component is then automatically notified when a document is added, removed or modified at a subscribed location or sub-location.

Read-Write Lock Pattern. *A read-write-lock allows concurrent read access to an object but requires exclusive access for write operations.* The mediation module itself has a read-write lock such that all write operations are handled as transactions: atomic, consistent, isolated and durable.

3.5 Discussion

So far, we presented the foundations and the principle concepts of the *Change-Oriented Architecture* (COA). The motivation for this architecture is the problem of *change management* within a *Service-Oriented Architecture* (SOA). Since services are composed by calling interface methods of another service, the composition needs to be adapted when the interface contract of a service is significantly changed or extended. Otherwise, the system is either broken or does not profit from new improved interface methods of the services in use. The problem of *change management* occurs very often in a fast-pace software development environment, where a prototype needs to be rapidly developed, or where frequent changes in the interface of the services are expected. In these situations, we propose to use a COA approach until the interfaces are mature.

To establish a connection between two services in a COA, we use a *mediation module*. This module manages the interface documents in a revision control style. Instead of calling interface methods of another service, a service changes an interface document in the mediation module. This new interface document is then transformed to the document type of the other connected service. Then, this service retrieves an optimal change description and reacts appropriately, for example by adding a result to the interface document.

The *Change-Oriented Architecture* is an *agile* extension of SOA because it sets the focus on the exchanged knowledge instead of concrete interface methods. The COA expects changes in services to happen and therefore it allows a service to change the granularity of its interface without the need to adapt all depending services. With a COA, the overall system may profit immediately from new features of its service components at no cost. Furthermore, the interface documents in the mediation modules provide a possibility to monitor the information flow and current state of the system, which may both help in the detection of flaws in the system design and runtime behavior. Additionally, the mediation module can be used as an adapter to translate between services that use different representations for their exchanged knowledge.

Besides all the benefits of a *Change-Oriented Architecture*, we have also to discuss the disadvantages of this architecture and whether this architecture introduces new threats and pitfalls for the software development process.

Trade-Off. Since the COA replaces direct method calls by interface documents in a mediation module whose changes trigger method execution, we essentially add the step of change computation. This is the trade-off of COA: We sacrifice some performance to achieve a loose coupling of service components. Note that a mixed approach of classical SOA method calls and COA mediation modules is possible in the same system.

Conflict Resolution. Everybody who used a revision control system knows that sooner or later conflicts occur between the local state of a document and its state in the repository. Does the COA introduce new possibilities for conflicts? The behavioral difference between COA and SOA is that a service component does not directly call the methods of another service component which in turn could throw an exception in case of a conflict. Indeed, the COA requires the first service component to place a request for action in the interface document. When the second service component retrieves the request, it detects the conflict and may in turn place an error message as feedback in the interface document. Thus, the difference between COA and SOA with respect to conflicts is that the detection of conflicts is deferred.

Transformation Integrity. The mediation module translates between interface documents of different types using a bidirectional transformation grammar. Thereby, the edit script for an interface document is only accepted if (1) the integrity of the patched interface document can be checked, for example by a schema validation (DTD, XSD, RelaxNG), if (2) this new interface document can be transformed to the other type, and if (3) the integrity of the transformed document can be checked. Assuming that a service component is able to guarantee property (1), we cannot guarantee the properties (2) and (3) in general because the transformation grammar may use value-passing rules. On the one hand, we should verify that the grammar generates defaults that satisfy the integrity constraints. On the other hand, the service components should be implemented in such a way that they recover gracefully from a failed patch attempt.

Lock Prevention. Since the COA introduces mediation modules which implement a read-write lock, there is a risk to run in a *deadlock* situation if some components intermix their write access to connecting mediation modules. This risk can be eliminated if the components clearly separate their connection to mediation modules. The advantage of the interface document in a mediation module is that the information flow and state of the system is made explicit. By analyzing the changes in the mediation modules we may successfully detect *livelock* situations. A livelock is similar to a deadlock, except that the states of the interface document in a livelock continually change in a cyclic manner.

The *mediation module* provides means to compute optimal edit scripts as a change description between two interface documents as well as an automatic translation between different types of interface documents. In the following chapters, we will introduce the required concepts and techniques: a specification for the optimality of edit scripts, an efficient method for computing optimal edit scripts, and an invertible grammar formalism.

4 Semantic Changes

In this chapter, we will focus on the aspects that have to be taken into account by a mediation module for computing an optimal edit script between two versions of an interface document. These aspects include the semantics of an interface document, but also the requirements for the service component that processes the computed changes. This service component may for example weigh some parts of the document differently than others. We will introduce specifications to declaratively define these aspects and thus allow for a component- and document-specific notion of optimal change descriptions.

The first step towards a change description between two interface documents is the identification of corresponding subtrees in the documents. Since the interface document is a partial serialization of the state or knowledge of a service component, it is reasonable to assume that subtrees can be identified by specific keys like a uniform identifier or characteristic attributes. Furthermore, the order of some subtrees may not be relevant in all cases. We will introduce a specification for defining this *semantic similarity* of documents. In general, more than one subtree can be semantically similar to a given subtree.

The second step is the definition of an acceptable solution, a change description which patches the first document to a state that is semantically equal to the second document. When should two documents be considered semantically equal? Where the semantic similarity requires just the similarity of specific key subtrees, the semantic equality requires the equality of all subtrees. Of course, there may exist more than one valid edit script to achieve the *semantic equality* of two documents.

Although these solutions are all valid, we need to take into account the needs of the consumer, the service component which is reacting on the changes of the interface document. The insertion, deletion or modification of subtrees in the interface document may have effects on depending parts of the internal knowledge of that service component. It is important that these effects are taken into account by providing a specification for the *edit weights* of subtrees in the interface document. Thus, an optimal solution is a valid edit script with minimal *edit costs*.

Finally, we want to support the evolution of the interface of a service component. Therefore, the component needs means to specify the granularity of change descriptions that it currently supports. This granularity limits the depth of the edit operations, thus it has the nice side-effect that the solution space for the computation of changes is reduced. Furthermore, the service component does not have to recalculate the edit script before processing. With a specification for the *edit granularity*, we are able to reduce the search space for computing an optimal solution at an adequate level of granularity.

4.1 Semantic Equality

To compute the change description between two interface documents, we have to identify corresponding subtrees in these documents. The following *similarity specification* represents the configuration for the semantic equality and semantic similarity of two labeled trees. It consists of two components: the *similarity order* and the *similarity keys*. The idea of taking the order of elements into account originates from [Radzevich, 2006] where six semantic equivalence classes for XML documents have been introduced. In the following, we present a generalization of this work to labeled trees.

The *similarity order* defines for all layers of a labeled tree whether their order is relevant for the semantic similarity of that labeled tree. For example, the order of axioms in a mathematical theory is not relevant for a service component that verifies proofs. Since we define the similarity order over the label of a subtree, it follows that equally labeled subtrees have an equal similarity order.

Definition 4.1.1 (Similarity Order): Let D be a labeled tree with the label $l_d = L(D)$, let $l \in \mathcal{L}_A$ be an edge label and let $\mathcal{C}_l(D)$ be a layer of D. The *similarity order* $\Sigma_O(l_d, l) \in \{\top, \bot\}$ is a Boolean value which indicates whether the order of the subtrees in the layer $\mathcal{C}_l(D)$ is relevant ($\top$) for the semantic similarity of the labeled tree D (when comparing it to other labeled trees) or not relevant ($\bot$).

The *similarity keys* define for all labeled trees and all layers which subtrees have to be compared to identify semantically similar labeled trees. For example, similarity keys are usually unique ids, names or specific attributes. Since we define the similarity keys over the label of a subtree, it follows that equally labeled subtrees have an equal set of similarity keys.

Definition 4.1.2 (Similarity Keys): Let D be a labeled tree with the label $l_d = L(D)$, let $l \in \mathcal{L}_A$ be an edge label and let $\mathcal{C}_l(D)$ be a layer of D. The *similarity keys* $\Sigma_K(l_d, l) \in \mathcal{P}(\mathcal{L}_V)$ are a set of node labels indicating which subtrees in the layer $\mathcal{C}_l(D)$ are relevant for the similarity of the labeled tree D (when comparing it to other labeled trees). Thereby, all subtrees having a label of the set of similarity keys are relevant.

We define an instance of the similarity specification as follows.

Definition 4.1.3 (Similarity Specification): A *similarity specification* $\Sigma_S = (\Sigma_O, \Sigma_K)$ is a pair of one similarity order Σ_O and similarity keys Σ_K.

Having introduced the *similarity specification*, we now define different notions of *similarity mappings* between labeled trees before we finally define the notion of *semantic equality* for labeled trees. Identifying similar subtrees means establishing a mapping between sequences of labeled trees that satisfies a *mapping condition*.

Notation 4.1.4 (Mapping Condition): Let V_1, V_2 be sequences of labeled trees, let f be a (bijective) multi-mapping from V_1 to V_2, and let P be a predicate over pairs of labeled trees. We define the *mapping condition* as the following shortcut predicate $\phi(f, P)$ which indicates whether all elements of the multi-mapping f satisfy the condition P.

$$\phi(f, P) \ :\Leftrightarrow \forall (x, y) \in f.\ P(x, y)$$

Thereby, the function f is a bijective multi-mapping between corresponding labeled trees in the sequences V_1 and V_2, where the correspondence is defined by the predicate P.

The set of mappings between sequences of labeled trees, that satisfy the mapping condition, is called *matching mappings*.

Definition 4.1.5 (Matching Mappings): Let V_1, V_2 be sequences of labeled trees and let P be a predicate over pairs of labeled trees. We define the set Ω of *matching mappings* between sequences of labeled trees as follows:

$$\Omega^{\mathrm{P}}_{V_1 \leftrightarrow V_2} := \left\{ f \in \mathfrak{F}_{V_1 \leftrightarrow V_2} \middle| \phi(f, P) \right\}$$

The matching mappings are defined with respect to a context $\tau = (\varphi, C)$ where φ is a Boolean value indicating the relevance of the order and where the sequence $C \subseteq V_1$ contains the elements of V_1 which can be mapped to multiple elements of V_2. We omit the context τ in the notation when it can be inferred.

The set Ω contains all bijective multi-functions $f \in \mathfrak{F}_{V_1 \leftrightarrow V_2}$ which respect the order ($\varphi = \top$) or not ($\varphi = \bot$) between corresponding labeled trees in V_1 and V_2, where the correspondence is defined by the predicate P. The sequence C contains all elements of V_1 which may have multiple mapping partners. The multi-functions will be relevant when we introduce document variables for the grammar formalism. For now, we define $C := \emptyset$ as the default unless explicitly specified. Then, the set $\Omega^{\mathrm{P}}_{V_1 \leftrightarrow V_2}$ contains only bijective functions without multiple mappings. For example, let $V_1 = [A, B, C]$ and $V_2 = [A, C, B]$ be sequences of labeled trees without children with $L(A) = "A"$, $L(B) = "B"$ and $L(C) = "C"$. In the ordered case $\tau = (\top, \emptyset)$, the set $\Omega^{\approx}_{V_1 \leftrightarrow V_2} = \emptyset$ is empty because of $\neg(B \approx C)$. In the unordered case $\tau' = (\bot, \emptyset)$, the set $\Omega^{\approx}_{V_1 \leftrightarrow V_2} = \{f_1\}$ contains $f_1 :=$

$\{((1,A),(1,A)),((2,B),(3,B)),((3,C),(2,C))\}$. Let $V_3 = V_4 = [E]$ be a sequence of a tree without children with $L(E) = "E"$. Then we have $\Omega^{\approx}_{V_3 \leftrightarrow V_4} = \{f_2\}$ with $f_2 \coloneqq \{((1,E),(1,E))\}$ in both the ordered and the unordered case.

Now we need to lift the notion of mappings from sequences to the level of whole trees. The set of *tree matching mappings* contains the combinations of all matching mappings between the children in the layers of two labeled trees.

Definition 4.1.6 (Tree Matching Mappings): Let D_1, D_2 be equally labeled trees with $L(D_1) = L(D_2) = l_0$ and let P be a predicate over pairs of labeled trees. Furthermore, let $V_{l,1} \coloneqq \mathcal{C}_l(D_1)$ and $V_{l,2} \coloneqq \mathcal{C}_l(D_2)$ be the children in the layer l of the trees D_1 and D_2. We define the set $\mathcal{T}$ of *tree matching mappings* as follows:

$$\mathcal{T}^{P}_{(D_1,D_2)} \coloneqq \left\{ \bigcup_{l \in \mathcal{L}_A} f_l \;\middle|\; f_l \in \Omega^{P}_{V_{l,1} \leftrightarrow V_{l,2}} \right\}$$

The tree matching mappings are defined with respect to a context $\kappa = (\Sigma_S, \Lambda)$ where $\Sigma_S = (\Sigma_O, \Sigma_K)$ is a similarity specification, and Λ is a function from labeled trees to Boolean values. Then the context $\tau_l = (\varphi_l, C_l)$ for Ω is computed for every layer l by $\varphi_l \coloneqq \Sigma_O(l_0, l)$ and $C_l \coloneqq \{x \in V_{l,1} | \Lambda(x)\}$. Hence, the function Λ indicates which subtrees of D_1 may match multiple subtrees of D_2. We define $\Lambda_\perp$ to assign $\perp$ to every labeled tree such that $C_l \coloneqq \emptyset$ for every layer l. By default we assume a context $\kappa = (\Sigma_S, \Lambda_\perp)$. We omit the context in the notation when it can be inferred. The tree matching mappings are all bijective multi-functions between corresponding children of the labeled trees D_1 and D_2, respecting the subtree partitioning by the layers of the tree and their order.

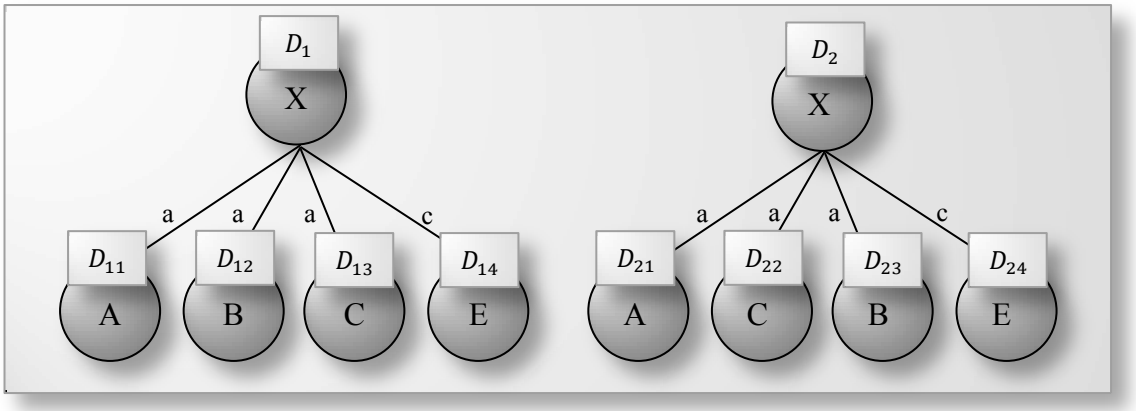


Figure 8. Example for tree matching mappings

Continuing our example from *Definition 4.1.5*, we define the labeled trees D_1 and D_2 shown in Figure 8 with $\mathcal{C}_a(D_1) \coloneqq V_1$, $\mathcal{C}_a(D_2) \coloneqq V_2$, $\mathcal{C}_c(D_1) \coloneqq V_3$ and $\mathcal{C}_c(D_2) \coloneqq V_4$. The tree matching mappings $\mathcal{T}^{\approx}_{(D_1,D_2)}$ are empty if Σ_S defines $\varphi_a = \top$, because of $\Omega^{\approx}_{V_1 \leftrightarrow V_2} = \emptyset$. If Σ_S defines $\varphi_a = \bot$, we have $\mathcal{T}^{\approx}_{(D_1,D_2)} = \{f_1 \cup f_2\}$ independent of the similarity order φ_c.

This allows us to define that two labeled trees are *semantically equal* if they are equally labeled and if there exists a tree matching mapping w.r.t. the similarity specification.

Definition 4.1.7 (Semantic Equality): Let $\Sigma_\mathrm{S} = (\Sigma_\mathrm{O}, \Sigma_\mathrm{K})$ be a similarity specification and D_1, D_2 be labeled trees. The *semantic equality* of the labeled trees D_1 and D_2 with respect to Σ_S, denoted by $D_1 =_{\Sigma_\mathrm{S}} D_2$, is a predicate over pairs of labeled trees defined as follows in the context of $\kappa = (\Sigma_\mathrm{S}, \Lambda_\bot)$:

$$D_1 =_{\Sigma_\mathrm{S}} D_2 \;\; :\Leftrightarrow \left((D_1 \approx D_2) \wedge \left|\mathcal{T}^{=_{\Sigma_\mathrm{S}}}_{(D_1,D_2)}\right| > 0\right)$$

Intuitively, two labeled trees are semantically equal if they have the same label and their layers are pairwise semantically equal. Two equally labeled layers are semantically equal if there exists a mapping between the children matching the semantic equality conditions.

The evaluation of this recursive definition terminates for concrete labeled trees because they have a finite amount of nodes and a finite amount of subtrees. The leaves of the evaluation tree are comparisons between subtrees which do not have any children in a specific layer. The tree key matching mappings contains in this case the empty mapping.

For the design of the algorithm for change computation between labeled trees, it is important to show that the semantic equality of labeled trees is an equivalence relation.

Lemma 4.1.8 (Semantic Equality is an Equivalence Relation): Let $\Sigma_\mathrm{S} = (\Sigma_\mathrm{O}, \Sigma_\mathrm{K})$ be a similarity specification. The *semantic equality* of labeled trees $=_{\Sigma_\mathrm{S}}$ is an equivalence relation on the set of labeled trees $\mathcal{D}$ in the context of $\kappa = (\Sigma_\mathrm{S}, \Lambda_\bot)$.

Proof: We have to show the reflexivity, symmetry and transitivity of $=_{\Sigma_\mathrm{S}}$.

(1) *Reflexivity* : $\forall D \in \mathcal{D}.\, (D =_{\Sigma_\mathrm{S}} D)$

We have to show that $D \approx D$ and $\left|\mathcal{T}^{=_{\Sigma_\mathrm{S}}}_{(D,D)}\right| > 0$ hold. Clearly, it holds that $D \approx D$ because $L(D) = L(D)$. For all $l \in \mathcal{L}_A$ let $\varphi_l \coloneqq \Sigma_O(L(D), l)$ be the relevance of the order of the layer l in D. Let $V_l = \mathcal{C}_l(D)$ be the direct subtree layer l of D, the statement $\left|\mathcal{T}^{=_{\Sigma_\mathrm{S}}}_{(D,D)}\right| > 0$ holds if $\forall l \in \mathcal{L}_A.\, \exists f_l \in \Omega^{=_{\Sigma_\mathrm{S}}}_{V_l \leftrightarrow V_l}$ with the layer specific context $\tau = (\varphi_l, \emptyset)$. By induction over the structure of the tree we will show that $\forall l \in \mathcal{L}_A.\, \exists f \in \mathfrak{F}_{V_l \leftrightarrow V_l}.\, \phi(f, =_{\Sigma_S})$.

(*Base Case*) Let D be a labeled tree with no children in all layers, hence it holds that $\forall l \in \mathcal{L}_A.|V_l| = 0$. Then the set $\mathfrak{F}_{V_l \leftrightarrow V_l}$ contains only the empty mapping f_0 which satisfies the mapping condition $\phi(f_0, =_{\Sigma_S})$. Thus, it holds that $D =_{\Sigma_S} D$.
(*Step Case*) Let D be a labeled tree with at least one child, thus $\exists l \in \mathcal{L}_A.|V_l| > 0$. Then the set $\mathfrak{F}_{V_l \leftrightarrow V_l}$ contains either the empty mapping f_0 if the layer has no children or at least the identity mapping f_{id} defined by $\forall x \in V_l.f_{id}(x) = x$ if the layer has some children. Thereby the identity mapping satisfies the mapping condition $\phi(f_{id}, =_{\Sigma_S})$, that is $\forall (x,y) \in f_{id}.\ x =_{\Sigma_S} y$, because of the induction hypothesis. Thus, it holds that $D =_{\Sigma_S} D$.

(2) *Symmetry* : $\forall D_1, D_2 \in \mathcal{D}.\big((D_1 =_{\Sigma_S} D_2) \Rightarrow (D_2 =_{\Sigma_S} D_1)\big)$

We have to show that $D_2 \approx D_1$ and $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_2,D_1)}\right| > 0$ follows from $D_1 =_{\Sigma_S} D_2$. We have $D_1 \approx D_2$, hence it holds that $L(D_1) = L(D_2) = l_0$ and $D_2 \approx D_1$. For all $l \in \mathcal{L}_A$ let $\varphi_l := \Sigma_O(l_0, l)$ be the relevance of the order of the layer l in D_1 and D_2. Let $V_{l,1} = \mathcal{C}_l(D_1)$ and $V_{l,2} = \mathcal{C}_l(D_2)$ be the direct subtree layers l in D_1 and D_2. We have to show that $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_1,D_2)}\right| > 0$ implies $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_2,D_1)}\right| > 0$, thus that for all $l \in \mathcal{L}_A$ it follows that $\exists f_l \in \Omega^{=_{\Sigma_S}}_{V_{l,1} \leftrightarrow V_{l,2}}$ implies $\exists f_l \in \Omega^{=_{\Sigma_S}}_{V_{l,2} \leftrightarrow V_{l,1}}$ with the layer specific context $\tau = (\varphi_l, \emptyset)$. By induction over the structure of the tree we will show that $\exists f_1 \in \mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,2}}.\phi(f_1, =_{\Sigma_S})$ implies $\exists f_2 \in \mathfrak{F}_{V_{l,2} \leftrightarrow V_{l,1}}.\phi(f_2, =_{\Sigma_S})$ for all $l \in \mathcal{L}_A$.
(*Base Case*) Let D_1 and D_2 be labeled trees without children, thus $\forall l \in \mathcal{L}_A.|V_{l,1}| = |V_{l,2}| = 0$. Then the set $\mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,2}}$ contains only the empty mapping f_0 which satisfies the mapping condition $\phi(f_0, =_{\Sigma_S})$ and which is also a member of $\mathfrak{F}_{V_{l,2} \leftrightarrow V_{l,1}}$. Thus, it holds that $D_2 =_{\Sigma_S} D_1$.
(*Step Case*) Let D_1 or D_2 be labeled trees with at least one child, thus $\exists l \in \mathcal{L}_A.|V_{l,1}| = |V_{l,2}| > 0$. In the case that the direct tree layer contains no children, the reasoning is analogously to the base case. In the case that the direct tree layer has at least one child, the set $\mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,2}}$ contains by the assumption at least one (bijective) function f_1 that satisfies the mapping condition $\phi(f_1, =_{\Sigma_S})$, that is $\forall (x,y) \in f_1.\ x =_{\Sigma_S} y$. The inverse function f_1^{-1} is a member of $\mathfrak{F}_{V_{l,2} \leftrightarrow V_{l,1}}$ and by the induction hypothesis it satisfies the mapping condition $\phi(f_1^{-1}, =_{\Sigma_S})$, that is $\forall (y,x) \in f_1^{-1}.\ y =_{\Sigma_S} x$. Thus, it holds that $D_2 =_{\Sigma_S} D_1$.

(3) *Transitivity*: $\forall D_1, D_2, D_3 \in \mathcal{D}.\Big(\big((D_1 =_{\Sigma_S} D_2) \wedge (D_2 =_{\Sigma_S} D_3)\big) \Rightarrow (D_1 =_{\Sigma_S} D_3)\Big)$

We have to show that $D_1 \approx D_3$ and $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_1,D_3)}\right| > 0$ follows from $D_1 =_{\Sigma_S} D_2$ and $D_2 =_{\Sigma_S} D_3$. It holds that $D_1 \approx D_2$ and $D_2 \approx D_3$, hence we have $L(D_1) = L(D_2) = L(D_3) = l_0$ and $D_1 \approx D_3$. For all $l \in \mathcal{L}_A$ let $\varphi_l \coloneqq \Sigma_O(l_0, l)$ be the relevance of the order of the layer l in D_1, D_2 and D_3. Let $V_{l,i} = \mathcal{C}_l(D_i)$ be the direct subtree layer l for the tree D_i with $i \in \{1,2,3\}$. We have to show that $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_1,D_2)}\right| > 0$ and $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_2,D_3)}\right| > 0$ imply $\left|\mathcal{T}^{=_{\Sigma_S}}_{(D_1,D_3)}\right| > 0$, thus that for all $l \in \mathcal{L}_A$ it follows that $\exists f_l \in \Omega^{=_{\Sigma_S}}_{V_{l,1} \leftrightarrow V_{l,2}}$ and $\exists f_l \in \Omega^{=_{\Sigma_S}}_{V_{l,2} \leftrightarrow V_{l,3}}$ imply $\exists f_l \in \Omega^{=_{\Sigma_S}}_{V_{l,1} \leftrightarrow V_{l,3}}$. By induction over the structure of the tree we will show that the properties $\exists f_1 \in \mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,2}}.\phi(f_1, =_{\Sigma_S})$ and $\exists f_2 \in \mathfrak{F}_{V_{l,2} \leftrightarrow V_{l,3}}.\phi(f_2, =_{\Sigma_S})$ imply $\exists f_3 \in \mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,3}}.\phi(f_3, =_{\Sigma_S})$ for all $l \in \mathcal{L}_A$.

(*Base Case*) Let D_1, D_2 and D_3 be labeled trees without children, thus $\forall l \in \mathcal{L}_A.|V_{l,1}| = |V_{l,2}| = |V_{l,3}| = 0$. Then the sets $\mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,2}}$ and $\mathfrak{F}_{V_{l,2} \leftrightarrow V_{l,3}}$ contain only the empty mapping f_0 which satisfies the mapping condition $\phi(f_0, =_{\Sigma_S})$. This mapping is also a member of $\mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,3}}$ satisfying the mapping condition. Thus, it holds that $D_1 =_{\Sigma_S} D_3$.

(*Step Case*) Let D_1, D_2 or D_3 be labeled trees with at least one child, thus $\exists l \in \mathcal{L}_A.|V_{l,1}| = |V_{l,2}| = |V_{l,3}| > 0$. In the case that the direct tree layer contains no children, the reasoning is analogously to the base case. In the case that the direct tree layer has at least one child, the set $\mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,2}}$ contains by the assumption at least one (bijective) function f_1 that satisfies the mapping condition $\phi(f_1, =_{\Sigma_S})$, that is $\forall (x, y) \in f_1.\ x =_{\Sigma_S} y$. Furthermore, the set $\mathfrak{F}_{V_{l,2} \leftrightarrow V_{l,3}}$ contains also by the assumption at least one (bijective) function f_2 that satisfies the mapping condition $\phi(f_2, =_{\Sigma_S})$, that is $\forall (x, y) \in f_2.\ x =_{\Sigma_S} y$. The composition $f_3 \coloneqq f_1 \circ f_2$ is a member of $\mathfrak{F}_{V_{l,1} \leftrightarrow V_{l,3}}$ and by the induction hypothesis it satisfies the mapping condition $\phi(f_3, =_{\Sigma_S})$, that is $\forall (x, y) \in f_3.\ x =_{\Sigma_S} y$. Thus, it holds that $D_1 =_{\Sigma_S} D_3$.

■

The *similarity order* and the *similarity key* of subtrees in an interface document can be declaratively configured by a *similarity specification*. This is a static document-specific specification which is usually defined once when setting up the interface document.

4.2 Semantic Similarity

Having defined the semantic equality for labeled trees, we now define semantic similarity of labeled trees. The step from semantic equality to similarity requires only the comparison of the key subtrees in the layers of a labeled tree. Therefore, we restrict the full tree matching mappings to tree key matching mappings as follows.

Definition 4.2.1 (Tree Key Matching Mappings): Let D_1, D_2 be equally labeled trees with $L(D_1) = L(D_2) = l_0$ and let P be a predicate over pairs of labeled trees. Furthermore, let $V_{l,1} \coloneqq \mathcal{C}_l(D_1)$ and $V_{l,2} \coloneqq \mathcal{C}_l(D_2)$ be the children in the layer l of the trees D_1 and D_2, and let $F \coloneqq \Sigma_K(l_0, l)$ be the similarity keys of that layer. Then $K_{l,1} \coloneqq V_{l,1}|F_l$ and $K_{l,2} \coloneqq V_{l,2}|F_l$ are the key children in the layer l of the trees D_1 and D_2. We define the set $\mathcal{K}$ of *tree key matching mappings* as follows:

$$\mathcal{K}^P_{(D_1,D_2)} \coloneqq \left\{ \bigcup_{l \in \mathcal{L}_A} f_l \,\middle|\, f_l \in \Omega^P_{K_{l,1} \leftrightarrow K_{l,2}} \right\}$$

The tree key matching mappings are defined with respect to a context $\kappa = (\Sigma_S, \Lambda)$ where $\Sigma_S = (\Sigma_O, \Sigma_K)$ is a similarity specification, and Λ is a function from labeled trees to Boolean values. Then the context $\tau_l = (\varphi_l, C_l)$ for Ω is computed for every layer l by $\varphi_l \coloneqq \Sigma_O(l_0, l)$ and $C_l \coloneqq \{x \in V_{l,1} | \Lambda(x)\}$. By default, we define Λ as $\Lambda_\perp$ such that $C_l \coloneqq \emptyset$. We omit the context when it can be inferred. The tree key matching mappings are all bijective multi-functions between corresponding subtrees labeled with a similarity key in the children of the labeled trees D_1 and D_2, respecting the subtree partitioning by the layers of the tree and their order. Continuing our example from *Definition 4.1.6*, let $L(D_1) = L(D_2) = l_0$, we define the similarity keys as $\Sigma_K(l_0, a) = \{"B"\}$ and $\Sigma_K(l_0, c) = \emptyset$. Then the tree key matching mappings $\mathcal{K}^{\approx}_{(D_1,D_2)} = \{f_3 \cup \emptyset\}$ contain exactly one mapping with $f_3 \coloneqq \{((2, B), (3, B))\}$, in this case independent of the specific similarity order.

This allows us now to define that two labeled trees are *semantically similar* if they are equally labeled and if there exists a tree key matching mapping.

Definition 4.2.2 (Semantic Similarity): Let $\Sigma_S = (\Sigma_O, \Sigma_K)$ be a similarity specification and let D_1, D_2 be labeled trees. The *semantic similarity* of the labeled trees D_1 and D_2 with respect to Σ_S, denoted by $D_1 \cong_{\Sigma_S} D_2$, is a predicate over pairs of labeled trees and defined as follows in the context of $\kappa = (\Sigma_S, \Lambda_\perp)$:

$$D_1 \cong_{\Sigma_S} D_2 \;:\Leftrightarrow \left((D_1 \approx D_2) \wedge \left|\mathcal{K}^{=\Sigma_S}_{(D_1,D_2)}\right| > 0\right)$$

Intuitively, two labeled trees are semantically similar if they have the same label and their layers are pairwise semantically similar. Two equally labeled layers are semantically similar if there exists a mapping between the similarity keys in these layers that matches the conditions of semantic equality.

In contrast to semantic equality, the semantic similarity requires only a mapping of the key children, not a complete mapping of all children. Analogously to semantic equality, we can state that the semantic similarity of labeled trees is an equivalence relation.

Lemma 4.2.3 (Semantic Similarity is an Equivalence Relation): Let $\Sigma_\mathrm{S} = (\Sigma_\mathrm{O}, \Sigma_\mathrm{K})$ be a similarity specification. The *semantic similarity* of labeled trees $\cong_{\Sigma_\mathrm{S}}$ is an equivalence relation on the set of labeled trees $\mathcal{D}$.

Proof: Analogously to *Lemma 4.1.8*.

■

It is obvious that semantic equality of two labeled trees implies their semantic similarity.

Lemma 4.2.4 (Semantic Equality implies Semantic Similarity): Let D_1 and D_2 be labeled trees and let $\Sigma_\mathrm{S} = (\Sigma_\mathrm{O}, \Sigma_\mathrm{K})$ be a similarity specification. The *semantic equality* $D_1 =_{\Sigma_\mathrm{S}} D_2$ of these labeled trees implies their *semantic similarity* $D_1 \cong_{\Sigma_\mathrm{S}} D_2$.

Proof: From $D_1 =_{\Sigma_\mathrm{S}} D_2$ it follows that $D_1 \approx D_2$ and $L(D_1) = L(D_2) = l_0$. Let $V_{l,1}$ and $V_{l,2}$ be the children in the layer l of the trees D_1 and D_2, and let $K_{l,1}$ and $K_{l,2}$ be the key children. There exists a matching mapping $f_l \in \Omega^{=_{\Sigma_\mathrm{S}}}_{V_{l,1} \leftrightarrow V_{l,2}}$ for all $l \in \mathcal{L}_A$. Therefore, it holds (recursively) that $f_l|_{K_{l,1}} \in \Omega^{\cong_{\Sigma_\mathrm{S}}}_{K_{l,1} \leftrightarrow K_{l,2}}$ and thus we have $D_1 \cong_{\Sigma_\mathrm{S}} D_2$.

■

Having defined the semantic similarity and equality, we are now able to introduce the notion of *change scripts*, which are valid edit scripts which patch a labeled tree into another labeled tree which is semantically equal to the given one.

Definition 4.2.5 (Change Script): Let D_1, D_1' and D_2 be labeled trees and let Σ_S be a similarity specification. The valid edit script Δ is a *change script* for D_1 and D_2 with respect to Σ_S if $(D_1, \Delta) \hookrightarrow_{\mathrm{PATCH}} \mathrm{D}_1'$ and $D_1' =_{\Sigma_\mathrm{S}} D_2$. The set of all change scripts for D_1 and D_2 with respect to Σ_S is denoted by $\mathbb{C}_{\Sigma_S}(D_1, D_2)$.

4.3 Edit Costs

In contrast to a classical revision control system, where an optimal change script is a change script that changes as few nodes as possible in the tree, we are facing different needs for interface documents in the context of the *Change-Oriented Architecture*. Changing some elements in the interface document can result in large dependent modifications inside the interfaced system. By introducing the notion of an *edit weight* for labeled trees, the interfaced service component will be able to specify the collateral costs of adding, deleting or modifying subtrees. These costs are then considered for computing a minimum-cost change script. Let us first introduce the notion of weights.

Definition 4.3.1 (Weights): The well-ordered set of *weights* is the non-empty totally ordered set $\mathcal{W}$ with the well-founded binary relation $\geq$ and the element 0 being the least element of $\mathcal{W}$. Furthermore, let $S(x)$ be the successor of x for all $x \in \mathcal{W}$, then we define the addition + on $\mathcal{W}$ as follows for all $x, y \in \mathcal{W}$.

1) $x + 0 = x$
2) $x + S(y) = S(x + y)$

In the following, we use the natural numbers defined by the standard Peano axioms as the set of weights. Furthermore, we distinguish between delete and insert weights to distinguish the weights of existing labeled trees from the weights of new labeled trees.

The service component can provide for any existing labeled tree a weight that indicates the collateral costs of deleting that labeled tree, for example a weight corresponding to the amount of dependent objects. This defines the delete weight of existing labeled trees.

Definition 4.3.2 (Edit Weight): Let D be an interface document. The *edit weight* $\Sigma_W = (\mathrm{W}_-, \mathrm{W}_+)$ is a pair of a delete weight and an insert weight. The *delete weight* is a total function W_- from the set of all subtrees $\mathcal{S}(D)$ to the weights $\mathcal{W}$. Let D_x be a subtree of D. The weight $\mathrm{W}_-(D_x)$ indicates the collateral costs of deleting the subtree D_x. The *insert weight* is a total function W_+ from the set of all labeled trees $\mathfrak{D}$ to the weights $\mathcal{W}$. Let D_x be a labeled tree, the weight $\mathrm{W}_+(D_x)$ indicates the collateral costs of inserting the labeled tree D_x.

In contrast to all introduced specifications so far, the edit weight is not defined over the label of a subtree but directly over a specific subtree.

The service component has complete freedom in defining the delete and insert weight, collectively called edit weight, for an interface document. However, the edit weight has to satisfy the following consistency axiom.

Axiom 4.3.3 (Edit Weight Consistency): Let Σ_S be a similarity specification and let D_1, D_2 be two labeled trees with $D_1 =_{\Sigma_S} D_2$. Let $V_1 := \mathcal{C}(D_1)$ be the children of the tree D_1. For an edit weight $\Sigma_W = (\mathrm{W}_-, \mathrm{W}_+)$ the following properties always hold:

1) $\mathrm{W}_-(D_1) \geq \sum_{D_x \in V_1} \mathrm{W}_-(D_x)$
2) $\mathrm{W}_+(D_1) \geq \sum_{D_x \in V_1} \mathrm{W}_+(D_x)$
3) $\mathrm{W}_+(D_1) = \mathrm{W}_+(D_2)$

The first two properties guarantee that the delete/insert weight of a labeled tree is always at least as great as the sum of the delete/insert weights of all children of that labeled tree. The last property ensures that semantically equal subtrees have the same insert weight.

This allows us to define the *edit costs of an edit operation* as the sum of the edit weights of the deleted (Ψ_-^δ) and inserted (Ψ_+^δ) labeled trees and the *edit costs of an edit script* as the sum of the edit costs of the contained edit operations.

Definition 4.3.4 (Edit Costs of an Edit Operation): Let $\Sigma_W = (\mathrm{W}_-, \mathrm{W}_+)$ be an edit weight and δ be a valid edit operation for the labeled tree D. The *edit costs of the edit operation* δ, denoted by $\xi(\delta)$, are defined as follows:

$$\xi(\delta) := \sum_{D_k \in \Psi_-^\delta} \mathrm{W}_-(\mathrm{D_k}) \quad + \sum_{D_k \in \Psi_+^\delta} \mathrm{W}_+(\mathrm{D_k})$$

Definition 4.3.5 (Edit Costs of an Edit Script): Let Σ_W be an edit weight and Δ be a valid edit script for the labeled tree D. The *edit costs of the edit script* Δ, denoted by $\xi(\Delta)$, are defined as follows:

$$\xi(\Delta) := \begin{cases} \xi(\delta) + \xi(\Delta') & if\ \Delta = \delta * \Delta' \\ 0 & if\ \Delta = [\,] \end{cases}$$

4.4 Edit Granularity

The interface implementation of a service component is able to handle edit scripts up to a specific level of granularity. In practice, too granular changes, that are too deep changes in the labeled tree, have to be lifted to changes for parent elements such that the adequate interface methods can be called. In order to natively support the computation of edit scripts with an adequate granularity by the mediation module, we introduce the notion of an *edit limitation* for labeled trees. Thus, the service component is able to provide the mediation module a precise specification of its own level of change granularity.

Definition 4.4.1 (Edit Limitation): Let D be an interface document. The *edit limitation* is a total predicate Σ_L from the set of all subtrees $\mathcal{S}(D)$ to Boolean values. Let D_x be a subtree of D. The edit limitation $\Sigma_L(D_x) \in \{\top, \bot\}$ indicates whether the labeled tree should be replaced ($\top$) if at least one change in a subtree is detected or whether the labeled tree should be compared deeply ($\bot$).

Thus the edit limitation allows the service component to restrict precisely the granularity of edit scripts. Note that the edit limitation of a subtree is irrelevant if one of its ancestors is defined to be edit limited. In case of a detected change, the highest edit limited ancestor has to be replaced anyway according to the definition of edit limitation.

As an illustrating example consider an interface document which contains mathematical theories with definitions, axioms, theorems and proofs. An interfaced proof assistant system may have a mechanism for management of change that reacts on changes at the granularity level of whole definitions, whole axioms, whole theorems and single proof steps. Computing for example a detailed change script for a deep modification of an axiom is a waste of time and resources because the management of change mechanism treats this change script anyway as a full replacement of the axiom. We enable change scripts to accommodate to these granularity requirements by using an edit limitation on all definitions, axioms and theorems in the interface document. Thus, a change script would then for example completely replace an axiom if at least one change is detected in this axiom.

The following *edit specification* represents the service-specific configuration for change computation of an interface document that can be sent to the mediation module in order to obtain an optimal change script at an adequate level of change granularity. The edit specification consists of two components which guide the service-specific computation of edit scripts for an interface document: the *edit weight* and the *edit limitation*.

Definition 4.4.2 (Edit Specification): An *edit specification* $\Sigma_{\mathrm{E}} = (\Sigma_W, \Sigma_L)$ is a pair of one edit weight Σ_W and one edit limitation Σ_L.

The notion of change script needs now to be adapted to the edit weight and the edit limitation. Furthermore, the current definition of change script allows for changing a subtree of a labeled tree into one that is no longer semantically similar. At first sight, this might not be an issue, but let us analyze the drawbacks for the example shown in Table 8.

```
\theory{}[                          \theory{}[
 \theorem{}[                         \theorem{}[
  \name{}[ "XY" ],                    \name{}[ "YZ" ],
  \conj{}[ "#2" ]],                   \conj{}[ "#2" ],
 \theorem{}[                         \theorem{}[
  \name{}[ "YZ" ],                    \name{}[ "XY" ],
  \conj{}[ "#3" ]]]                   \conj{}[ "#3" ]]]
```

Table 8. Comparing two formal documents D_1 (on the left) and D_2 (on the right)

With the names of theorems being their similarity keys, a potential change script may be the following valid edit script, which simply replaces the similarity key elements to establish the semantic equality.

```
[ \replace{ target=/theory[1]/theorem[1]/name[1]/text[1] }
     [ "YZ" ],
  \replace{ target=/theory[1]/theorem[2]/name[1]/text[1] }
     [ "XY" ] ]
```

We clearly do not want such a change script because the exchange of identities instead of content is usually not an operation supported by an interfaced service component.

Additionally, the search space for the new subtrees is currently infinite. We want to restrict this space to the set of all subtrees of the target document. By Axiom 4.3.3 we know that inserting a subtree or a semantically equal one produces equal edit costs. Since the inserted subtree needs to be semantically equal to a subtree of the target document, we do not loose potential solutions with lower edit costs by adding this restriction.

In the following, we introduce the notion of a *limited change script* that extends the notion of change script by accounting for all these service-specific aspects. Thereby we employ previously introduced notations like for example $\mathcal{A}_D(D_x)$ for the set of ancestor trees of a tree D_x in a tree D, $\mathcal{S}(D_1)$ for the set of all subtrees of the tree D_1 including the tree D_1 itself, $\vec{D}$ for a reference to the subtree D, precisely a reference to the root node of the subtree D, and Ψ_+^{Δ} for the set of subtrees inserted by the valid edit script Δ.

Definition 4.4.3 (Limited Change Script): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and $\Sigma_{\mathrm{E}} = (\Sigma_{\mathrm{W}}, \Sigma_{\mathrm{L}})$ be an edit specification. Let Δ be a change script for D_1 and D_2 with respect to Σ_S. Hence, there exists a labeled tree D_1' with $(D_1, \Delta) \hookrightarrow_{\mathrm{PATCH}} \mathrm{D}_1'$ and $D_1' =_{\Sigma_{\mathrm{S}}} D_2$. We call Δ a *limited change script* for D_1 and D_2 with respect to Σ_{S} and Σ_{E} if the following properties hold additionally:

1) *Limited Edit Operations:*
$$\forall \delta \in \Delta. \forall D \in \mathcal{A}_{D_1}\big(target(\delta)\big). \Sigma_L(D) = \bot$$

2) *Semantic Similarity Consistency:*
$$\forall D \in \mathcal{S}(D_1). \forall D' \in \mathcal{S}(D_1'). \left(\left(\vec{D} = \vec{D'}\right) \Rightarrow \left(D \cong_{\Sigma_{\mathrm{S}}} D'\right)\right)$$

3) *Closed Edit Operations:*
$$\forall D \in \Psi_+^{\Delta}. D \in \mathcal{S}(D_2)$$

The set of all limited change scripts for D_1 and D_2 with respect to Σ_S and Σ_{E} is denoted by $\mathbb{L}_{\Sigma_S}^{\Sigma_{\mathrm{E}}}(D_1, D_2)$.

Property 1 guarantees that there is no edit operation targeting a labeled tree whose antecedents contain a labeled tree that is limited by Σ_L. Indeed, this antecedent should have been replaced according to the semantics of Σ_L. *Property 2* ensures that all preserved subtrees are still semantically similar. *Property 3* requires that any inserted subtree D must be a subtree of D_2. This property reduces the search space for limited change scripts by an order of magnitude.

Finally, we define the notion of an *optimal change script* as a limited change script with minimal edit costs.

Definition 4.4.4 (Optimal Change Script): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and $\Sigma_{\mathrm{E}} = (\Sigma_{\mathrm{W}}, \Sigma_{\mathrm{L}})$ be an edit specification. Let Δ be a limited change script for D_1 and D_2 with respect to Σ_S and Σ_E. We call Δ an *optimal change script* for D_1 and D_2 with respect to Σ_{S} and Σ_{E} if $\xi(\Delta') \geq \xi(\Delta)$ holds for all limited change scripts Δ' for D_1 and D_2 with respect to Σ_{E} and Σ_S. The set of all optimal change scripts for D_1 and D_2 with respect to Σ_S and Σ_{E} is denoted by $\mathbb{O}_{\Sigma_S}^{\Sigma_{\mathrm{E}}}(D_1, D_2)$.

This raises the question whether an optimal change script always exists that describes the changes between two labeled trees.

Lemma 4.4.5 (Existence of Optimal Change Script): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and $\Sigma_\mathrm{E} = (\Sigma_\mathrm{W}, \Sigma_\mathrm{L})$ be an edit specification. There exists an optimal change script Δ for D_1 and D_2 with respect to Σ_S and Σ_E.

Proof: The edit script $\Delta_0 = [\delta_R(\overrightarrow{D_1}, D_2)]$ is always a change script for D_1 and D_2 with respect to Σ_S because Δ_0 patches D_1 into D_2, which is clearly semantically equal to D_2 with respect to Σ_S. Hence, there exists a labeled tree D_1' with $(D_1, \Delta_0) \hookrightarrow_{\mathrm{PATCH}} \mathrm{D}_1'$ and $D_1' =_{\Sigma_\mathrm{S}} D_2$. Furthermore, Δ_0 is a limited change script with respect to Σ_S and Σ_E because *Property 1* holds since the root node does not have parents, *Property 2* holds because $\forall D \in \mathcal{S}(D_1). \forall D' \in \mathcal{S}(D_1'). \vec{D} \neq \overrightarrow{D'}$, and *Property 3* because $\Psi_+^{\Delta_0} = \{D_2\}$. Thus, $c_0 = \xi(\Delta_0)$ is an upper bound of the edit costs of all optimal change scripts for D_1 and D_2 with respect to Σ_S and Σ_E.

If Δ_0 is an optimal change script, we are done, otherwise there exists a limited change script Δ' for D_1 and D_2 with respect to Σ_S and Σ_E with edit costs $c' = \xi(\Delta')$ and $c_0 > c'$. The strict total ordering $>$ inferred from the well-founded total ordering $\geq$ on weights is well-founded, too. Thus, any chain of limited change scripts, that is descending with respect to their edit costs and $>$, is finite. In fact, the edit cost 0 of the empty edit script is the least possible weight and a lower bound of the edit costs. Hence, the descending chain that starts with Δ_0 contains finitely many elements. Thus, there exists an optimal change script Δ for D_1 and D_2 with respect to Σ_S and Σ_E, which is either Δ_0 or the last element in the descending chain.

■

Let us briefly summarize the results of this chapter: We introduced a *similarity specification* which can be used to configure the *similarity order* and *similarity key* of subtrees in an interface document. This specification needs to be set up only once for the interface document in a mediation module and can be considered static.

Furthermore, we introduced the *edit specification* which can be used to configure the *edit weight* and *edit limitation* of subtrees in an interface document. This specification is used by a service component to retrieve an optimal change script of an interface document from a mediation module. The edit specification changes dynamically with every request.

In this chapter, we introduced the *constrained weighted tree alignment problem* which is the problem of computing an optimal change script between two labeled trees with respect to a *similarity specification* and an *edit specification*.

4.5 Use Case

Before we illustrate the different possibilities of the similarity specification and edit specification we need to define the concrete serialized representation of these specifications. We begin with the similarity specification. By default, we define that all layers in all trees are ordered and that they do not have any key elements. Thus, we have $\Sigma_O(l_1, l_2) = \top$ and $\Sigma_K(l_1, l_2) = \emptyset$ for all $l_1 \in \mathcal{L}_V$ and $l_2 \in \mathcal{L}_A$. This default definition can now be overridden for combinations of tree labels and layer labels as follows:

```
\similarity{}[
    \order{ name="foo1", layer="bar"}[],
    \order{ name="foo2", layer="bar"}[],
    \keys{ name="foo1", layer="bar"}[ "id", "type" ],
    \keys{ name="foo2", layer="bar"}[ "id" ] ]
```

This similarity specification corresponds to $\Sigma_O("foo1","bar") = \Sigma_O("foo2","bar") = \perp$ overriding the default similarity order, and to $\Sigma_K("foo1","bar") = \{"id","type"\}$, $\Sigma_K("foo2","bar") = \{"id"\}$ overriding the default empty similarity keys.

Practical Definition of Edit Weights. Regarding the edit specification, the difficulty with the edit weight is that it has to satisfy the consistency criterions of *Axiom 4.3.3*. Therefore, instead of asking the service component to give the full insert weight W_+ and delete weight W_- for all trees, we expect that the component just provides the additional *insert payload* Υ_+ and *delete payload* Υ_- for specific trees. Let D be the interface document and $D_x \in \mathcal{S}(D)$ be a subtree of D, then we define the edit weight by $W_+(D_x) \coloneqq |D_x| + \sum_{D_k \in \mathcal{S}(D_x)} \Upsilon_+(D_k)$ and $W_-(D_x) \coloneqq |D_x| + \sum_{D_k \in \mathcal{S}(D_x)} \Upsilon_-(D_k)$. Thereby, the initial payloads are by default the neutral weight 0 for all subtrees. This clearly satisfies the structural criterions 1) and 2) of *Axiom 4.3.3*. Furthermore, we can guarantee criterion 3), equal insert weight for semantically equal trees, by defining the insert payload over the label of trees.

Practical Definition of Edit Limitations. Besides that, the service component needs also a reasonable way to define the edit limitations for the subtrees in an interface document. By default we define that there is no limitation for all subtrees $D_x \in \mathcal{S}(D)$. Then we allow the service component to define *global limitations* and *local limitations*. In case of a global limitation, the label of a tree $l \in \mathcal{L}_V$ is given and the edit limitation $\Sigma_L(D_x) = \top$ limits all subtrees D_x with the label $L(D_x) = l$. In case of a local limitation, the subtree D_k is specified and thus limited by $\Sigma_L(D_k) = \top$.

The edit payloads and the default edit limitation can be overridden as follows, using in this example natural numbers as weights:

```
\edit{}[
    \weight{}[
      \insert{ name="foo1", weight="10" }[],
      \delete{ path=/foo2[1] weight="50" }[] ],
    \limit{}[
      \global{ name="foo1" }[],
      \local{ path=/foo2[1] }[] ] ]
```

This edit specification corresponds to $\Upsilon_+(D_1) = 10$ for all D_1 with $L(D_1) = "foo1"$ and $\Upsilon_-(D_2) = 10$ for D_2 referenced by the path `/foo2[1]`. Additionally, we override the default edit limitation globally by $\Sigma_L(D_3) = \top$ for all D_3 with $L(D_3) = "foo1"$, and locally by $\Sigma_L(D_4) = \top$ for D_4 referenced by the path `/foo2[1]`.

Let us now illustrate the specifications by considering a concrete example with the two documents D_1 and D_2 as shown in Table 9. These documents contain two mathematical theories which both consist of one axiom and two theorems. In order to concentrate on the important parts of this example, we replaced complex mathematical conjectures by the schema `"#i"`.

D_{theory}	`\theory{}[`	D'_{theory}	`\theory{}[`
D_{axiom}	`  \axiom{}[`	$D'_{theorem1}$	`  \theorem{}[`
	`    \name{}[ "AB" ]],`		`    \name{}[ "YZ" ],`
$D_{theorem1}$	`  \theorem{}[`		`    \conj{}[ "#3" ]],`
	`    \name{}[ "XY" ],`	$D'_{theorem2}$	`  \theorem{}[`
	`    \conj{}[ "#3" ]],`		`    \name{}[ "XY" ],`
$D_{theorem2}$	`  \theorem{}[`		`    \conj{}[ "#1" ]],`
	`    \name{}[ "YZ" ],`	D'_{axiom}	`  \axiom{}[`
	`    \conj{}[ "#2" ]]]`		`    \name{}[ "AB" ]]]`

Table 9. Comparing two mathematical documents D_1 (on the left) and D_2 (on the right)

There are almost always several valid change scripts that transform a document into another one. Note that our motivation for computing the change script between two documents is not to obtain the most concise representation of the modifications. In addition to conciseness, we are considering the similarity of subtrees, the collateral costs of changing subtrees and the desired granularity of the service component that is going to react on the change script. Let us explore these aspects in the following and observe how a different specification influences the resulting change script.

To begin, we compute the first optimal change script for D_1 and D_2 using the default similarity specification Σ_S, which postulates that every layer in the tree is ordered and that every layer does not have any similarity keys. Furthermore, the service component uses the default edit specification Σ_E. Thereby, the edit weight $\Sigma_W = (W_-, W_+)$ assigns the delete and insert weight according to $\forall D \in \mathcal{S}(D_1).\, W_-(D) = W_+(D) = |D|$. Thus the edit weight reflects the size of the labeled tree, which is the standard edit weight used by the majority of algorithms. Additionally, the service component defines no edit limitations, hence we have $\forall D \in \mathcal{S}(D_1).\, \neg\Sigma_L(D)$. Altogether, we use the following specifications.

```
\similarity{}[]
\edit{}[]
```

An optimal change script for D_1 and D_2 with respect to Σ_S and Σ_E is the following one:

```
[ \append{ target=/theory[1], layer="content" }
     [ \axiom{}[
          \name{}[ "AB" ]] ],
  \replace{ target=/theory[1]/theorem[1]/name[1]/text[1] }
     [ "YZ" ],
  \replace{ target=/theory[1]/theorem[2]/name[1]/text[1] }
     [ "XY" ],
  \replace{ target=/theory[1]/theorem[2]/conj[1]/text[1] }
     [ "#1" ],
  \delete{ target=/theory[1]/axiom[1] }[] ]
```

This change script is problematic since the name of a theorem, the unique identifier, gets replaced. It is more desirable to change only the corresponding labeled subtrees, and to completely remove non-matched subtrees. In particular, the change script has to preserve the similarity relation of nodes. Therefore, let us change the similarity keys as follows $\Sigma_K(\text{"}theorem\text{"}, \text{"}content\text{"}) = \Sigma_K(\text{"}axiom\text{"}, \text{"}content\text{"}) = \{\, \text{"}name\text{"} \,\}$. By using the label "$theorem$" to assign the similarity key value, we assign this similarity key value to any equally labeled subtree, that is the equivalence class of subtrees having the label "$theorem$". In this example, the new similarity key states that theorems and axioms are identified by a child element having the label "$name$". Only identified elements are changed more deeply, not identified elements are always deleted. Altogether, we use the following specifications now.

```
\similarity{}[
    \keys{ name="theorem", layer="content"}[ "name" ],
    \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[]
```

The effect on the resulting optimal change script is the following:

```
[ \append{ target=/theory[1], layer="content" }
    [ \theorem{}[
        \name{}[ "XY" ],
        \conj{}[ "#1" ]],
      \axiom{}[
        \name{}[ "AB" ]] ],
  \replace{ target=/theory[1]/theorem[2]/conj[1]/text[1] }
    [ "#3" ],
  \delete{ target=/theory[1]/axiom[1] }[],
  \delete{ target=/theory[1]/theorem[1] }[] ]
```

The name of the theorem is no longer modified. Indeed, the last theorem is preserved with its name and only its conjecture content is replaced. This change script is already a bit more adequate than the previous one. Let us continue changing the similarity specification. Assume the order of the axioms and theorems inside of a theory is not relevant for the interfaced service component. We can change the similarity order of theories by $\Sigma_O("theory",\ "content") = \bot$. Thus we use the following specifications.

```
\similarity{}[
   \order{ name="theory", layer="content"}[],
   \keys{ name="theorem", layer="content"}[ "name" ],
   \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[]
```

Then, we obtain the following optimal change script:

```
[ \replace{ target=/theory[1]/theorem[1]/conj[1]/text[1] }
    [ "#1" ],
  \replace{ target=/theory[1]/theorem[2]/conj[1]/text[1] }
    [ "#3" ] ]
```

Surprisingly this change script is even more concise than all previous change scripts by taking the semantic similarity into account. Now, let us in addition to the previous case assume that the granularity of the component interface is only implemented up to the level of theorems. Hence, the fine-grained information of this change script is too detailed for the service component. By changing the edit limitation for theorems globally to $\Sigma_L(D_x) = \top$ for all subtrees D_x in D_1 with $L(D_x) = "theorem"$, we are able to restrict the granularity of the optimal change script. We use now the following specifications.

```
\similarity{}[
   \order{ name="theory", layer="content"}[],
   \keys{ name="theorem", layer="content"}[ "name" ],
   \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[
   \limit{}[
     \global{ name="theorem" }[] ] ]
```

This results in the following change script which replaces both theorems by their modified versions.

```
[ \replace{ target=/theory[1]/theorem[1] }
    [ \theorem{}[ \name{}[ "XY" ], \conj{}[ "#1" ]] ],
  \replace{ target=/theory[1]/theorem[2] }
    [ \theorem{}[ \name{}[ "YZ" ], \conj{}[ "#3" ]] ] ]
```

We continue changing the specification by restoring the similarity order to its default with $\forall l \in \mathcal{L}_A. \forall D \in \mathcal{S}(D_1). \Sigma_O(L(D), l)$ because we assume now that the order is again relevant for the service component. By using natural numbers as weights, let us assume further that the collateral costs of changing the axiom are a value of 10 higher than usual because for example 10 proofs depend on the axiom. Therefore, we define the delete payload of the axiom subtree D_{axiom} to be $\Upsilon_-(D_{axiom}) = 10$. Hence, we use the following specifications.

```
\similarity{}[
   \keys{ name="theorem", layer="content"}[ "name" ],
   \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[
   \weight{}[
     \delete{ path=/theory[1]/axiom[1] weight="10" }[] ],
   \limit{}[
     \global{ name="theorem" }[] ] ]
```

It is now cheaper to remove and insert the theorems than changing the axiom, which results for example in the following optimal change script.

```
[ \insert{ target=/theory[1]/axiom[1] }
    [ \theorem{}[ \name{}[ "YZ" ], \conj{}[ "#3" ]],
      \theorem{}[ \name{}[ "XY" ], \conj{}[ "#1" ]] ],
  \delete{ target=/theory[1]/theorem[1] }[],
  \delete{ target=/theory[1]/theorem[2] }[] ]
```

4.6 Discussion

In the context of the tree-to-tree correction problem one usually differentiates between the following three problems: (1) *tree edit distance*, (2) *tree alignment distance*, and (3) *tree inclusion*. We will give a short overview of these problems and the state-of-the-art algorithms for solving these problems. Then, we will discuss their relationship to our problem which results from the paradigm of using *documents as interfaces* to service components. Thereby, we focus in our problem setting on the computation of an optimal edit script with respect to the semantics of the interface document, the edit weights of the subtrees of this document and the granularity requirements of the interfaced service component.

Tree edit distance. This is the problem of computing the optimal edit script between two trees, which is defined as a sequential edit script with minimal costs. Thereby, the costs of the edit script are called the *tree edit distance* [Tai, 1979]. The costs of a sequential edit script are calculated by using a given cost function defined on each edit operation and satisfying the properties of a distance metric. Algorithms for the tree edit distance problem of ordered trees are given in [Zhang & Shasha, 1989], [Klein, 1998], [Chen, 2001] and [Rönnau *et al*, 2009]. It has been shown in [Zhang *et al*, 1992] that algorithms with polynomial time complexity exist only for special cases of this problem. Finally, there are also solutions to a constrained variant of the tree edit distance problem ([Zhang, 1995], [Richter, 1996]), that use a notion of similarity to keep as much of the structure of the trees as possible.

Tree alignment distance. An alignment between two trees is obtained by first inserting nodes with empty labels into both trees such that the trees become isomorphic when ignoring the labels. The tree alignment [Jiang *et al*, 1994] is then an overlay of both trees, where each node is labeled by a pair of labels. The cost of an alignment is the sum of costs of all pairs with different labels. The costs are computed by a given cost function defining a distance metric for tree labels. Algorithms for the tree alignment distance problem of ordered trees are given in [Jiang *et al*, 1994] and [Jansson & Lingas, 2001]. The first algorithm can be easily modified to handle the problem for unordered trees as well.

Tree Inclusion. As the name of the problem already indicates, the tree inclusion problem [Knuth, 1969] determines whether a tree can be obtained from another tree by only deleting nodes. Algorithms for the tree inclusion problem of ordered trees are given in [Kilpelainen & Mannila, 1995], [Richter, 1997] and [Chen, 1998]. The first algorithm can be easily modified to handle the problem for unordered trees as well.

First of all, we need to classify our problem into one of these three classes. Thereby, we can clearly exclude the tree inclusion problem. From a high-level point of view the difference between the tree edit and the tree alignment distance problem is that for the tree edit a *sequential edit script* is computed whereas for the tree alignment a *parallel edit script* is computed. According to our change model and the Definition 3.3.8 of a valid edit script we apply edit operations simultaneously on a tree. Thus, we want to compute a parallel edit script and we are looking for a solution to a tree alignment problem.

With the similarity specification we introduced possibilities to define the similarity order of the trees, hence we indicate for every subtree whether we need to solve the ordered or unordered tree alignment problem. Additionally, the similarity specification allows to define the similarity keys of the trees. Together with the notion of change script in Definition 4.2.5 this defines a constrained version of the tree alignment distance problem.

Why not use or extend the existing algorithms? We would like to, but our change model and edit specification are not compatible with the requirements of these algorithms. Let us discuss the issues in more details:

Change model. The existing algorithms use a change model that includes the *relabel* operation, an edit operation that substitutes the label of a subtree by another label. Unfortunately, this operation is not available in our mediation context, because the subtrees in an interface document correspond to knowledge entities in a component. A subtree has to be deleted before a new subtree with different label is inserted at its position. Not having the relabel operation has the side-effect that we cannot compute a tree alignment by simply ignoring the labels. This problem now heavily depends on possible subtree alignments.

Edit specification. The existing algorithms require the cost function to be a distance metric. In our mediation context, we cannot ensure this requirement, because there is no relationship between the insert weight and the delete weight of a subtree. In practice, the delete weight is often greater than the insert weight for the same subtree, which is not compatible with the symmetry of costs required by a distance metric. Furthermore, the limitation of granularity is not natively supported, hence we would waste resources.

The only algorithm to the best of our knowledge that solves a subcase of our problem is given in [Radzevich, 2006], an adaptation of [Jiang *et al*, 1994] for partial semantic similarity. However, this algorithm is not an efficient solution for our constrained tree alignment problem because it requires all possible combinations of subtree alignments to be solved. Furthermore, it only employs a pre-defined cost function, the standard distance metric, and it does not deal with the limitation of edit granularity.

5 Computing Changes

In this chapter we will develop a new algorithm for solving the constrained weighted tree alignment problem, which has been introduced in the previous chapter. The issue of applying the state-of-the-art algorithm in [Radzevich, 2006] to our problem is essentially the recursive nature of that algorithm. In order to compute the optimal changes between two trees, that algorithm requires the optimal changes between all possible combinations of children to be computed. Regardless of the input trees, that algorithm always performs the maximal number of tree comparisons. Since we expect in the *Change-Oriented Architecture* frequent small changes in a relatively large interface document, this is a severe performance drawback.

The main idea of our algorithm is to reduce the constrained weighted tree alignment problem to a search problem. We accomplish this goal by introducing the notion of a *change script modulo*, which is a partially computed change script where some pairs of subtrees are considered semantically equal by postulation. This set of *critical tree pairs* is the set of remaining tasks. By adding the change scripts for each of these pairs of subtrees we can complete a change script modulo to a full change script. The search for an optimal change script between two trees starts with an empty change script modulo the pair of both initial trees. Then, the search for an optimal change script proceeds as follows. In every search step, we select one of the least cost change script modulo and select one of the task subtree pairs. For this pair of subtrees we compute all combinations of change scripts modulo pairs of semantically similar children. The original change script modulo is then extended, hence the search graph is dynamically expanded by the resulting set of change scripts modulo. The search for an optimal change script terminates when a least cost change script modulo an empty set is selected for expansion.

This chapter starts with an introduction of partial matching mappings between labeled trees and the notion of change script modulo critical tree pairs. We prove that change scripts modulo can be extended to full change scripts. Then we describe the generation of change scripts modulo from the partial matching mappings of a tree layer. We prove that these change scripts modulo correctly expand critical tree pairs. To find the optimal solution in the space of change scripts modulo, we use Dijkstra's algorithm [Dijkstra, 1959] for the single-source shortest-path-to-goal problem. We adapt this algorithm to the dynamic expansion of the search graph of change scripts modulo. We prove the termination, soundness and completeness of our algorithm. Finally, we illustrate the algorithm with a use case before we compare the average runtime complexity of our algorithm with the state-of-the-art algorithm given in [Radzevich, 2006].

5.1 Critical Tree Pairs

In contrast to the state-of-the-art algorithm we do not compute the change script by comparing both trees bottom-up. Instead we explore the change script search space by comparing both trees top-down. Thereby the strategy is to extend *partial matching mappings* to full matching mappings by computing a *change script modulo*. We will first introduce the new concepts of matching mappings.

Definition 5.1.1 (Partial Matching Mappings): Let V_1, V_2 be sequences of labeled trees, let P be a predicate over pairs of labeled trees. We define the set ω of *partial matching mappings* between sequences of labeled trees as follows:

$$\omega^{\mathrm{P}}_{V_1 \leftrightarrow V_2} := \{ f \in \mathfrak{M}_{V_1 \leftrightarrow V_2} \,|\, \phi(f, P) \}$$

The partial matching mappings are defined with respect to a context $\tau = (\varphi, C)$ where φ is a Boolean value indicating the relevance of the order and where the sequence $C \subseteq V_1$ contains the elements of V_1 which can be mapped to multiple elements of V_2. We define $C := \emptyset$ as the default unless explicitly specified. We omit the context τ in the notation when it can be inferred. The set ω contains all bijective multi-mappings which respect the order ($\varphi = \top$) or not ($\varphi = \bot$) between corresponding labeled trees in V_1 and V_2, where the correspondence is defined by the predicate P.

In contrast to the *Definition 4.1.5* of matching mappings Ω, we do not require a corresponding mapping partner tree for each labeled tree in V_1. This definition of partial matching mappings implies that subsets of partial matching mappings are on their own valid partial matching mappings.

For example, let $V_1 = [A, B, C]$ and $V_2 = [C, A]$ be sequences of labeled trees without children with $L(A) = "A"$, $L(B) = "B"$ and $L(C) = "C"$. In the ordered case $\tau = (\top, \emptyset)$, the set $\omega^{\approx}_{V_1 \leftrightarrow V_2} = \{\emptyset, f_1, f_2\}$ contains $f_1 := \{((1, A), (2, A))\}$ and $f_2 := \{((3, C), (1, C))\}$. In the unordered case $\tau' = (\bot, \emptyset)$, the set $\omega^{\approx}_{V_1 \leftrightarrow V_2} = \{\emptyset, f_1, f_2, f_3\}$ contains additionally $f_3 := \{((1, A), (2, A)), ((3, C), (1, C))\}$. Let $V_3 = V_4 = [E]$ be a sequence of a labeled tree without children with $L(E) = "E"$. Then we have the set $\omega^{\approx}_{V_3 \leftrightarrow V_4} = \{\emptyset, f_4\}$ with $f_4 := \{((1, E), (1, E))\}$ in both the ordered and the unordered case.

In the context of the tree alignment problem, we are only interested in the partial matching mappings with maximal coverage.

Definition 5.1.2 (Maximal Partial Matching Mappings): Let V_1, V_2 be sequences of labeled trees, let P be a predicate over pairs of labeled trees. We define the set ϖ of *maximal partial matching mappings* between sequences of labeled trees as follows:

$$\varpi^{\mathrm{P}}_{V_1 \leftrightarrow V_2} := \left\{ f \in \omega^{\mathrm{P}}_{V_1 \leftrightarrow V_2} \middle| \neg \exists \mathrm{g} \in \omega^{\mathrm{P}}_{V_1 \leftrightarrow V_2} . f \subsetneq g \right\}$$

The maximal partial matching mappings are defined with respect to a context $\tau = (\varphi, C)$ with $C := \emptyset$ by default. We omit the context when it can be inferred. The set ϖ contains all maximal bijective multi-mappings which respect the order ($\varphi = \top$) or not ($\varphi = \bot$) between corresponding labeled trees in V_1 and V_2, where the correspondence is defined by the predicate P. It follows that $\varpi^{\mathrm{P}}_{V_1 \leftrightarrow V_2} \subseteq \omega^{\mathrm{P}}_{V_1 \leftrightarrow V_2}$.

Continuing our example from Definition 5.1.1, the set $\varpi^{\approx}_{V_1 \leftrightarrow V_2} = \{f_1, f_2\}$ contains $f_1 := \{((1, A), (2, A))\}$ and $f_2 := \{((3, C), (1, C))\}$ in the ordered case $\tau = (\top, \emptyset)$. In the unordered case $\tau' = (\bot, \emptyset)$, the set $\varpi^{\approx}_{V_1 \leftrightarrow V_2} = \{f_3\}$ contains the only maximal matching mapping $f_3 := \{((1, A), (2, A)), ((3, C), (1, C))\}$. Furthermore, the set $\varpi^{\approx}_{V_3 \leftrightarrow V_4} = \{f_4\}$ contains $f_4 := \{((1, E), (1, E))\}$ in both the ordered and the unordered case.

Analogously to the *Definition 4.1.6* of tree matching mappings, we can combine maximal partial matching mappings to maximal partial tree matching mappings.

Definition 5.1.3 (Maximal Partial Tree Matching Mappings): Let D_1, D_2 be labeled trees, let P be a predicate over pairs of labeled trees. Furthermore, let $V_{l,1} := \mathcal{C}_l(D_1)$ and $V_{l,2} := \mathcal{C}_l(D_2)$ be the children in the layer l of the trees D_1 and D_2. We define the set $\mathcal{M}$ of *maximal partial tree matching mappings* as follows:

$$\mathcal{M}^{P}_{(D_1, D_2)} := \left\{ \bigcup_{l \in \mathcal{L}_A} f_l \middle| f_l \in \varpi^{P}_{V_{l,1} \leftrightarrow V_{l,2}} \right\}$$

The maximal partial tree matching mappings are defined with respect to a context $\kappa = (\Sigma_{\mathrm{S}}, \Lambda)$ where $\Sigma_{\mathrm{S}} = (\Sigma_{\mathrm{O}}, \Sigma_{\mathrm{K}})$ is a similarity specification, and Λ is a function from labeled trees to Boolean values. Then the context $\tau_l = (\varphi_l, C_l)$ for ϖ is computed for every layer l with $\varphi_l := \Sigma_O(l_0, l)$ and $C_l := \{x \in V_{l,1} | \Lambda(\mathrm{x})\}$.

Hence, the function Λ indicates which subtrees of D_1 may match multiple subtrees of D_2. We define $\Lambda_\perp$ to assign $\perp$ to every labeled tree such that $C_l \coloneqq \emptyset$ for every layer l. By default we assume a context $\kappa = (\Sigma_S, \Lambda_\perp)$ with an arbitrary similarity specification Σ_S. We omit the context in the notation when it can be inferred. The maximal partial tree matching mappings are all combinations of all maximal partial matching mappings between corresponding subtrees of D_1 and D_2.

Continuing our example from *Definition 5.1.1*, we define two labeled trees D_1 and D_2 with $\mathcal{C}_a(D_1) \coloneqq V_1$, $\mathcal{C}_a(D_2) \coloneqq V_2$, $\mathcal{C}_c(D_1) \coloneqq V_3$ and $\mathcal{C}_c(D_2) \coloneqq V_4$. The maximal partial tree matching mappings is the set $\mathcal{M}^{\approx}_{(D_1,D_2)} = \{f_1 \cup f_4, f_2 \cup f_4\}$ if Σ_S defines $\varphi_a = \top$. If Σ_S defines $\varphi_a = \perp$, then we have $\mathcal{M}^{\approx}_{(D_1,D_2)} = \{f_3 \cup f_4\}$ independent of the similarity order φ_c.

In the context of semantic change computation by search we want to delete non-matched subtrees, adapt the matched ones and insert missing subtrees. Therefore, we are only interested in extensible tree matching mappings that are extensions of a tree key matching mapping, an element of the set $\mathcal{K}^{P}_{(D_1,D_2)}$ as introduced in *Definition 4.2.1*. The reason is that all adapted subtrees have to be semantically similar to the original subtree, which is the similarity consistency required by limited change scripts.

Definition 5.1.4 (Extensible Tree Matching Mappings): Let D_1, D_2 be labeled trees, let P be a predicate over pairs of labeled trees. We define the set $\mathcal{E}$ of *extensible tree matching mappings* as follows:

$$\mathcal{E}^{P}_{(D_1,D_2)} \coloneqq \left\{ f \in \mathcal{M}^{P}_{(D_1,D_2)} \middle| \exists g \in \mathcal{K}^{P}_{(D_1,D_2)}. g \subseteq f \right\}$$

The extensible tree matching mappings are defined with respect to a context $\kappa = (\Sigma_S, \Lambda)$ and contain all maximal partial tree matching mappings that are extensions of a tree key matching mapping. Thus, the similarity key children of every labeled tree preserve their semantic similarity. By default we assume $\Lambda \coloneqq \Lambda_\perp$. We omit the context in the notation when it can be inferred.

Continuing our example, we define the similarity key of the layer a to be the element with the label "B". Then the set of extensible tree matching mappings $\mathcal{E}^{\approx}_{(D_1,D_2)}$ is empty because none of the maximal partial tree matching mappings $f \in \mathcal{M}^{\approx}_{(D_1,D_2)}$ extends a tree key matching mapping $g \in \mathcal{K}^{\approx}_{(D_1,D_2)}$. If we define the similarity key of the layer a to be the element with the label "A" then the set $\mathcal{E}^{\approx}_{(D_1,D_2)} = \{f_1 \cup f_4\}$ contains an extensible tree matching mapping if Σ_S defines $\varphi_a = \top$. If Σ_S defines $\varphi_a = \perp$, then we have $\mathcal{E}^{\approx}_{(D_1,D_2)} = \{f_3 \cup f_4\}$ independent of the similarity order φ_c.

An extensible tree matching mapping is a set containing pairs of direct subtrees of two labeled trees D_1 and D_2. We introduce the pairs of all tree layer subtrees as the set of *tree pairs* of two labeled trees D_1 and D_2.

Definition 5.1.5 (Tree Pairs): Let D_1 and D_2 be labeled trees. A *tree pair* of D_1 and D_2 is an element of the following set:

$$TP(D_1, D_2) \coloneqq \{(D_1, D_2)\} \cup \{TP(t_1, t_2) | (t_1, t_2) \in TLP(D_1, D_2)\}$$

Tree pairs are a subset of all pairs of subtrees of D_1 and D_2. Let D_1 and D_2 be labeled trees. A *tree layer pair* of D_1 and D_2 is an element of the following set:

$$TLP(D_1, D_2) \coloneqq \bigcup_{l \in \mathcal{L}_A} \mathcal{C}_l(D_1) \times \mathcal{C}_l(D_2)$$

Let Σ_S be a similarity specification. We emphasize that an extensible tree matching mapping $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_1, D_2)}$ contains only tree layer pairs, thus it holds that $f \subseteq TLP(D_1, D_2)$. Since the aim of the search for an optimal change script will be to reduce the remaining critical tree pairs of a change script modulo, we need a measure for the size of tree pairs.

Definition 5.1.6 (Size of Tree Pairs): Let D_1 and D_2 be labeled trees, and let $(t_1, t_2) \in TP(D_1, D_2)$ be a tree pair. The *size of a tree pair* (t_1, t_2), denoted by $\mathrm{W_{TP}}\big((t_1, t_2)\big)$, is defined as $\mathrm{W_{TP}}\big((t_1, t_2)\big) \coloneqq |t_1| + |t_2|$.

A step in the search process will select a critical tree pair and compute the extensible tree matching mappings for this pair of subtrees, together with change scripts that adapt the non-matched subtrees. In order to prove later that this process is terminating, we need to show that the size of a tree pair is greater than the size of the tree layer pairs given by an extensible tree matching mapping.

Lemma 5.1.7 (Size Relation between Tree Pairs and Tree Layer Pairs): Let D_1 and D_2 be labeled trees, and let Σ_S be a similarity specification. For all $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_1, D_2)}$ it holds that

$$\mathrm{W_{TP}}\big((D_1, D_2)\big) > \sum_{(t_1, t_2) \in f} \mathrm{W_{TP}}\big((t_1, t_2)\big)$$

Proof: Because of $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_1,D_2)}$ it holds that $dom(f) \subseteq \mathcal{C}(D_1)$ and $ran(f) \subseteq \mathcal{C}(D_2)$, where $\mathcal{C}(D_i)$ denotes the set of all children of D_i. Since f is a bijective mapping (without multiple matching), every child of D_1 and D_2 occurs at most once in an element of f. Therefore it follows that

$$\mathrm{W}_{\mathrm{TP}}\big((D_1, D_2)\big) = |D_1| + |D_2| \; > \sum_{t_1 \in \mathcal{C}(D_1)} |t_1| \; + \sum_{t_2 \in \mathcal{C}(D_2)} |t_2| \; \geq \sum_{(t_1,t_2) \in f} \mathrm{W}_{\mathrm{TP}}\big((t_1, t_2)\big)$$

■

Before we define change scripts modulo, we need to adapt the notions of semantic similarity and semantic equality to allow for postulating this property for selected tree pairs.

Definition 5.1.8 (Semantic Equality Modulo): Let Σ_S be a similarity specification and D_1, D_2 be labeled trees and $t_1 \in \mathcal{S}(D_1)$, $t_2 \in \mathcal{S}(D_2)$. Let $\Theta \subseteq TP(D_1, D_2)$ be a subset of tree pairs of D_1 and D_2. The *semantic equality* of the trees t_1 and t_2 *modulo* Θ with respect to Σ_S, denoted by $t_1 =^{\Theta}_{\Sigma_S} t_2$, is a predicate over pairs of labeled trees and defined as follows in the context of $\kappa = (\Sigma_S, \Lambda_\perp)$:

$$t_1 =^{\Theta}_{\Sigma_S} t_2 \; :\Leftrightarrow \Big(\big((t_1, t_2) \in \Theta \big) \vee \Big(t_1 \approx t_2 \wedge \Big| \mathcal{T}^{=^{\Theta}_{\Sigma_S}}_{t_1 \leftrightarrow t_2} \Big| > 0 \Big) \Big)$$

Note that the set $\mathcal{T}$ contains the tree matching mappings as introduced in *Definition 4.1.6*. The semantic equality modulo is in principle semantic equality with an additional set of tree pairs of which we postulate their semantic equality. An analogous relation holds for the semantic similarity modulo.

Definition 5.1.9 (Semantic Similarity Modulo): Let Σ_S be a similarity specification and D_1, D_2 be labeled trees and $t_1 \in \mathcal{S}(D_1)$, $t_2 \in \mathcal{S}(D_2)$. Let $\Theta \subseteq TP(D_1, D_2)$ be a subset of tree pairs of D_1 and D_2. The *semantic similarity* of the trees t_1 and t_2 *modulo* Θ with respect to Σ_S, denoted by $t_1 \cong^{\Theta}_{\Sigma_S} t_2$, is a predicate over pairs of labeled trees and defined as follows in the context of $\kappa = (\Sigma_S, \Lambda_\perp)$:

$$t_1 \cong^{\Theta}_{\Sigma_S} t_2 \; :\Leftrightarrow \Big(\big((t_1, t_2) \in \Theta \big) \vee \Big(t_1 \approx t_2 \wedge \Big| \mathcal{K}^{=^{\Theta}_{\Sigma_S}}_{t_1 \leftrightarrow t_2} \Big| > 0 \Big) \Big)$$

Note that the set $\mathcal{K}$ contains the tree key matching mappings which have been introduced in *Definition 4.2.1*.

Analogously to *Lemma 4.1.8* it can be shown that these new notions of *semantic similarity modulo* and *semantic equality modulo* are equivalence relations. This leads to the definition of the new notion of *change script modulo* which is a change script for two labeled trees under the assumption that a given set of tree pairs is already semantically equal.

Definition 5.1.10 (Change Script Modulo): Let D_1, D_1' and D_2 be labeled trees and let Σ_S be a similarity specification. Let $\Theta \subseteq TP(D_1, D_2)$ be a subset of tree pairs of D_1 and D_2 with the property $\forall (t_1, t_2) \in \Theta . TP(t_1, t_2) \cap \Theta = \{(t_1, t_2)\}$. We call Θ a set of *critical tree pairs*. Let Δ be an edit script that does not modify any tree contained in the set of critical tree pairs Θ. The pair (Δ, Θ) is a *change script* for D_1 and D_2 *modulo* Θ with respect to Σ_S if $(D_1, \Delta) \hookrightarrow_{\mathrm{PATCH}} D_1'$ and $D_1' =_{\Sigma_S}^{\Theta} D_2$. The set of all change scripts for D_1 and D_2 modulo Θ with respect to Σ_S is denoted by $\mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta)$.

In the context of semantic change computation by search we are going to successively improve change scripts modulo by reducing the set of critical tree pairs. Therefore, it is important to show that a change script modulo can be expanded by adding a change script modulo for one of its critical tree pairs.

Lemma 5.1.11 (Expandability of Change Scripts Modulo): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and $\Sigma_E = (\Sigma_W, \Sigma_L)$ be an edit specification. Let $\Theta \subseteq TP(D_1, D_2)$ and $(\Delta, \Theta) \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta)$ hold. For all $(t_1, t_2) \in \Theta$ and $\Delta' \in \mathbb{C}_{\Sigma_S}(t_1, t_2, \Theta')$ with $\Theta' = \emptyset$ or $\Theta' \in \mathcal{E}^{\cong_{\Sigma_S}}_{(t_1, t_2)}$, it holds that $(\Delta \boxplus \Delta', \Theta'') \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta'')$ with $\Theta'' \coloneqq (\Theta \backslash \{(t_1, t_2)\}) \cup \Theta'$ and $\xi(\Delta \boxplus \Delta') \geq \xi(\Delta)$.

Proof: We have $(\Delta, \Theta) \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta)$ and thus $(D_1, \Delta) \hookrightarrow_{\mathrm{PATCH}} D_1'$ with $D_1' =_{\Sigma_S}^{\Theta} D_2$. Because of $(t_1, t_2) \in \Theta$, it holds that $t_1 =_{\Sigma_S}^{\Theta} t_2$. Furthermore, let $\Delta' \in \mathbb{C}_{\Sigma_S}(t_1, t_2, \Theta')$ with $\Theta' = \emptyset$ or $\Theta' \in \mathcal{E}^{\cong_{\Sigma_S}}_{(t_1, t_2)}$. Then it holds that $(t_1, \Delta') \hookrightarrow_{\mathrm{PATCH}} t_1'$ with $t_1' =_{\Sigma_S}^{\Theta'} t_2$. With $\Theta'' \coloneqq (\Theta \backslash \{(t_1, t_2)\}) \cup \Theta'$ it follows that $t_1' =_{\Sigma_S}^{\Theta''} t_2$ because $\Theta' \subseteq \Theta''$. Since Δ does not modify any tree in $\mathcal{S}(t_1)$ by the *Definition 5.1.10* of change script modulo, and since Δ' only modifies trees in $\mathcal{S}(t_1)$, the union $\Delta \boxplus \Delta'$ is a valid edit script. Additionally, it holds that $(D_1, \Delta \boxplus \Delta') \hookrightarrow_{\mathrm{PATCH}} D_1''$ and $D_1'' =_{\Sigma_S}^{\Theta''} D_2$. Thus we have $(\Delta \boxplus \Delta', \Theta'') \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta'')$. Furthermore, it holds that $\xi(\Delta \boxplus \Delta') = \xi(\Delta) + \xi(\Delta') \geq \xi(\Delta)$ since $\xi(\Delta') \geq 0$ for all Δ'.

■

The expansion of change scripts modulo uses the property that an extensible tree matching mapping contains tree pairs. This expansion principle is the key to understand the idea of the search for an optimal change script, which is, successively completing a minimal change script modulo towards a change script by reducing its set of critical tree pairs.

Since for example a replace operation can be represented by a delete and insert operation with equal costs, we need means to talk about similar change scripts in general.

Definition 5.1.12 (Similar Change Script Modulo): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and Σ_E be an edit specification. Let $\Delta_1, \Delta_2 \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta)$ with $\Theta \subseteq TP(D_1, D_2)$. The change script Δ_1 is *similar* to the change script Δ_2, denoted by $\Delta_1 \cong_{\Sigma_\mathrm{E}} \Delta_2$, if both change scripts delete the same subtrees, that is, $\Psi_-^{\Delta_1} = \Psi_-^{\Delta_2}$, and if they insert semantically equal trees, that is, $\left|\Omega^{=\Sigma_\mathrm{S}}_{\Psi_+^{\Delta_1} \leftrightarrow \Psi_+^{\Delta_2}}\right| > 0$ holds in the context $\tau = (\bot, \emptyset)$.

Note that similar change scripts do not need to behave in the same way. The similarity of change scripts is defined over the trees which they delete (Ψ_-) and insert (Ψ_+). Similar change scripts delete the same subtrees in a labeled tree and insert semantically equal trees. Thus we can show that their edit costs are equal.

Lemma 5.1.13 (Edit Cost Equality of Similar Change Scripts Modulo): Let Σ_S be a similarity specification, $\Sigma_\mathrm{E} = (\Sigma_\mathrm{W}, \Sigma_\mathrm{L})$ be an edit specification with $\Sigma_W = (\mathrm{W}_-, \mathrm{W}_+)$. Furthermore, let D_1, D_2 be labeled trees. Let (Δ_1, Θ),$(\Delta_2, \Theta) \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta)$ with $\Theta \subseteq TP(D_1, D_2)$ and $\Delta_1 \cong_{\Sigma_E} \Delta_2$. Then it holds that the costs of both similar change scripts are equal, that is, $\xi(\Delta_1) = \xi(\Delta_2)$.

Proof: By *Axiom 4.3.3* it follows that $\forall f \in \Omega^{=\Sigma_\mathrm{S}}_{\Psi_+^{\Delta_1} \leftrightarrow \Psi_+^{\Delta_2}}. \forall D \in \Psi_+^{\Delta_1}. \mathrm{W}_+(D) = \mathrm{W}_+(f(D))$.

Thus we have:

$$\xi(\Delta_1) = \sum_{D_i \in \Psi_-^{\Delta_1}} \mathrm{W}_-(\mathrm{D_i}) + \sum_{D_k \in \Psi_+^{\Delta_1}} \mathrm{W}_+(\mathrm{D_k})$$

$$= \sum_{D_i \in \Psi_-^{\Delta_2}} \mathrm{W}_-(\mathrm{D_i}) + \sum_{D_k \in \Psi_+^{\Delta_2}} \mathrm{W}_+(\mathrm{D}_k) \; = \xi(\Delta_2)$$

Hence the edit costs of similar change scripts are equal.

■

The notion of limited change scripts, as introduced in *Definition 4.4.3*, has to be adapted as well to account for critical tree pairs. In particular, *Property 1* will be extended to prevent critical tree pairs of subtrees limited by Σ_L. This extension induces a normal form for critical tree pairs in a way that a limited tree is either itself part of a critical tree pair or not, but none of its descendant trees is allowed to be part of any critical tree pair. Furthermore, the semantic similarity consistency defined by *Property 2* has to be adapted modulo critical tree pairs.

Definition 5.1.14 (Limited Change Script Modulo): Let D_1, D_1' and D_2 be labeled trees and let Σ_S be a similarity specification and $\Sigma_\mathrm{E} = (\Sigma_\mathrm{W}, \Sigma_\mathrm{L})$ be an edit specification. Let $\Theta \subseteq TP(D_1, D_2)$ be a set of critical tree pairs of D_1 and D_2, and let Δ be an edit script. The pair (Δ, Θ) is a *limited change script* for D_1 and D_2 *modulo* Θ with respect to Σ_S and Σ_E if $(\Delta, \Theta) \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta)$ holds, thus $(D_1, \Delta) \hookrightarrow_{\mathrm{PATCH}} D_1'$ and $D_1' =_{\Sigma_\mathrm{S}}^{\Theta} D_2$, and if the following properties hold.

1) *Limited Edit Operations and Critical Tree Pairs*
   a) $\forall \delta \in \Delta. \forall D \in \mathcal{A}_{D_1}\big(target(\delta)\big). \Sigma_L(D) = \bot$
   b) $\forall (t_1, t_2) \in \Theta. \forall D \in \mathcal{A}_{D_1}(t_1). \Sigma_L(D) = \bot$
2) *Semantic Similarity Consistency*
$$\forall D \in \mathcal{S}(D_1). \forall D' \in \mathcal{S}(D_1'). \left(\left(\vec{D} = \overrightarrow{D'}\right) \Rightarrow \left(D \cong_{\Sigma_\mathrm{S}}^{\Theta} D'\right)\right)$$
3) *Closed Edit Operations:*
$$\forall D \in \Psi_+^{\Delta}. D \in \mathcal{S}(D_2)$$

The set of all limited change scripts for D_1 and D_2 modulo Θ with respect to Σ_S and Σ_E is denoted by $\mathbb{L}_{\Sigma_S}^{\Sigma_\mathrm{E}}(D_1, D_2, \Theta)$.

Analogously to change scripts modulo, we need to prove that a limited change script modulo can be expanded by adding a limited change script modulo for one of its critical tree pairs. This statement is important for the validation of the expansion principle of the search for an optimal change script.

Lemma 5.1.15 (Expandability of Limited Change Scripts Modulo): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and $\Sigma_\mathrm{E} = (\Sigma_\mathrm{W}, \Sigma_\mathrm{L})$ be an edit specification. Let $\Theta \subseteq TP(D_1, D_2)$ and $(\Delta, \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_\mathrm{E}}(D_1, D_2, \Theta)$. For all $(t_1, t_2) \in \Theta$ and $\Delta' \in \mathbb{L}_{\Sigma_S}^{\Sigma_\mathrm{E}}(t_1, t_2, \Theta')$ with $\Theta' = \emptyset$ or $\Theta' \in \mathcal{E}_{(t_1,t_2)}^{\cong_{\Sigma_\mathrm{S}}}$, it holds that $(\Delta \boxplus \Delta', \Theta'') \in \mathbb{L}_{\Sigma_S}^{\Sigma_\mathrm{E}}(D_1, D_2, \Theta'')$ with $\Theta'' \coloneqq (\Theta \backslash \{(\mathrm{t}_1, t_2)\}) \cup \Theta'$ and $\xi(\Delta \boxplus \Delta') \geq \xi(\Delta)$.

Proof: By *Lemma 5.1.11* it follows that $(\Delta \boxplus \Delta', \Theta'') \in \mathbb{C}_{\Sigma_S}(D_1, D_2, \Theta'')$ with $\Theta'' \coloneqq (\Theta \backslash \{(t_1, t_2)\}) \cup \Theta'$ and $\xi(\Delta \boxplus \Delta') \geq \xi(\Delta)$. Since $(\Delta, \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, \Theta)$ and $(\Delta', \Theta') \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(t_1, t_2, \Theta')$ hold, we have to show that the *Properties 1a, 1b, 2 and 3* are also satisfied by $\Delta \boxplus \Delta'$. Because $(t_1, t_2) \in \Theta$ we know that $\forall D \in \mathcal{A}_{D_1}(t_1).\, \Sigma_L(D) = \perp$. Hence we have $\forall \delta \in \Delta'.\, \forall D \in \mathcal{A}_{D_1}(target(\delta)).\, \Sigma_L(D) = \perp$ and it follows that *Property 1a* holds for $\Delta \boxplus \Delta'$. Analogously, we have $\forall (t_x, t_y) \in \Theta'.\, \forall D \in \mathcal{A}_{D_1}(t_x).\, \Sigma_L(D) = \perp$ because $t_x \in \mathcal{C}(t_1)$. Therefore, *Property 1b* holds for $\Delta \boxplus \Delta'$. *Property 2* is satisfied by composition because $\Theta' \subseteq \Theta''$ and $\Theta \backslash \{(t_1, t_2)\} \subseteq \Theta''$ hold. Since $t_2 \in \mathcal{S}(D_2)$ holds, the *Property 3* is clearly satisfied, too. Thus, we have $(\Delta \boxplus \Delta', \Theta'') \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, \Theta'')$.

■

Finally, the notion of optimal change scripts, as introduced in *Definition 4.4.4*, has to be adapted as well to account for critical tree pairs. An *optimal change script modulo* is a limited change script modulo with minimal edit costs.

Definition 5.1.16 (Optimal Change Script Modulo): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and Σ_E be an edit specification. Let $\Theta \subseteq TP(D_1, D_2)$ be a set of critical tree pairs of D_1 and D_2, and let Δ be an edit script. The pair (Δ, Θ) is an *optimal change script* for D_1 and D_2 *modulo* Θ with respect to Σ_S and Σ_E if $(\Delta, \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, \Theta)$ holds and if $\xi(\Delta') \geq \xi(\Delta)$ holds for all $(\Delta', \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, \Theta)$. The set of all optimal change scripts for D_1 and D_2 modulo Θ with respect to Σ_S and Σ_E is denoted by $\mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, \Theta)$.

Not every optimal change script modulo can be extended to an optimal change script. The changes that turn out to be necessary to make the critical tree pairs semantically equal may increase the costs arbitrarily. Thus, the strategy that we will use for the search of an optimal change script, is the successive expansion of a critical tree pair of a limited change script modulo with minimal costs. The question rises whether such an optimal change script modulo always exists.

Lemma 5.1.17 (Existence of Optimal Change Script Modulo): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and Σ_E be an edit specification. Let $\Theta \subseteq TP(D_1, D_2)$ be a set of critical tree pairs of D_1 and D_2 such that $\forall (t_1, t_2) \in \Theta.\, \forall D \in \mathcal{A}_{D_1}(t_1).\, \Sigma_L(D) = \perp$. Then there exists an optimal change script modulo (Δ, Θ) for D_1 and D_2 modulo Θ with respect to Σ_S and Σ_E.

Proof: In the case of $(D_1, D_2) \in \Theta$, the edit script $\Delta_x = [\,]$ represents with (Δ_x, Θ) always a change script for D_1 and D_2 modulo Θ with respect to Σ_S because $D_1 =^{\Theta}_{\Sigma_S} D_2$. Furthermore, (Δ_x, Θ) is a limited change script modulo Θ with respect to Σ_S and Σ_E because *Property 1a* and *Property 3* hold since the edit script is empty. *Property 1b* holds by the assumptions of this lemma, and *Property 2* holds because of the reflexivity of $\cong^{\Theta}_{\Sigma_S}$. Since $c_x = \xi(\Delta_x) = 0$ is both an upper and lower bound of the edit costs of all optimal change scripts for D_1 and D_2 modulo Θ with respect to Σ_S and Σ_E, it follows that $(\Delta_x, \Theta) \in \mathbb{O}^{\Sigma_E}_{\Sigma_S}(D_1, D_2, \Theta)$.

In the case of $(D_1, D_2) \notin \Theta$, the edit script $\Delta_0 = [\delta_R(\overrightarrow{D_1}, D_2)]$ represents with (Δ_0, Θ) always a change script for D_1 and D_2 modulo Θ with respect to Σ_S because Δ_0 patches D_1 into D_2, which is clearly semantically equal to D_2 with respect to Σ_S. Hence, there exists a labeled tree D_1' with $(D_1, \Delta_0) \hookrightarrow_{\text{PATCH}} D_1'$ and $D_1' =^{\Theta}_{\Sigma_S} D_2$. Furthermore, (Δ_0, Θ) is a limited change script modulo Θ with respect to Σ_S and Σ_E because *Property 1a* hold since the root node does not have parents. *Property 1b* holds by the assumptions of this lemma. *Property 2* holds because $\forall D \in \mathcal{S}(D_1). \forall D' \in \mathcal{S}(D_1'). \vec{D} \neq \vec{D'}$, and *Property 3* because $\Psi_+^{\Delta_0} = \{D_2\}$. Thus, $c_0 = \xi(\Delta_0)$ is an upper bound of the edit costs of all optimal change scripts for D_1 and D_2 modulo Θ with respect to Σ_S and Σ_E.

If (Δ_0, Θ) is an optimal change script modulo, we are done, otherwise there exists a limited change script (Δ', Θ) for D_1 and D_2 modulo Θ with respect to Σ_S and Σ_E with edit costs $c' = \xi(\Delta')$ and $c_0 > c'$. The strict total ordering $>$ inferred from the well-founded total ordering $\geq$ on weights is well-founded, too. Thus, any chain of limited change scripts modulo is finite because it is descending with respect to their edit costs and $>$. In fact, the edit cost 0 of the empty edit script is the least possible weight and a lower bound of the edit costs. Hence, the descending chain that starts with (Δ_0, Θ) contains finitely many elements. Thus, there exists an optimal change script (Δ, Θ) for D_1 and D_2 with respect to Σ_S and Σ_E, which is either (Δ_0, Θ) or the last element in the descending chain.

■

We introduced extensible tree matching mappings between labeled trees and we introduced critical tree pairs. Based on these notions, we adapted the semantic equality and similarity to account for critical tree pairs, which are assumed to fulfill the properties by postulation. Finally, we introduced the notion of change scripts modulo critical tree pairs, which is the fundamental concept of the search for an optimal change script. Thereby, the search strategy is to expand always a limited change script modulo with minimal costs by adding a limited change script modulo for a critical tree pair, and thus reducing the set of critical tree pairs.

5.2 Change Script Generation

After having introduced the theoretical notions for the semantic change computation, let us now take a look at algorithms for the generation of change scripts. All algorithms, which we present in the following, operate in a predefined shared context consisting of a similarity specification Σ_S and an edit specification Σ_E. We begin with the algorithm for the generation of change scripts for tree layers.

Definition 5.2.1 (Generating Layer Edit Scripts): Let D_x and D_y be labeled trees, $l \in \mathcal{L}_A$ and let $\Sigma_S = (\Sigma_K, \Sigma_O)$ be a similarity specification. Let $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_x, D_y)}$ be an extensible tree matching mapping. The judgment of deriving the valid edit script Δ for the layer l of the extensible tree matching mapping f is denoted by $(f, l) \hookrightarrow_{DELTA} \Delta$. The operational semantics of $\hookrightarrow_{DELTA}$ is defined by the following inference rules. Thereby, we define the set $V_+ \coloneqq \mathcal{C}_l(D_y) \backslash ran(f)$ of non-matched (new) elements of $\mathcal{C}_l(D_y)$ and the set $V_- \coloneqq \mathcal{C}_l(D_x) \backslash dom(f)$ of non-matched (old) elements of $\mathcal{C}_l(D_x)$. In the ordered case $\Sigma_O(D_x, l) = \top$, we furthermore generate from V_+ a function $\hat{g}$ which assigns the maximal contiguous subsequences V of V_+ in $\mathcal{C}_l(D_y)$ to the direct right sibling in $\mathcal{C}_l(D_y)$ which is in turn also an element of $ran(f)$. If there is no such element in case of a tailing subsequence, this subsequence is assigned to ε.

(ordered case)

$$\frac{\begin{array}{c}\Sigma_O(D_x, l) \\ \Delta_1 \coloneqq \left[\delta_I(\vec{t}, V) | (t, V) \in \hat{g}\right] \\ \Delta_2 \coloneqq \left[\delta_A(\overrightarrow{D_x}, l, V) | (\varepsilon, V) \in \hat{g}\right] \\ \Delta_3 \coloneqq \left[\delta_E(\vec{t}) | t \in V_-\right]\end{array}}{(f, l) \hookrightarrow \Delta_1 :: \Delta_2 :: \Delta_3}$$

(unordered case)

$$\frac{\begin{array}{c}\neg\Sigma_O(D_x, l) \\ \Delta_1 \coloneqq \left[\delta_A(\overrightarrow{D_x}, l, V_+) | V_+ \neq [\,]\right] \\ \Delta_2 \coloneqq \left[\delta_E(\vec{t}) | t \in \mathcal{X}_-\right]\end{array}}{(f, l) \hookrightarrow \Delta_1 :: \Delta_2}$$

Table 10. Algorithm DELTA

In the ordered case $\Sigma_O(D_x, l)$, the non-matched children of D_y on the layer l are inserted as ordered contiguous sequence before a matched subtree or appended if they form the end of the children of D_y on the layer l. In the orderless case $\neg\Sigma_O(D_x, l)$, the resulting valid layer edit script removes the children of D_x that are not matched by f on the layer l and appends the children of D_y on the layer l not matched by f.

We can show that the resulting valid layer edit script contains closed edit operations, which means that inserted trees are subtrees of D_y and deleted trees are subtrees of D_x.

Lemma 5.2.2 (Closed Edit Operations of Layer Edit Script Modulo): Let D_x and D_y be labeled trees and let $\Sigma_S = (\Sigma_K, \Sigma_O)$ be a similarity specification. Let $l \in \mathcal{L}_A$ and $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)}$. If we have $(f,l) \hookrightarrow_{DELTA} \Delta_l$, then it holds that

1) $\Psi^{\Delta_l}_{-} = \{t | t \in \mathcal{C}_l(D_x) \backslash dom(f)\}$,
2) $\Psi^{\Delta_l}_{+} = \{t | t \in \mathcal{C}_l(D_y) \backslash ran(f)\}$.

Proof: In the case of $\neg\Sigma_O(D_x, l)$ the properties follow immediately from the construction of the edit script. In the case of $\Sigma_O(D_x, l)$, the *Property 1* follows immediately from the construction, too. Furthermore, we have split the set $V_+ \coloneqq \mathcal{C}_l(D_y) \backslash ran(f)$ into contiguous subsequences for the function $\hat{g}$. These subsequences are all inserted by the construction of the edit script. Therefore, the *Property 2* is satisfied, too.

■

Furthermore, the resulting edit script establishes a matching mapping on a tree layer if applied to the source tree, which means there exists a bijective function between a tree layer of D_x and D_y with respect to a similarity specification Σ_S.

Lemma 5.2.3 (Matching Mapping of Layer Edit Script Modulo): Let D_x and D_y be labeled trees and let $\Sigma_S = (\Sigma_K, \Sigma_O)$ be a similarity specification. Let $l \in \mathcal{L}_A$ and $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)}$. If we have $(f,l) \hookrightarrow_{DELTA} \Delta_l$ and $(D_x, \Delta_l) \hookrightarrow_{\mathrm{PATCH}} D'_x$, then it holds with $V_{l,1} \coloneqq \mathcal{C}_l(D'_x)$ and $V_{l,2} \coloneqq \mathcal{C}_l(D_y)$ in the context $\tau = (\varphi_l, \emptyset)$ and $\varphi_l \coloneqq \Sigma_O(D'_x, l)$ that

$$\left| \Omega^{=^f_{\Sigma_S}}_{V_{l,1} \leftrightarrow V_{l,2}} \right| > 0.$$

Thereby, we use the tree pairs contained in an extensible tree matching mapping $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)}$ as the set of critical tree pairs $\Theta = f$ in the semantic equality modulo $=^f_{\Sigma_S}$.

Proof: We construct a mapping $g \in \Omega^{=^f_{\Sigma_S}}_{V_{l,1} \leftrightarrow V_{l,2}}$ as follows: By *Lemma 5.2.2* we know that the inserted trees are subtrees of the target tree. Therefore we map these trees onto themselves with $h \coloneqq \{(t,t) | t \in \Psi^{\Delta_l}_{+}\}$. In case of $\Sigma_O(D'_x, l) = \top$ the order is respected by the construction of Δ_l. Finally, we construct $g \coloneqq h \cup f$.

■

By composing the generated layer edit scripts we are able to define an algorithm for generating change scripts modulo with respect to a given similarity specification Σ_S.

Definition 5.2.4 (Generating Change Scripts Modulo): Let D_x and D_y be labeled trees and let Σ_S be a similarity specification. The judgment of deriving the set S of change scripts modulo for D_x and D_y with respect to Σ_S from the set of extensible tree matching mappings $F \subseteq \mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)}$ is denoted by $F \hookrightarrow_{GENERATE} S$. The operational semantics of $\hookrightarrow_{GENERATE}$ is defined by the following inference rules.

$$\frac{\Delta \coloneqq \biguplus_{l \in \mathcal{L}_A} \{\Delta_l | (f, l) \hookrightarrow_{DELTA} \Delta_l\} \quad F \hookrightarrow S}{\{f\} \cup F \hookrightarrow \{(\Delta, f)\} \cup S} \qquad \frac{}{\emptyset \hookrightarrow \emptyset}$$

Table 11. Algorithm GENERATE

The algorithm $\hookrightarrow_{GENERATE}$ iterates on the given set of extensible tree matching mappings and computes for every mapping a change script modulo. We use the tree pairs contained in an extensible tree matching mapping $f \in \mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)}$ as the set of critical tree pairs $\Theta = f$ in the change script modulo (Δ, f). Thereby, the algorithm $\hookrightarrow_{DELTA}$ is used to compute a valid edit script for every tree layer that does establish semantic equality on that layer modulo the mapped tree pairs. Thus, the union of all layer edit scripts is a change script modulo critical tree pairs, precisely the tree pairs contained in the specific extensible tree matching mapping.

The definition states that the algorithm computes change scripts modulo for D_x and D_y with respect to a given similarity specification Σ_S. We will prove the correctness of this statement and additionally that this change script modulo is a limited change script modulo with respect to Σ_S and a given edit specification Σ_E if the tree D_x is not edit limited, that is $\neg\Sigma_L(D_x)$ holds, and if both trees D_x and D_y are semantically similar, that is $D_x \cong_{\Sigma_S} D_y$ holds.

Lemma 5.2.5 (Correctness of Generated Change Scripts Modulo): Let D_x and D_y be labeled trees, let Σ_S be a similarity specification and let $\Sigma_E = (\Sigma_W, \Sigma_L)$ be an edit specification. Furthermore, let $\neg\Sigma_L(D_x)$, $D_x \cong_{\Sigma_S} D_y$ and $\mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)} \hookrightarrow_{GENERATE} S$. Then, for all $(\Delta, \Theta) \in S$ it holds that $(\Delta, \Theta) \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_x, D_y, \Theta)$.

Proof: We show that for all $(\Delta, \Theta) \in S$ it holds that $(\Delta, \Theta) \in \mathbb{C}_{\Sigma_S}(D_x, D_y, \Theta)$, which means that $(D_x, \Delta) \hookrightarrow_{\mathrm{PATCH}} \mathrm{D_x'}$ and $D_x' =_{\Sigma_S}^{\Theta} D_y$. By *Lemma 3.3.10* it holds that Δ is a valid edit script as the union of valid edit scripts for different tree layers. By definition of $\hookrightarrow_{DELTA}$ we know that Δ does not modify any tree contained in the critical tree pairs Θ. From $D_x \cong_{\Sigma_S} D_y$ it follows that $D_x \approx D_y$, and since D_x is neither replaced nor removed by Δ, we have $D_x' \approx D_y$. It remains to be shown that $\left|\mathcal{T}_{(D_x', D_y)}^{=_{\Sigma_S}^{\Theta}}\right| > 0$ holds. By *Lemma 5.2.3* we know that $\left|\Omega_{V_{l,1} \leftrightarrow V_{l,2}}^{=_{\Sigma_S}^{\Theta}}\right| > 0$ holds for all $l \in \mathcal{L}_A$ with $V_{l,1} \coloneqq \mathcal{C}_l(D_x')$ and $V_{l,2} \coloneqq \mathcal{C}_l(D_y)$ in the context $\tau = (\varphi_l, \emptyset)$ and $\varphi_l \coloneqq \Sigma_{\mathrm{O}}(\mathrm{D_x'}, \mathrm{l})$. Thus there exists at least one element of $\mathcal{T}_{(D_x', D_y)}^{=_{\Sigma_S}^{\Theta}}$ as the composition of elements from $\Omega_{V_{l,1} \leftrightarrow V_{l,2}}^{=_{\Sigma_S}^{\Theta}}$ for all $l \in \mathcal{L}_A$. Therefore we have $(\Delta, \Theta) \in \mathbb{C}_{\Sigma_S}(D_x, D_y, \Theta)$.

It remains to be shown that all resulting change scripts modulo satisfy the properties of limited change scripts modulo, too. The *Properties 1a and 1b* of limited change scripts modulo are satisfied because $\neg\Sigma_L(D_x)$ holds and the edit operations in Δ target D_x or one of its direct subtrees, furthermore we have $\Theta \subseteq TP(D_x, D_y)$. The *Properties 2 and 3* are satisfied by *Lemma 5.2.2* as follows. Since all $f \in \mathcal{E}_{(D_x, D_y)}^{\cong_{\Sigma_S}}$ map the key subtrees of D_x and D_y, they are neither replaced nor removed by Δ, hence *Property 2* holds. Inserted trees are trees in $\mathcal{C}(D_y)$ which are clearly member of $\mathcal{S}(D_y)$, thus *Property 3* holds. Altogether, it follows that $(\Delta, \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$ for all $(\Delta, \Theta) \in S$.

■

Additionally, we can show that the resulting limited change scripts modulo are optimal.

Lemma 5.2.6 (Optimality of Generated Change Scripts Modulo): Let D_x and D_y be labeled trees, let Σ_S be a similarity specification and let $\Sigma_E = (\Sigma_W, \Sigma_L)$ be an edit specification. Furthermore, let $\neg\Sigma_L(D_x)$, $D_x \cong_{\Sigma_S} D_y$ and $\mathcal{E}_{(D_x, D_y)}^{\cong_{\Sigma_S}} \hookrightarrow_{GENERATE} S$. Then, for all $(\Delta, \Theta) \in S$ it holds that $(\Delta, \Theta) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$.

Proof: By *Lemma 5.2.5* we know that $\Delta \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$ holds for all $(\Delta, \Theta) \in S$. Furthermore, let $f \in \mathcal{E}_{(D_x, D_y)}^{\cong_{\Sigma_S}}$ be an extensible tree matching mapping. For the corresponding generated limited change script (Δ, Θ) it holds that $\Theta = f$. The children of D_x that are not matched by f are not semantically similar to any non-matched children of D_y in the same layer because of the maximality of f.

Since the semantic similarity is an equivalence relation by *Lemma 4.1.8* we are not allowed to modify the non-matched children of D_x because this would violate the semantic similarity consistency of limited change scripts modulo. Thus, the only option is to delete the non-matched children. The missing children in a layer, which are the non-matched children of D_y in that layer, have to be inserted with respect to the order.

By the closed edit operations property of limited change scripts modulo, these inserted trees are subtrees of D_y. Therefore the generated limited change script modulo (Δ, Θ) is similar to an optimal change script modulo (Δ', Θ). By *Lemma 5.1.13* both change scripts modulo have the same costs. Thus, we have $(\Delta, \Theta) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$.

■

So far we considered the generation of optimal change scripts modulo for the case $\neg\Sigma_L(D_x)$ with $D_x \cong_{\Sigma_S} D_y$. Let us now consider the general case.

Definition 5.2.7 (Expanding Critical Tree Pairs): Let D_x and D_y be labeled trees and let Σ_S be a similarity specification and $\Sigma_E = (\Sigma_W, \Sigma_L)$ be an edit specification. The judgment of deriving the set S of change scripts modulo for D_x and D_y with respect to Σ_S from the pair of labeled trees (D_x, D_y) is denoted by $(D_x, D_y) \hookrightarrow_{EXPAND} S$. The operational semantics of $\hookrightarrow_{EXPAND}$ is defined by the following inference rules. The algorithm distinguishes four different cases.

(limited and not semantically equal)

$$\frac{\Sigma_L(D_x) \quad \neg(D_x =_{\Sigma_S} D_y)}{(D_x, D_y) \hookrightarrow \{([\,\delta_R(D_x, D_y)\,], \emptyset)\}}$$

(limited and semantically equal)

$$\frac{\Sigma_L(D_x) \quad D_x =_{\Sigma_S} D_y}{(D_x, D_y) \hookrightarrow \{([\,], \emptyset)\}}$$

(not limited and not semantically similar)

$$\frac{\neg\Sigma_L(D_x) \quad \neg(D_x \cong_{\Sigma_S} D_y)}{(D_x, D_y) \hookrightarrow \{([\,\delta_R(D_x, D_y)\,], \emptyset)\}}$$

(not limited and semantically similar)

$$\frac{\neg\Sigma_L(D_x) \quad D_x \cong_{\Sigma_S} D_y \quad \mathcal{E}_{(D_x, D_y)}^{\cong_{\Sigma_S}} \hookrightarrow_{GENERATE} S}{(D_x, D_y) \hookrightarrow S}$$

Table 12. Algorithm EXPAND

In the case of an edit limitation for the tree D_x with $\Sigma_L(D_x)$, the algorithm $\hookrightarrow_{EXPAND}$ only needs to check for the semantic equality of both trees $D_x =_{\Sigma_S} D_y$. If they are not semantically equal, then the tree D_x is replaced by the tree D_y with the returned change script modulo. If they are semantically equal, then the algorithm returns an empty change script modulo an empty set of critical tree pairs.

In the case of no edit limitation for the tree D_x with $\neg\Sigma_L(D_x)$, the algorithm $\hookrightarrow_{EXPAND}$ first checks for the semantic similarity of both trees $D_x \cong_{\Sigma_S} D_y$. If they are not semantically similar, then the tree D_x is replaced by the tree D_y with the returned change script modulo. If they are semantically similar, then the algorithm computes the set of extensible tree matching mappings $\mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)}$ and uses the algorithm $\hookrightarrow_{GENERATE}$ for computing the corresponding set of change scripts modulo. Such a change script modulo establishes the semantic equality on every tree layer modulo the set of critical tree pairs defined by an extensible tree matching mapping.

We will show now the correctness of the algorithm by proving additionally that the computed change scripts modulo for D_x and D_y with respect to a given similarity specification Σ_S are limited change scripts modulo with respect to Σ_S and a given edit specification Σ_E.

Lemma 5.2.8 (Correctness of Expanding Critical Tree Pairs): Let D_x and D_y be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Furthermore, let $(D_x, D_y) \hookrightarrow_{EXPAND} S$. Then, for all $(\Delta, \Theta) \in S$ it holds that $(\Delta, \Theta) \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_x, D_y, \Theta)$.

Proof: In the case of $\Sigma_L(D_x)$ and $D_x =_{\Sigma_S} D_y$ it holds that $([\,], \emptyset) \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_x, D_y, \emptyset)$ because [ ] is a change script for D_x and D_y, and all properties of limited change script modulo are clearly satisfied. In the case of $\Sigma_L(D_x)$ and $\neg(D_x =_{\Sigma_S} D_y)$ it holds that $([\,\delta_R(\overrightarrow{D_x}, D_y)\,], \emptyset) \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_x, D_y, \emptyset)$ because $[\,\delta_R(\overrightarrow{D_x}, D_y)\,]$ is a change script for D_x and D_y, the *Properties 1a, 1b and 2* are clearly satisfied and *Property 3* by the fact that $D_y \in \mathcal{S}(D_y)$ holds. The case of $\neg\Sigma_L(D_x)$ and $\neg(D_x \cong_{\Sigma_S} D_y)$ is analogously. In the last case of $\neg\Sigma_L(D_x)$ and $D_x \cong_{\Sigma_S} D_y$ it holds for $\mathcal{E}^{\cong_{\Sigma_S}}_{(D_x,D_y)} \hookrightarrow_{GENERATE} S$ by *Lemma 5.2.5* that we have $(\Delta, \Theta) \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_x, D_y, \Theta)$ for all $(\Delta, \Theta) \in S$.

■

Lemma 5.2.9 (Optimality of Expanding Critical Tree Pairs): Let D_x and D_y be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Furthermore, let $(D_x, D_y) \hookrightarrow_{EXPAND} S$. Then, for all $(\Delta, \Theta) \in S$ it holds that $(\Delta, \Theta) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$.

Proof: By *Lemma 5.2.8* we know that $(\Delta, \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$ holds for all $(\Delta, \Theta) \in S$. In the case of $\Sigma_L(D_x)$ and $D_x =_{\Sigma_S} D_y$ it holds that $([\,], \emptyset) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \emptyset)$ because the edit cost of [ ] is the global minimum. In the case of $\Sigma_L(D_x)$ and $\neg(D_x =_{\Sigma_S} D_y)$ it holds that all limited change scripts modulo have to delete D_x and insert D_y because of the *Properties 1a and 3*. Therefore the limited change script modulo $([\,\delta_R(\overrightarrow{D_x}, D_y)\,], \emptyset)$ is similar to all $(\Delta', \emptyset) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \emptyset)$. Hence we have $([\,\delta_R(\overrightarrow{D_x}, D_y)\,], \emptyset) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \emptyset)$ by *Lemma 5.1.13*. In the case of $\neg(\Sigma_L(D_x))$ and $\neg(D_x \cong_{\Sigma_S} D_y)$ it holds that all limited change scripts have to delete D_x and insert D_y because of the *Properties 2 and 3*. Therefore the limited change script $([\,\delta_R(\overrightarrow{D_x}, D_y)\,], \emptyset)$ is similar to all $(\Delta', \emptyset) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \emptyset)$. Hence we have $([\,\delta_R(\overrightarrow{D_x}, D_y)\,], \emptyset) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \emptyset)$ by *Lemma 5.1.13*. In the last case of $\neg\Sigma_L(D_x)$ and $D_x \cong_{\Sigma_S} D_y$ it holds for $\mathcal{E}_{(D_x, D_y)}^{\cong_{\Sigma_S}} \hookrightarrow_{GENERATE} S$ by *Lemma 5.2.6* that we have $(\Delta, \Theta) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_x, D_y, \Theta)$ for all $(\Delta, \Theta) \in S$.

■

Finally, we observe that either no new critical tree pair is added, or the critical tree level pairs given by an extensible tree matching mapping.

Lemma 5.2.10 (Limited Expansion of Critical Tree Pairs): Let D_x and D_y be labeled trees and let Σ_S be a similarity specification and let Σ_E be an edit specification. Furthermore, let $(D_x, D_y) \hookrightarrow_{EXPAND} S$. Then, for all $(\Delta, \Theta) \in S$ it holds that either $\Theta = \emptyset$ or $\Theta \in \mathcal{E}_{(D_x, D_y)}^{\cong_{\Sigma_S}}$.

Proof: In both cases with $\Sigma_L(D_x)$ this follows from the construction. In the case of $\neg(\Sigma_L(D_x))$ with $\neg(D_x \cong_{\Sigma_S} D_y)$ it holds also by construction. In the case of $\neg(\Sigma_L(D_x))$ and $D_x \cong_{\Sigma_S} D_y$ it follows immediately from the operational semantics of $\hookrightarrow_{GENERATE}$.

■

5.3 Change Graph Search

After introducing the building blocks for the generation of change scripts we will now develop an efficient algorithm for computing an optimal change script by the exploration of the search space of limited change scripts modulo. The general idea of reducing the change computation to a search problem is to start the search with the source document and to use valid edit scripts to successively transform this document into the target document (or a semantically equal one). The problem with this approach is the unrestricted and therefore infinite search space. To the best of our knowledge, there is as yet no efficient method to restrict the search space in such a way that an optimal change script is still guaranteed to be in the search space.

The essence of the method which we will present in this chapter is to restrict the search space by considering the optimization problem modulo a set of critical tree pairs. The key to this change of representation is that the nodes in the change graph represent sets of critical tree pairs. They can thus be interpreted as tasks to compute an optimal change script for each contained critical tree pair. We will start the search with an empty change script, which is a limited change script modulo the critical tree pair of the source and target tree. Then, we will dynamically expand the search space by expanding a critical tree pair of one of the current limited change scripts modulo. Thereby, we compute an optimal change script modulo tree level pairs of the selected critical tree pair with minimal cost. When we find a limited change script modulo an empty set of critical tree pairs and this change script has minimal edit cost, we finally have constructed an optimal change script.

Let us now introduce the notion of a change graph, which is a representation for a subset of the search space, the current expansion towards an optimal change script.

Definition 5.3.1 (Change Graph): Let D_1 and D_2 be labeled trees. A *change graph* $C = (V, A)$ for D_1 and D_2 is a directed acyclic graph consisting of a finite set of nodes $V \subseteq \mathcal{P}\big(TP(D_1, D_2)\big)$, which are subsets of tree pairs of D_1 and D_2, and a finite set of edges $A \subseteq V \times V \times D_1^{\Delta}$, which connect nodes by a valid edit script for D_1. The graph contains exactly one root node $v_0 = \{(D_1, D_2)\}$ without incoming edges. Furthermore, we define the following notions:

1) *Active Nodes*
   The set of *active nodes* $\mathrm{O}_C = \{v_x \in V | \neg\exists(v_1, v_2, \Delta) \in A. v_1 = v_x\}$ contains all nodes without outgoing edges.

2) *Change Path*
   The change graph contains a *change path* $p = [v_0, \dots, v_n]$ if $v_i \in V$ for $i \in \{0, \dots, n\}$ and $(v_i, v_{i+1}, \Delta_{i+1}) \in A$ for $i \in \{0, \dots, n-1\}$.
3) *Path Edges*
   We denote the set of *path edges* of a change path p by $A_C(p) \coloneqq \{(v_i, v_{i+1}, \Delta_{i+1}) \in A | 0 \leq i \leq n-1\}$.
4) *Change Paths*
   For all $v \in V \backslash \{v_0\}$ the set of *change paths* $\mathcal{Y}_C(v)$ contains all change paths from v_0 to v. We require the change graph to be connected such that $|\mathcal{Y}_C(v)| > 0$ for all $v \in V \backslash \{v_0\}$.
5) *Change Path Script*
   The *change path script* $\Delta(p)$ of the change path p is defined by $\Delta(p) \coloneqq \bigboxplus_{i \in \{0,\dots,n-1\}} \Delta_{i+1}$ with $(v_i, v_{i+1}, \Delta_{i+1}) \in A_C(p)$.
6) *Node Size*
   The *node size* $\mathrm{W_C}(v)$ for any node $v \in V$ is defined by $\mathrm{W_C}(v) \coloneqq \sum_{\mu \in v} \mathrm{W_{TP}}(\mu)$. Note that a node v is a set of tree pairs and thus $\mu \in v$ is a tree pair.
7) *Goal Node*
   The *goal node* is $\emptyset$, the empty set of tree pairs.

The nodes in the change graph are sets of critical tree pairs that still have to be made semantically equal. This set still contains open change computation tasks. The edges in the change graph contain optimal tree layer change scripts modulo. The idea is to start the search at the root node and to dynamically expand an active node with minimal cost until the goal node is an active node with minimal cost. To define the costs of a node in the change graph we first introduce the notion of an optimal change path script for a node.

Definition 5.3.2 (Optimal Change Path Script): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Let $C = (V, A)$ be a change graph for D_1 and D_2, and $v_0 \in V$ the root node with $v_0 = \{(D_1, D_2)\}$. For the root node v_0 we define the *optimal change path script* as $\overline{\Delta}_C(v_0) \coloneqq [\,]$. An *optimal change path script* $\overline{\Delta}_C(v)$ for a node $v \in V \backslash \{v_0\}$ is defined by $\overline{\Delta}_C(v) \coloneqq \Delta(p)$ where the path p satisfies the following properties:

1) *Change Path*
   $p \in \mathcal{Y}_C(v)$
2) *Minimal Cost Change Path Script*
   $\forall p' \in \mathcal{Y}_C(v). \xi\big(\Delta(p')\big) \geq \xi\big(\Delta(p)\big)$

Then we can define the change path cost of a node as the cost of an optimal change path script for that node. With this notion of change path cost we can also introduce the set of minimal cost active nodes of a change graph.

Definition 5.3.3 (Change Path Cost): The *change path cost* $\xi_C(v)$ for any node $v \in V\backslash\{v_0\}$ are defined by $\xi_C(v) \coloneqq \xi\left(\overline{\Delta}_C(v)\right)$. For the root node v_0 we define the change path cost by $\xi_C(v_0) \coloneqq 0$. Furthermore, we define the *set of minimal cost active nodes* of a change graph C by $\Pi_C = \{v_x \in \mathrm{O}_C | \forall v \in \mathrm{O}_C . \xi_C(v) \geq \xi_C(v_x)\}$.

Using the example change graph shown in Figure 9 we will discuss the different notions introduced with the change graph. The nodes of the change graph are represented in the figure by circles containing the set of critical tree pairs of that node. We use the box label v_i shown above any node to easily refer to it. The node v_0 is the root of this example tree. Nodes are connected by edges labeled by a valid edit script Δ_i.

The set of active nodes for this change graph is $\mathrm{O}_C = \{v_3, v_4\}$ because these nodes do not have outgoing edges. The set of change paths for the node v_3 is $\mathcal{Y}_C(v_3) = \{[v_0, v_1, v_3], [v_0, v_2, v_3]\}$ and for the node v_4 is $\mathcal{Y}_C(v_4) = \{[v_0, v_2, v_4]\}$. Examples for change path scripts are $\Delta([v_0, v_1, v_3]) = \Delta_1 \boxplus \Delta_3$, $\Delta([v_0, v_2, v_3]) = \Delta_2 \boxplus \Delta_4$ and $\Delta([v_0, v_2, v_4]) = \Delta_2 \boxplus \Delta_5$. Let the costs of the valid edit scripts in the change graph be given by $\xi(\Delta_1) = 10$, $\xi(\Delta_2) = 5$, $\xi(\Delta_3) = 3$, $\xi(\Delta_4) = 6$ and $\xi(\Delta_5) = 8$.

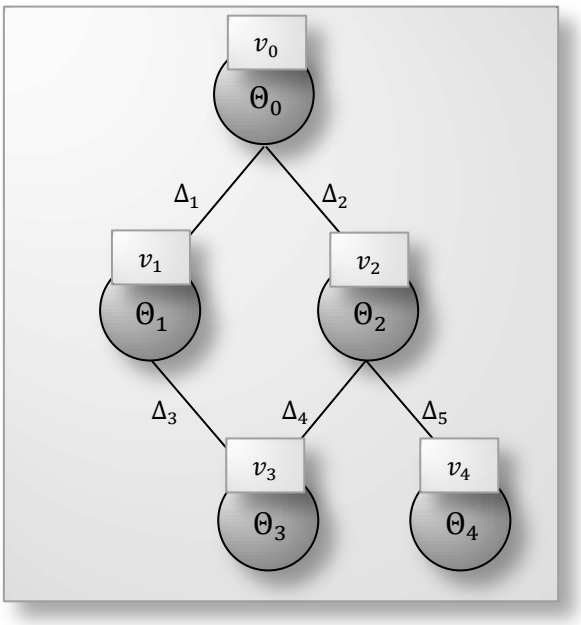


Figure 9. Example of a change graph

The optimal change path script for the node v_3 is $\overline{\Delta}_C(v_3) = \Delta([v_0, v_2, v_3])$ and for the node v_4 is $\overline{\Delta}_C(v_4) = \Delta([v_0, v_2, v_4])$. The change path cost for the node v_3 is $\xi_C(v_3) = 5 + 6 = 11$ and for the node v_4 is $\xi_C(v_4) = 5 + 8 = 13$. Thus, the set of minimal cost active nodes is $\Pi_C = \{v_3\}$. Hence, the node v_3 is the best choice for expansion. As we outlined before, we do not explore the complete search space but a restricted subset.

Definition 5.3.4 (Restricted Change Graph): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. The change graph $C = (V, A)$ is a *restricted change graph* for D_1 and D_2 if there exists for all edges $(v_1, v_2, \Delta) \in A$ a critical tree pair $(t_1, t_2) \in v_1$ such that $v_2 = (v_1 \backslash \{(t_1, t_2)\}) \cup \Theta$ with:

1) *Limited Expansion*
   $\Theta = \emptyset \quad \text{or} \quad \Theta \in \mathcal{E}^{\cong_{\Sigma_S}}_{(t_1,t_2)}$
2) *Optimal Expansion*
   $(\Delta, \Theta) \in \mathbb{O}^{\Sigma_E}_{\Sigma_S}(t_1, t_2, \Theta)$

Property 1 guarantees that between connected nodes exactly one critical tree pair is expanded either by providing the final solution with $\Theta = \emptyset$ or by expanding exactly one tree layer using an extensible tree matching mapping $\Theta \in \mathcal{E}^{\cong_{\Sigma_S}}_{(t_1,t_2)}$. Furthermore, *Property 2* requires that the connecting valid edit script is an optimal change script modulo Θ for the expanded pair of trees t_1 and t_2.

We show now that all change path scripts for a path in a restricted change graph are limited change scripts modulo the critical tree pairs defined by the node with that path.

Lemma 5.3.5 (Change Path Scripts are Limited Change Scripts Modulo): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Let $C = (V, A)$ be a restricted change graph for D_1 and D_2. Then it holds that $\forall v \in V \backslash \{v_0\}. \forall p \in \mathcal{Y}_C(v). (\Delta(p), v) \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_1, D_2, v)$.

Proof: Let $v_n \in V \backslash \{v_0\}$ and $p \in \mathcal{Y}_C(v_n)$, $p = [v_0, \dots, v_n]$. By definition we have $\Delta(p) \coloneqq \boxplus_{i \in \{0,\dots,n-1\}} \Delta_{i+1}$ where $(v_i, v_{i+1}, \Delta_{i+1}) \in A$ for $i \in \{0, \dots, n-1\}$. For $\Delta_R \coloneqq ([\,], v_0)$ it holds that $\Delta_R \in \mathbb{L}^{\Sigma_E}_{\Sigma_S}(D_1, D_2, v_0)$. By the *Property 1* of restricted change graphs we know that $\exists (t_1, t_2) \in v_1. (v_2 = (v_1 \backslash \{(t_1, t_2)\}) \cup \Theta)$ with $\Theta = \emptyset$ or $\Theta \in \mathcal{E}^{\cong_{\Sigma_S}}_{(t_1,t_2)}$.

Furthermore, by *Property 2* it holds that $(\Delta, \Theta) \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(t_1, t_2, \Theta)$ and thus $(\Delta, \Theta) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(t_1, t_2, \Theta)$. By applying *Lemma 5.1.15* to the chain $\Delta_R \boxplus \Delta(p)$ of unions of limited change scripts modulo it follows that $(\Delta_R \boxplus \Delta(p), v) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, v)$. Since $\Delta_R \boxplus \Delta(p) = \Delta(p)$ we have $(\Delta(p), v) \in \mathbb{L}_{\Sigma_S}^{\Sigma_E}(D_1, D_2, v)$.

■

From the properties of the restricted change graph it does not follow directly that an optimal change path script to the goal node in the change graph is an optimal change script. Indeed, this depends additionally on the optimality of the expansion strategy as we will show. Let us now introduce a concrete algorithm for extending restricted change graphs.

Definition 5.3.6 (Restricted Change Graph Extension): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification, let Σ_E be an edit specification, and let $C = (V, A)$ be a restricted change graph for D_1 and D_2. The judgment of deriving the restricted change graph C' of the restricted change graph C and the critical tree pair $\mu = (D_x, D_y) \in TP(D_1, D_2)$ is denoted by $(C, \mu) \hookrightarrow_{EXTEND} C'$. The operational semantics of $\hookrightarrow_{EXTEND}$ is defined by the following inference rule.

$$\frac{\begin{array}{c}\mu \hookrightarrow_{EXPAND} S \\ V_A \coloneqq \{v \in V | v \in \Pi_C \wedge \mu \in v\} \\ A_S \coloneqq \left\{(v, v', \Delta') \middle| \begin{array}{l} v \in V_A \wedge (\Delta', \Theta') \in S \\ \wedge\, v' \coloneqq (v \backslash \{\mu\}) \cup \Theta' \end{array}\right\} \\ V_S \coloneqq \{v' | (v, v', \Delta') \in A_S\}\end{array}}{\big((V, A), \mu\big) \hookrightarrow (V \cup V_S, A \cup A_S)}$$

Table 13. Algorithm EXTEND

The algorithm $\hookrightarrow_{EXTEND}$ expands the change graph C by computing change scripts modulo for the critical tree pair $\mu = (D_x, D_y)$ using the algorithm $\hookrightarrow_{EXPAND}$. Let $(\Delta', \Theta') \in S$ be such a computed change script modulo, then the algorithm expands the change graph simultaneously at all active nodes $v \in \Pi_C$ which contain the critical tree pair, that is $\mu \in v$ holds. We denote this set of affected nodes in the algorithm $\hookrightarrow_{EXPAND}$ with V_A. An affected node v is expanded by adding an edge with the valid edit script Δ' to a node $v' \coloneqq (v \backslash \{\mu\}) \cup \Theta'$. The target node v' is either an already existing node or a new node that has to be added to the change graph. Altogether, the described simultaneous expansion strategy realizes the idea of *dynamic programming*.

We show now that the resulting change graph is indeed a restricted change graph.

Lemma 5.3.7 (Soundness of Restricted Change Graph Extension): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Let $C = (V, A)$ be a restricted change graph for D_1 and D_2, let $\mu \in TP(D_1, D_2)$ and $(C, \mu) \hookrightarrow_{EXTEND} C'$, then C' is a restricted change graph for D_1 and D_2.

Proof: We have to show that the properties of restricted change graph still hold for C'. Let $\mu = (t_1, t_2)$, for $\mu \hookrightarrow_{EXPAND} S$ we know by *Lemma 5.2.10* that for all $(\Delta, \Theta) \in S$ it holds either $\Theta = \emptyset$ or $\Theta \in \mathcal{E}^{\cong_{\Sigma_S}}_{(t_1,t_2)}$. Thus *Property 1* is satisfied by the construction of the new edges in the set A_S. From *Lemma 5.2.9* we know additionally that for all $(\Delta, \Theta) \in S$ it holds that $(\Delta, \Theta) \in \mathbb{O}^{\Sigma_E}_{\Sigma_S}(t_1, t_2, \Theta)$. Thus *Property 2* is also satisfied. Altogether, C' is a restricted change graph for D_1 and D_2.

■

We now proceed to the central algorithm for the change computation by search, the algorithm which defines the strategy of exploring the search space of limited change scripts modulo. For this purpose, we adapted Dijkstra's algorithm [Dijkstra, 1959] for the single-source shortest-path-to-goal problem. In our context, the length of a path is defined by the change path cost.

Definition 5.3.8 (Change Graph Search): Let Σ_S be a similarity specification, let Σ_E be an edit specification and let C be a restricted change graph. The judgment of deriving an optimal change script Δ for D_1 and D_2 with respect to Σ_S and Σ_E of the valid change graph $C = (V, A)$ is denoted by $C \hookrightarrow_{SEARCH} (\Delta, C')$. The operational semantics of $\hookrightarrow_{SEARCH}$ is defined by the following inference rules.

$$\frac{\begin{array}{c} v_x \in \Pi_C \\ \mu \in v_x \\ (C_1, \mu) \hookrightarrow_{EXTEND} C_2 \\ C_2 \hookrightarrow (\Delta, C_3) \end{array}}{C_1 \hookrightarrow (\Delta, C_3)} \qquad \frac{\emptyset \in \Pi_C}{C \hookrightarrow \left(\overline{\Delta}_C(\emptyset), C\right)}$$

Table 14. Algorithm SEARCH
(adapted from [Dijkstra, 1959])

This algorithm continuously expands the change graph at a node in the set of minimal cost active nodes Π_C until the goal node is a minimal cost active node. Note that there exist two choice points in this algorithm: (1) $v_x \in \Pi_C$ and (2) $\mu \in v_x$. We will discuss potential heuristics and their implications after completing the presentation and proofs for this algorithm. Let us now prove the termination, correctness and optimality of the change graph search algorithm.

Lemma 5.3.9 (Termination of Change Graph Search): Let D_1 and D_2 be labeled trees and let Σ_S be a similarity specification and let Σ_E be an edit specification. Let $C = (V, A)$ be a restricted change graph for D_1 and D_2. Then it holds that $C \hookrightarrow_{SEARCH} (\Delta, C')$ terminates.

Proof: By the definition of change graph we know that it consists of a finite set of nodes $V \subseteq \mathcal{P}\big(TP(D_1, D_2)\big)$ and a finite set of edges $A \subseteq V \times V \times D_1^{\Delta}$. By *Lemma 5.3.7* it follows that every inference step expands the restricted change graph to another restricted change graph. By the *Property 1* of restricted change graphs and *Lemma 5.1.7* it follows that for all edges $(v_1, v_2, \Delta) \in A$ the node size is decreasing, formally $W_C(v_1) > W_C(v_2)$. Hence, an inference step expands an active node v_x, that is, a node without outgoing edges, by adding at least one edge to the change graph and possibly nodes with a smaller size if they are not already contained in the change graph. Thereby, the goal node $\emptyset$ forms a lower bound with its node size 0. Thus, for any finite node size n there is a finite number of steps s such that after s inference steps the node size of all active nodes is smaller than n. Therefore, after finitely many steps we have $\Pi_C = \{\emptyset\}$ which clearly satisfies the termination condition. Altogether, it holds that $C \hookrightarrow_{SEARCH} (\Delta, C')$ terminates.

■

Note that in practice, the change graph search does not fully extend the change graph because the termination condition is usually satisfied earlier. Indeed, the change graph search does not expand nodes with a higher node size than the goal node.

We proceed by proving the correctness and optimality of change graph search.

Lemma 5.3.10 (Soundness and Completeness of Change Graph Search): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Let $C_1 = (V, A)$ be a restricted change graph for D_1 and D_2, and let $C_1 \hookrightarrow_{SEARCH} (\Delta, C_2)$. Then it holds that $\Delta \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_1, D_2)$.

Proof: Since $C_1 \hookrightarrow_{SEARCH} (\Delta, C_2)$ terminates by *Lemma 5.3.9*, C_2 is the last deduced change graph with $\Delta = \overline{\Delta}_{C_2}(\emptyset)$. By *Lemma 5.3.7* it follows that C_2 is a restricted change graph. By *Lemma 5.3.5* we know that all change path scripts are limited change scripts modulo. Furthermore, by the *Property 2* of restricted change graphs we know that in every step the generated edit script is an optimal change script modulo critical tree layer pairs of that pair of trees. Since we expand always a critical tree pair of a minimal cost active node, an optimal change path script of the active node $\emptyset$ is an optimal change script because of the optimality property of Dijkstra's algorithm. Assume to the contrary that there exists a limited change script Δ' with less costs, then there exists a path $p \in \mathcal{Y}_{C_2}(\emptyset)$ with $\Delta(p) = \Delta'$ because this path must have been extended by the change graph search. Since $\xi(\Delta) > \xi(\Delta')$ holds, this is a contradiction to $\Delta = \overline{\Delta}_{C_2}(\emptyset)$. Therefore, it holds that $\Delta \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_1, D_2)$.

■

It remains to initialize the search algorithm with an initial restricted change graph, which is simply a change graph consisting only of the root node and no edges.

Definition 5.3.11 (Differencing): Let D_1, D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. The judgment of deriving the optimal change script Δ for D_1 and D_2 with respect to Σ_S and Σ_E of the labeled trees D_1 and D_2 is denoted by $(D_1, D_2) \hookrightarrow_{DIFF} (\Delta, C)$. The operational semantics of $\hookrightarrow_{DIFF}$ is defined by the following inference rule.

$$\frac{C_1 \hookrightarrow_{SEARCH} (\Delta, C_2)}{(D_1, D_2) \hookrightarrow (\Delta, C_2)} \quad \begin{array}{l} \textit{with} \\ v_0 \coloneqq \{(D_1, D_2)\}, \\ C_1 \coloneqq (\{v_0\}, \emptyset) \end{array}$$

Table 15. Algorithm DIFF

The initial change graph $C_1 \coloneqq (\{v_0\}, \emptyset)$ contains only the root node $v_0 \coloneqq \{(D_1, D_2)\}$. Since the nodes of a change graph are sets of critical tree pairs, we can interpret the root node as the task to compute an optimal change script for the labeled trees D_1 and D_2.

Finally, we can combine all results to prove the termination, soundness and completeness of the presented differencing algorithm.

Theorem 5.3.12 (Termination, Soundness and Completeness of Differencing): Let D_1 and D_2 be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Then it holds that $(D_1, D_2) \hookrightarrow_{DIFF} (\Delta, \mathrm{C})$ terminates and that $\Delta \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_1, D_2)$.

Proof: The change graph $C_1 \coloneqq (\{v_0\}, \emptyset)$ with $v_0 \coloneqq \{(D_1, D_2)\}$ is clearly a restricted change graph because it contains only a root node and no edges. Hence by *Lemma 5.3.9* it holds that $C_1 \hookrightarrow_{SEARCH} (\Delta, \mathrm{C}_2)$ terminates and by *Lemma 5.3.10* it follows that $\Delta \in \mathbb{O}_{\Sigma_S}^{\Sigma_E}(D_1, D_2)$.

■

Let us summarize the results of this chapter. We presented an algorithm for computing an optimal change script between two labeled trees with respect to a similarity specification and an edit specification. Thereby, we reduced the constrained weighted tree alignment problem to a single-source shortest-path-to-goal problem. Furthermore, we adapted Dijkstra's algorithm to efficiently find an optimal solution in the search space of limited change scripts modulo, which is expanded dynamically by the presented algorithm.

In the description of the algorithm $\hookrightarrow_{SEARCH}$ in Table 14 we pointed out the two choice points (1) $v_x \in \Pi_C$ and (2) $\mu \in v_x$. We will discuss now potential heuristics and their implications. The choice point (1) selects a node in the change graph that is a minimal cost active node. A potential heuristic may select a node with minimal node size. Thus, if we for example expect very few and deep changes in a document, this heuristic helps to reduce the amount of extension steps required to find the goal node. In particular, if both documents are semantically equal, we directly extend the change graph towards the goal node using this heuristic. The choice point (2) selects a critical tree pair of the node selected by (1). A potential heuristic may select a critical tree pair with minimal size. Thus, analogously to (1) we can optimize the search strategy to account for relatively small changes between both documents. As an alternative to the described heuristics, we can also employ heuristics which extend specific labeled subtrees first, for example because we expect significant changes in these subtrees.

Altogether, both choice points allow for fine-tuning the strategy of exploring the search space. These choice points define essentially a second dimension of the search for an optimal change script and the decision can be roughly compared with choosing depth-first, breadth-first or custom-first comparison of the documents under consideration.

5.4 Use Case

Let us now illustrate the change graph search with a concrete example. We use our running example with the two documents D_1 and D_2 shown in Table 16. These documents contain two mathematical theories which both consist of one axiom and two theorems.

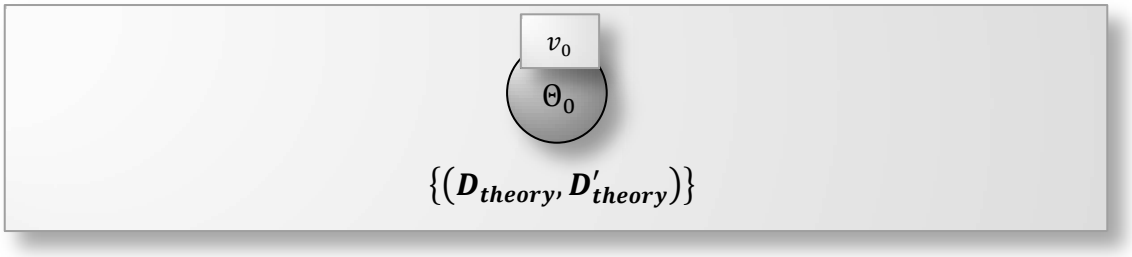
D_{theory}	\theory{}[	D'_{theory}	\theory{}[
D_{axiom}	\axiom{}[	$D'_{theorem1}$	\theorem{}[
	\name{}["AB"]],		\name{}["YZ"],
$D_{theorem1}$	\theorem{}[		\conj{}["#3"]],
	\name{}["XY"],	$D'_{theorem2}$	\theorem{}[
	\conj{}["#3"]],		\name{}["XY"],
$D_{theorem2}$	\theorem{}[		\conj{}["#1"]],
	\name{}["YZ"],	D'_{axiom}	\axiom{}[
	\conj{}["#2"]]]		\name{}["AB"]]]

Table 16. Comparing two mathematical documents D_1 (on the left) and D_2 (on the right)

We use the following similarity specification $\Sigma_S = (\Sigma_O, \Sigma_K)$. The similarity order Σ_O is the default $\forall l \in \mathcal{L}_A. \forall D \in \mathcal{S}(D_1). \Sigma_O(L(D), l)$. Furthermore, we define no similarity keys Σ_K except the following $\Sigma_K(\text{"}theorem\text{"}, \text{"}content\text{"}) = \Sigma_K(\text{"}axiom\text{"}, \text{"}content\text{"}) = \{\text{"}name\text{"}\}$. For the edit specification $\Sigma_E = (\Sigma_W, \Sigma_L)$ we use the payloads to define the edit weight. We define the delete payload of the axiom subtree D_{axiom} to be $\Upsilon_-(D_{axiom}) = 10$, because proofs in other theories depend on this axiom. Finally, we set an edit limitation for axioms and theorems globally to $\Sigma_L(D_x) = \top$ for all subtrees D_x in D_1 with $L(D_x) = \text{"}theorem\text{"}$ or $L(D_x) = \text{"}axiom\text{"}$. Thus, we restrict the granularity of the optimal change script. Altogether, we use the following specifications.

```
\similarity{}[
   \keys{ name="theorem", layer="content"}[ "name" ],
   \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[
   \weight{}[
     \delete{ path=/theory[1]/axiom[1] weight="10" }[] ],
   \limit{}[
     \global{ name="theorem" }[],
     \global{ name="axiom" }[] ] ]
```

The search for an optimal change script begins with the initial change graph $C_0 = (V, A) = (\{v_0\}, \emptyset)$ with $v_0 = \Theta_0 = \{(D_{theory}, D'_{theory})\}$. The minimal cost active nodes are $\Pi_{C_0} = \{v_0\}$. The algorithm selects $\mu_1 = (D_{theory}, D'_{theory}) \in v_0$ and calls $\hookrightarrow_{EXTEND}$ to extend the change graph by expanding the critical tree pair μ_1 with $\hookrightarrow_{EXPAND}$.



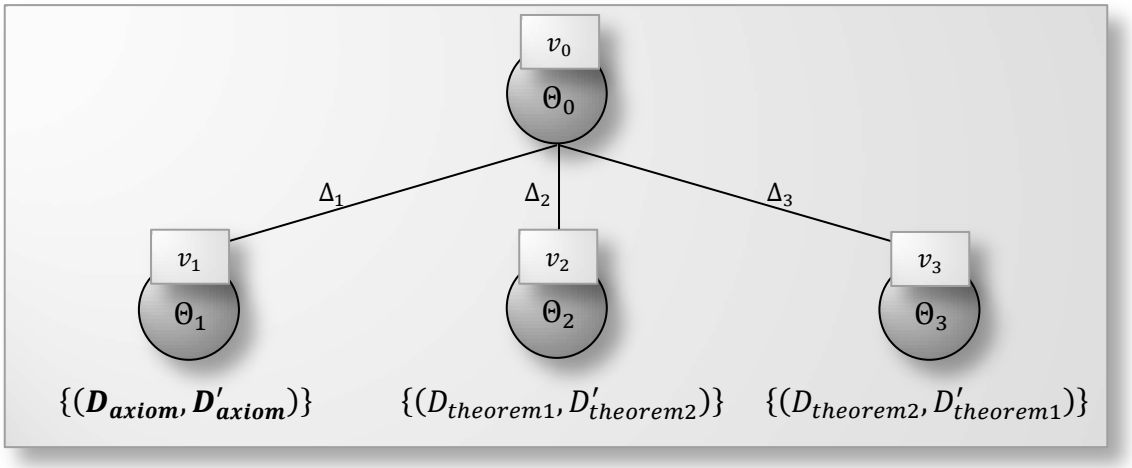
There, we have the case $\Sigma_L(D_{theory}) = \perp$ and $D_{theory} \cong_{\Sigma_S} D'_{theory}$ because both trees are equally labeled. Thus, we need to compute the set of extensible tree matching mappings $\mathcal{E}^{\cong_{\Sigma_S}}_{(D_{theory}, D'_{theory})}$. This set contains the following three mappings.

$$f_1 = \{(D_{axiom}, D'_{axiom})\}$$
$$f_2 = \{(D_{theorem1}, D'_{theorem2})\}$$
$$f_3 = \{(D_{theorem2}, D'_{theorem1})\}$$

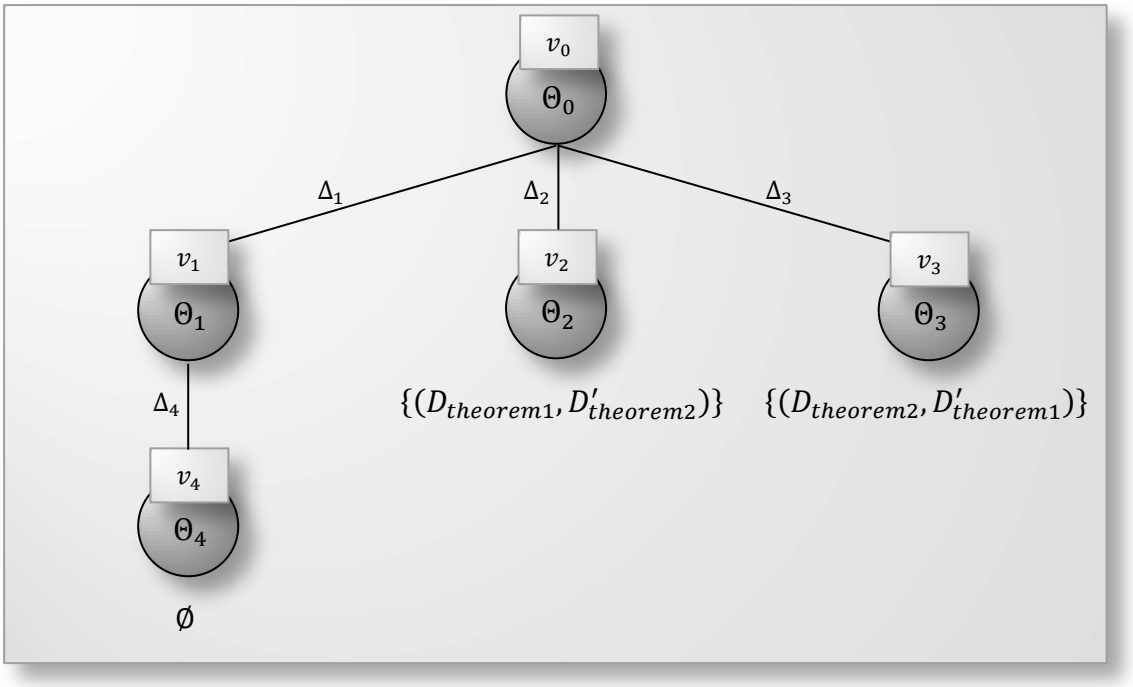
Using the algorithms $\hookrightarrow_{GENERATE}$ and $\hookrightarrow_{DELTA}$ we compute the following corresponding valid edit scripts.

$$\Delta_1 = \left[\delta_I\left(\overrightarrow{D_{axiom}}, [D'_{theorem1}, D'_{theorem2}]\right), \delta_E\left(\overrightarrow{D_{theorem1}}\right), \delta_E\left(\overrightarrow{D_{theorem2}}\right)\right]$$
$$\Delta_2 = \begin{bmatrix}\delta_I\left(\overrightarrow{D_{theorem1}}, [D'_{theorem1}]\right), \delta_A\left(\overrightarrow{D_{theory}}, \text{"}content\text{"}, [D'_{axiom}]\right), \\ \delta_E\left(\overrightarrow{D_{axiom}}\right), \delta_E\left(\overrightarrow{D_{theorem2}}\right)\end{bmatrix}$$
$$\Delta_3 = [\delta_A\left(\overrightarrow{D_{theory}}, \text{"}content\text{"}, [D'_{theorem2}, D'_{axiom}]\right), \delta_E\left(\overrightarrow{D_{axiom}}\right), \delta_E\left(\overrightarrow{D_{theorem1}}\right)]$$

Therefore, we have $\mu_1 \hookrightarrow_{EXPAND} \{(\Delta_1, f_1), (\Delta_2, f_2), (\Delta_3, f_3)\}$. The edges are extended by $A_{S1} = \{(v_0, v_1, \Delta_1), (v_0, v_2, \Delta_2), (v_0, v_3, \Delta_3)\}$ with the new nodes $V_{S1} = \{v_1, v_2, v_3\}$ where $v_i = \Theta_i = f_i$. Hence we have $(C_0, \mu_1) \hookrightarrow_{EXTEND} C_1$ with $C_1 = (V \cup V_{S1}, A \cup A_{S1})$.



In the second round of $\hookrightarrow_{SEARCH}$, the minimal cost active nodes are $\Pi_{C_1} = \{v_1\}$ because $\xi(\Delta_1) = 20$, $\xi(\Delta_2) = 26$ and $\xi(\Delta_3) = 26$. The algorithm selects $\mu_2 = (D_{axiom}, D'_{axiom}) \in v_1$ and calls $\hookrightarrow_{EXTEND}$ to extend the change graph by expanding the critical tree pair μ_2 with $\hookrightarrow_{EXPAND}$. There, we have the case $\Sigma_L(D_{axiom}) = \top$ and $D_{axiom} =_{\Sigma_S} D'_{axiom}$ because both trees are semantically equal. Thus, the algorithm returns $\mu_2 \hookrightarrow_{EXPAND} \{([\,], \emptyset)\}$. The edges are extended by $A_{S2} = \{(v_1, v_4, [\,])\}$ with the new node $V_{S2} = \{v_4\}$ where $v_4 = \Theta_4 = \emptyset$ and $\Delta_4 = [\,]$. Hence we have $(C_1, \mu_2) \hookrightarrow_{EXTEND} C_2$ with $C_2 = (V \cup V_{S2}, A \cup A_{S2})$.



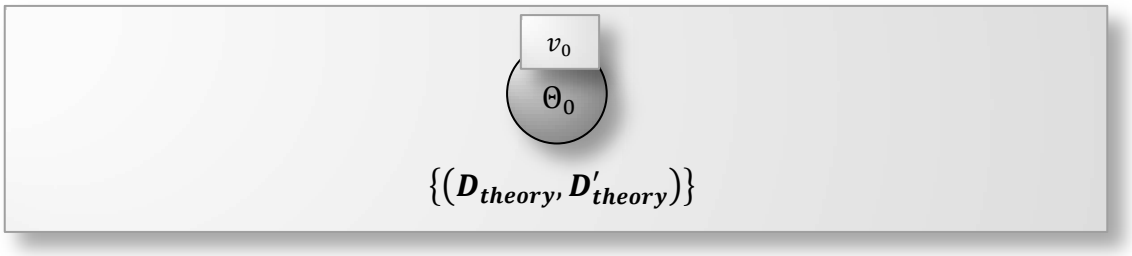
The third round of $\hookrightarrow_{SEARCH}$ is also the last round because the goal node $\emptyset$ is an element of $\Pi_{C_2} = \{\emptyset\}$. The algorithm finally returns the optimal change script $\Delta_X = \overline{\Delta}_{C_2}(\emptyset) = \Delta_1 \boxplus \Delta_4$.

$$\Delta_X = \left[\delta_I\left(\overrightarrow{D_{axiom}}, [D'_{theorem1}, D'_{theorem2}]\right), \delta_E\left(\overrightarrow{D_{theorem1}}\right), \delta_E\left(\overrightarrow{D_{theorem2}}\right)\right]$$

As a second example, we will compute an optimal change script for a slightly modified similarity specification. We postulate that the order of the children in the content layer of any theory subtree is not relevant, thus we have $\Sigma_O(\text{"}theory\text{"}, \text{"}content\text{"}) = \perp$. Let us now discover how this modification influences the change graph search. We use now the following specification.

```
\similarity{}[
   \order{ name="theory", layer="content"}[],
   \keys{ name="theorem", layer="content"}[ "name" ],
   \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[
   \weight{}[
     \delete{ path=/theory[1]/axiom[1] weight="10" }[] ],
   \limit{}[
     \global{ name="theorem" }[],
     \global{ name="axiom" }[] ] ]
```

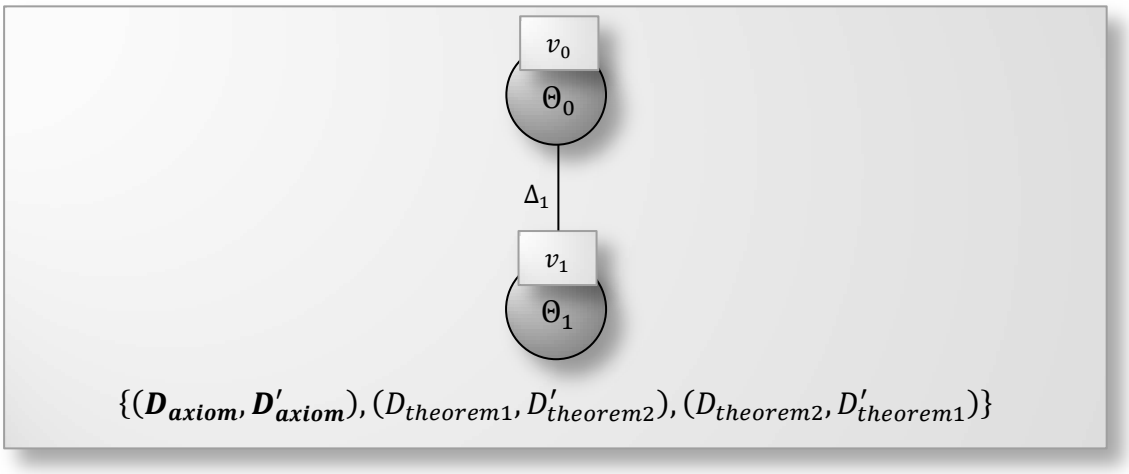
The search for an optimal change script begins again with the initial change graph $C_0 = (V, A) = (\{v_0\}, \emptyset)$ with $v_0 = \Theta_0 = \{(D_{theory}, D'_{theory})\}$. The minimal cost active nodes are $\Pi_{C_0} = \{v_0\}$. Thus, the algorithm selects $\mu_1 = (D_{theory}, D'_{theory}) \in v_0$ and calls $\hookrightarrow_{EXTEND}$ to extend the change graph.



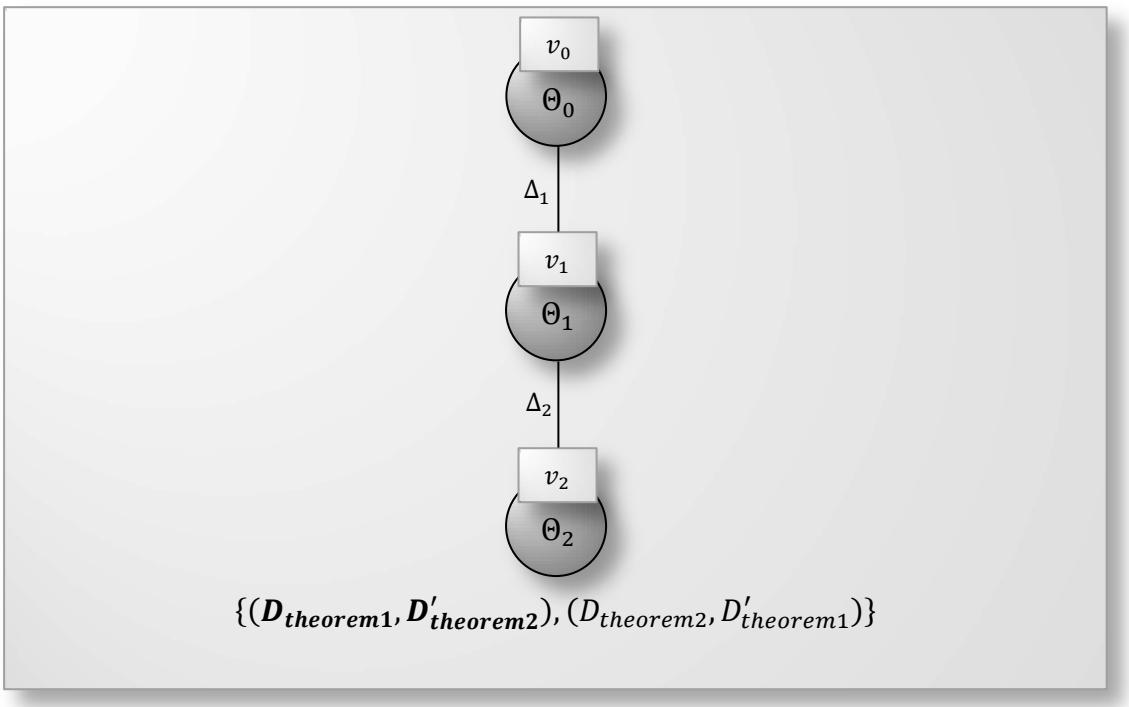
When expanding the critical tree pair μ_1 with $\hookrightarrow_{EXPAND}$, we have again the case $\Sigma_L(D_{theory}) = \perp$ and $D_{theory} \cong_{\Sigma_S} D'_{theory}$ because both trees are equally labeled. Thus, we need to compute the set of extensible tree matching mappings $\mathcal{E}^{\cong_{\Sigma_S}}_{(D_{theory}, D'_{theory})}$. This turn, the set contains only the following mapping.

$$f_1 = \{(D_{axiom}, D'_{axiom}), (D_{theorem1}, D'_{theorem2}), (D_{theorem2}, D'_{theorem1})\}$$

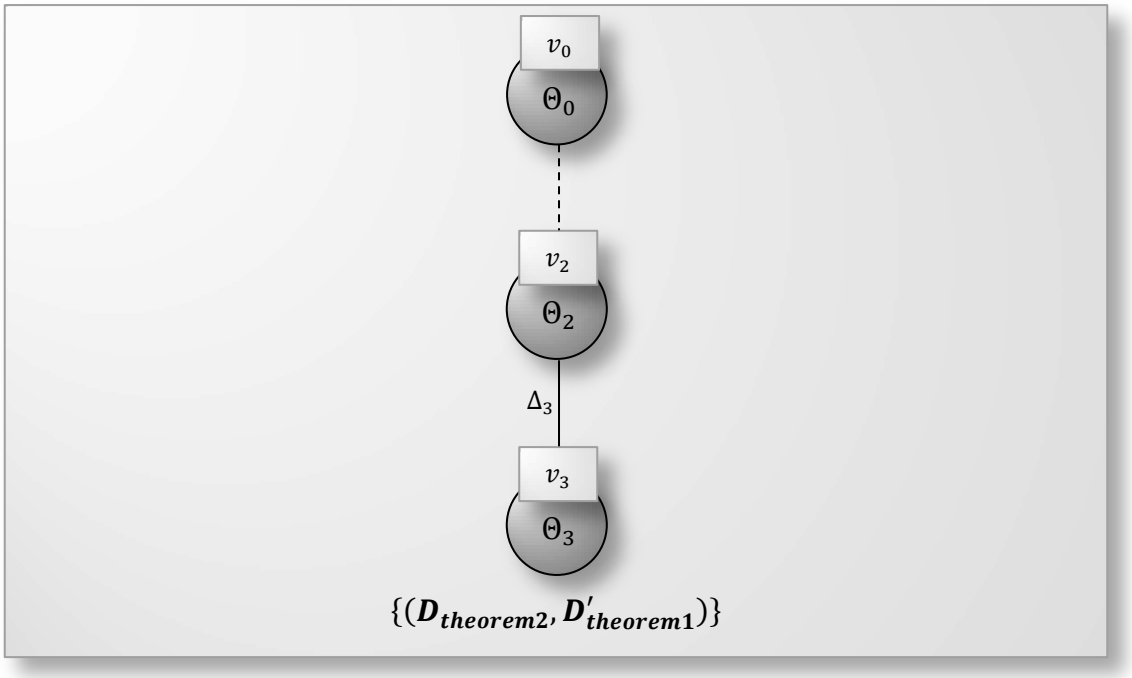
Using the algorithms $\hookrightarrow_{GENERATE}$ and $\hookrightarrow_{DELTA}$ we compute the corresponding valid edit script $\Delta_1 = [\,]$. Therefore, we have $\mu_1 \hookrightarrow_{EXPAND} \{(\Delta_1, f_1)\}$. The edges are extended by $A_{S1} = \{(v_0, v_1, \Delta_1)\}$ with the new node $V_{S1} = \{v_1\}$ where $v_1 = \Theta_1 = f_1$. Hence we have $(C_0, \mu_1) \hookrightarrow_{EXTEND} C_1$ with $C_1 = (V \cup V_{S1}, A \cup A_{S1})$.



In the second round of $\hookrightarrow_{SEARCH}$, the minimal cost active nodes are $\Pi_{C_1} = \{v_1\}$. The algorithm selects $\mu_2 = (D_{axiom}, D'_{axiom}) \in v_1$ but there are two other possible choices as well. We call $\hookrightarrow_{EXTEND}$ to extend the change graph by expanding the critical tree pair μ_2 with $\hookrightarrow_{EXPAND}$. There, we have the case $\Sigma_L(D_{axiom}) = \top$ and $D_{axiom} =_{\Sigma_S} D'_{axiom}$ because both trees are semantically equal. Thus, the algorithm returns $\mu_2 \hookrightarrow_{EXPAND} \{([\,], \emptyset)\}$. The edges are extended by $A_{S2} = \{(v_1, v_2, [\,])\}$ with the new node $V_{S2} = \{v_2\}$ where $v_2 = \Theta_2 = \{(D_{theorem1}, D'_{theorem2}), (D_{theorem2}, D'_{theorem1})\}$ and $\Delta_2 = [\,]$. Hence we have $(C_1, \mu_2) \hookrightarrow_{EXTEND} C_2$ with $C_2 = (V \cup V_{S2}, A \cup A_{S2})$.

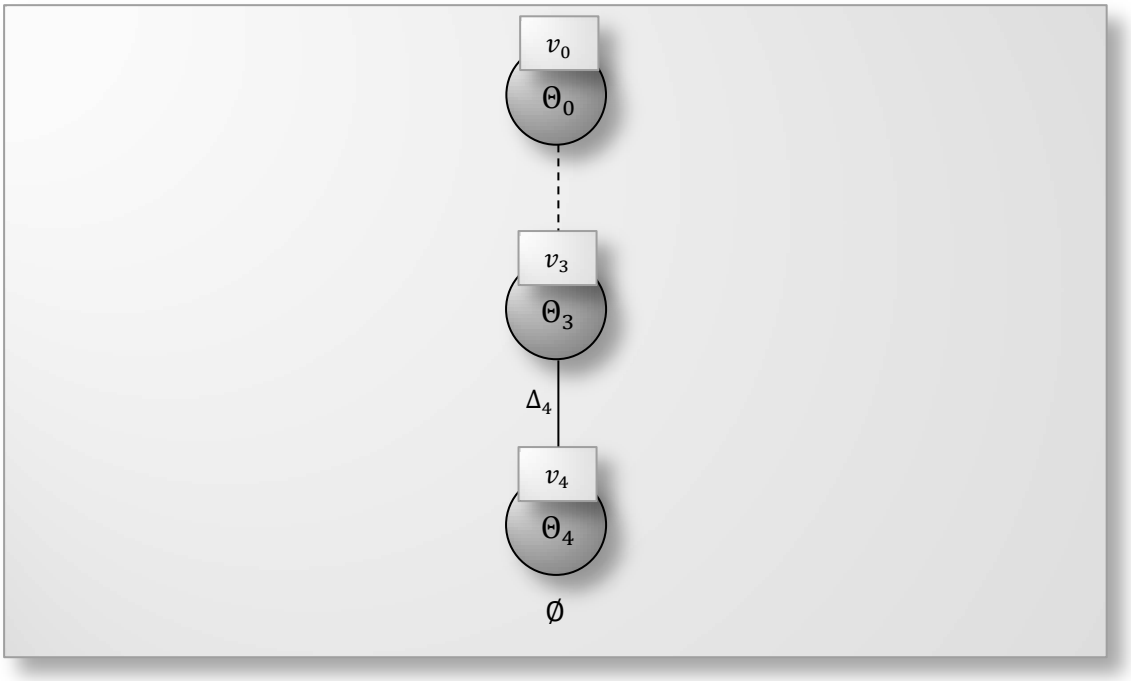


In the third round of $\hookrightarrow_{SEARCH}$, the minimal cost active nodes are $\Pi_{C_2} = \{v_2\}$. The algorithm selects $\mu_3 = (D_{theorem1}, D'_{theorem2}) \in v_2$ but there is another possible choice. We call $\hookrightarrow_{EXTEND}$ to extend the change graph by expanding the critical tree pair μ_2 with $\hookrightarrow_{EXPAND}$. There, we have the case $\Sigma_L(D_{theorem1}) = \top$ and $\neg\left(D_{theorem1} =_{\Sigma_S} D'_{theorem2}\right)$ because the "*conj*" subtrees of both trees are not semantically equal. Thus, the algorithm returns $\mu_3 \hookrightarrow_{EXPAND} \{(\Delta_3, \emptyset)\}$ with $\Delta_3 = [\delta_R(\overrightarrow{D_{theorem1}}, D'_{theorem2})]$. The edges are extended by $A_{S3} = \{(v_2, v_3, \Delta_3)\}$ with the new node $V_{S3} = \{v_3\}$ where $v_3 = \Theta_3 = \{(D_{theorem2}, D'_{theorem1})\}$. Hence we have $(C_2, \mu_3) \hookrightarrow_{EXTEND} C_3$ with $C_3 = (V \cup V_{S3}, A \cup A_{S3})$.



In the fourth round of $\hookrightarrow_{SEARCH}$, the minimal cost active nodes are $\Pi_{C_3} = \{v_3\}$. The algorithm selects $\mu_4 = (D_{theorem2}, D'_{theorem1}) \in v_3$ as the only possible choice. We call $\hookrightarrow_{EXTEND}$ to extend the change graph by expanding the critical tree pair μ_4 with $\hookrightarrow_{EXPAND}$. There, we have the case $\Sigma_L(D_{theorem2}) = \top$ and $\neg\left(D_{theorem2} =_{\Sigma_S} D'_{theorem1}\right)$ because the "*conj*" subtrees of both trees are not semantically equal.

Thus, we have $\mu_4 \hookrightarrow_{EXPAND} \{(\Delta_4, \emptyset)\}$ with $\Delta_4 = [\delta_R(\overrightarrow{D_{theorem2}}, D'_{theorem1})]$. The edges are extended by $A_{S4} = \{(v_3, v_4, \Delta_4)\}$ with the new node $V_{S4} = \{v_4\}$ where $v_4 = \Theta_4 = \emptyset$. Hence we have $(C_3, \mu_4) \hookrightarrow_{EXTEND} C_4$ with $C_4 = (V \cup V_{S4}, A \cup A_{S4})$.



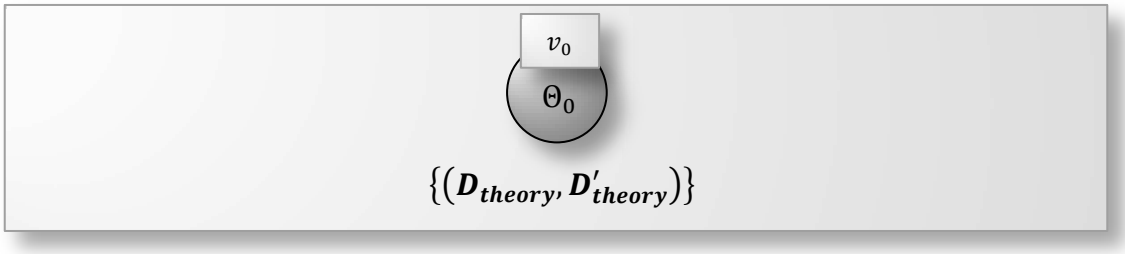
The fifth round of $\hookrightarrow_{SEARCH}$ is the last round because the goal node $\emptyset$ is an element of $\Pi_{C_4} = \{\emptyset\}$. The algorithm finally returns the optimal change script $\Delta_X = \overline{\Delta}_{C_4}(\emptyset) = [\,] \boxplus [\,] \boxplus \Delta_3 \boxplus \Delta_4$.

$$\Delta_X = \left[\delta_R(\overrightarrow{D_{theorem1}}, D'_{theorem2}), \delta_R(\overrightarrow{D_{theorem2}}, D'_{theorem1})\right]$$

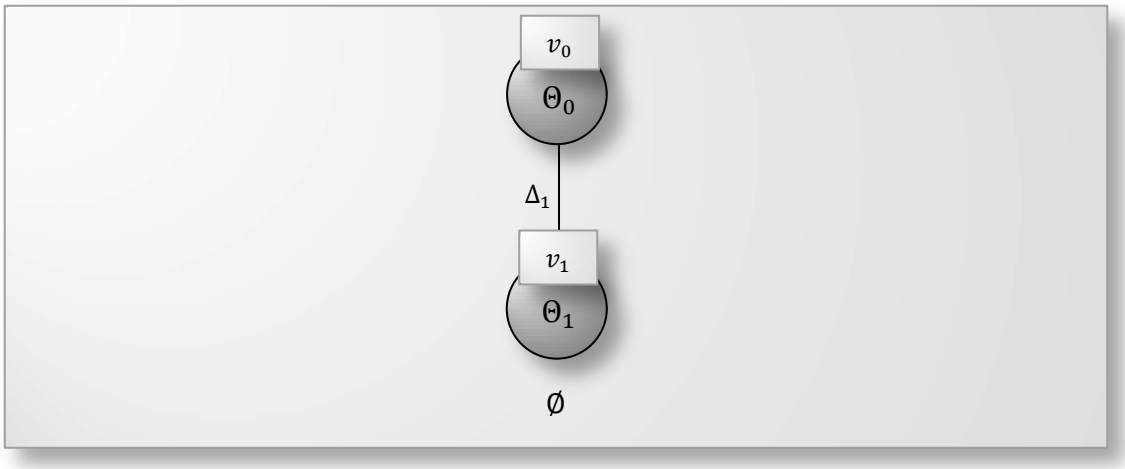
As a third and last example, we will demonstrate the efficiency gain by using the edit limitation. We consider the similarity specification with the default similarity order $\forall l \in \mathcal{L}_A. \forall D \in \mathcal{S}(D_1). \Sigma_O(L(D), l)$. Furthermore, as in the examples before we define no similarity keys except the following $\Sigma_K(\text{"}theorem\text{"}, \text{"}content\text{"}) = \Sigma_K(\text{"}axiom\text{"}, \text{"}content\text{"}) = \{\,\text{"}name\text{"}\,\}$. For the edit specification $\Sigma_E = (\Sigma_W, \Sigma_L)$ we use no additional payloads to define the edit weight. Finally, we set an edit limitation for theories globally to $\Sigma_L(D_x) = \top$ for all subtrees D_x in D_1 with $L(D_x) = \text{"}theory\text{"}$. Thus, we restrict the granularity of the optimal change script to the topmost subtree.

```
\similarity{}[
   \keys{ name="theorem", layer="content"}[ "name" ],
   \keys{ name="axiom", layer="content"}[ "name" ] ]
\edit{}[
   \limit{}[
     \global{ name="theory" }[] ] ]
```

The search for an optimal change script begins again with the initial change graph $C_0 = (V, A) = \left(\{v_0\}, \emptyset, D_{theory}, D'_{theory}\right)$ with $v_0 = \Theta_0 = \left\{\left(D_{theory}, D'_{theory}\right)\right\}$.



The minimal cost active nodes are $\Pi_{C_0} = \{v_0\}$. Thus, the algorithm selects $\mu_1 = \left(D_{theory}, D'_{theory}\right) \in v_0$ as the only possible choice and calls $\hookrightarrow_{EXTEND}$ to extend the change graph by expanding the critical tree pair μ_1 with $\hookrightarrow_{EXPAND}$. There, we have this time the case $\Sigma_L\left(D_{theory}\right) = \top$. The algorithm is able to immediately verify that the condition $\neg(D_{theory} =_{\Sigma_S} D'_{theory})$ holds, because the first child in the content layer of both trees is not semantically equal. Thus, the algorithm returns $\mu_1 \hookrightarrow_{EXPAND} \{(\Delta_1, \emptyset)\}$ with $\Delta_1 = [\delta_R(\overrightarrow{D_{theory}}, D'_{theory})]$. The edges are extended by $A_{S1} = \{(v_0, v_1, \Delta_1)\}$ with the new node $V_{S1} = \{v_1\}$ where $v_1 = \Theta_1 = \emptyset$. Hence we have $(C_0, \mu_1) \hookrightarrow_{EXTEND} C_1$ with $C_1 = (V \cup V_{S1}, A \cup A_{S1})$.



In the second and last round of $\hookrightarrow_{SEARCH}$, the goal node $\emptyset$ is already an element of $\Pi_{C_1} = \{\emptyset\}$. Finally, the algorithm returns the optimal change script $\Delta_X = \overline{\Delta}_{C_1}(\emptyset) = \Delta_1$.

$$\Delta_X = \left[\delta_R(\overrightarrow{D_{theory}}, D'_{theory})\right]$$

5.5 Complexity Comparison

In this section, we will compare the time and space complexity of the change graph search with the state-of-the-art algorithm given in [Radzevich, 2006]. Since we improved and extended the definition of the semantic tree alignment problem, a direct comparison of both algorithms is not adequate. Therefore, we first adapt the state-of-the-art algorithm such that it uses the $\hookrightarrow_{EXPAND}$ algorithm for generating all extensible tree matching mappings together with corresponding optimal tree layer change scripts with respect to a given similarity and edit specification. The adapted algorithm is shown in Table 17 and defined in the context of a similarity specification Σ_S and an edit specification Σ_{E}.

$$\frac{\begin{array}{c}(D_x, D_y) \hookrightarrow_{EXPAND} S \\ S \hookrightarrow_{RDIFF} G \\ \Delta_{min} \in G \\ \forall \Delta \in G.\, \xi(\Delta) \geq \xi(\Delta_{min})\end{array}}{(D_x, D_y) \hookrightarrow_{SDIFF} \Delta_{min}}$$

$$\frac{\begin{array}{c}R := \{\Delta_{\mathrm{x}} | \mu \in \Theta \wedge \mu \hookrightarrow_{SDIFF} \Delta_{\mathrm{x}}\} \\ \Delta_R := \boxplus_{\Delta_{\mathrm{x}} \in R} \Delta_x \\ S \hookrightarrow_{RDIFF} G\end{array}}{\{(\Delta, \Theta)\} \cup S \hookrightarrow_{RDIFF} \{\Delta \boxplus \Delta_R\} \cup G}$$

$$\frac{}{\emptyset \hookrightarrow_{RDIFF} \emptyset}$$

Table 17. Algorithms SDIFF and RDIFF

The main idea of the algorithm $\hookrightarrow_{SDIFF}$ is (1) to compute all possible extensible tree matching mappings, for which we use the algorithm $\hookrightarrow_{EXPAND}$, then (2) to compute the corresponding change scripts recursively, which is done by the algorithm $\hookrightarrow_{RDIFF}$, and finally (3) to return a computed change script with minimal edit cost.

In the spirit of *dynamic programming*, it is clearly possible to reuse computed results for similar cases. We will now analyze whether reusing previous results is in these cases reasonable from the point of view of efficiency. The ideal candidate for reusing results is the judgment $(D_x, D_y) \hookrightarrow_{SDIFF} \Delta_{min}$. Depending on the concrete case, the algorithm $\hookrightarrow_{EXPAND}$, which is called by $\hookrightarrow_{SDIFF}$, computes either $\neg(D_x =_{\Sigma_S} D_y)$ and a replacing change script, or for every tree layer recursively the set of extensible tree matching mappings $\mathcal{E}^{\cong_{\Sigma_S}}_{(D_x, D_y)}$ and optimal tree layer change scripts. The time and space complexity of all these cases is approximately less or equal to one single successful test for semantic equality $D_x =_{\Sigma_S} D_y$. But in order to reuse the result of $(D_x, D_y) \hookrightarrow_{SDIFF} \Delta_{min}$ for a pair of trees (D'_x, D'_y), we need to perform two successful tests for semantic equality $D'_x =_{\Sigma_S} D_x$ and $D'_y =_{\Sigma_S} D_y$. Thus, reusing computed results for similar cases is not an efficient approach.

Nevertheless, it is of course both possible and efficient to reuse computed results for the exact same case, where we just have to compare the references to the trees provided as arguments.

For the change graph search, this benefit from reuse has been made explicit in the definition of the algorithm $\hookrightarrow_{EXTEND}$ in Table 13. From this observation and the full recursive definition of the algorithm $\hookrightarrow_{SDIFF}$ we can conclude that the algorithm $\hookrightarrow_{SDIFF}$ always expands the complete search space of limited change scripts modulo. Thus, the time and space complexity for the algorithm $\hookrightarrow_{SDIFF}$ is always the same for all cases: the best case, the average case and the worst case. Furthermore, this time and space complexity is an upper bound for our change graph search algorithm $\hookrightarrow_{DIFF}$.

The interesting question is whether our algorithm $\hookrightarrow_{DIFF}$ performs even better in the average case than in the worst case. Without further information about the concrete shape of an average tree, we cannot precisely analyze the average time and space complexity. What we can actually analyze is whether there exists a relationship between the degree of the extension of the change graph and the properties of an optimal change script.

Let us assume the change graph search has computed an optimal change script by $(D_1, D_2) \hookrightarrow_{DIFF} \Delta$. Before returning the optimal change script Δ, the algorithm $\hookrightarrow_{DIFF}$ has constructed and extended a restricted change graph C. By the properties of Dijkstra's algorithm, we know that no limited change script modulo has been expanded that has greater edit cost than the returned optimal change script. This does not prove but it strongly indicates that the change graph search is very efficient for computing an optimal change script for two semantically equal or almost equal documents, where we expect an optimal change script with low edit cost.

Let us assume that we use heuristics for the algorithm $\hookrightarrow_{SEARCH}$ which select in the choice point (1) $v_x \in \Pi_C$ a minimal cost active node with minimal node size and in the choice point (2) $\mu \in v_x$ a critical tree pair with minimal size. We assume further that these heuristics need on average asymptotically $\log n$ attempts out of n possibilities to select an extensible tree matching mapping that can be extended to show the semantic equality or almost equality of two documents. Then, the time and space complexity for the change graph search is indeed on average even quasi-linear for computing an optimal change script for two semantically equal or almost equal documents.

Since the time and space complexity of the algorithm $\hookrightarrow_{SDIFF}$ is an upper bound for the complexity of our algorithm $\hookrightarrow_{DIFF}$, the change graph search is always at least as efficient as the state-of-the-art algorithm. In the case of two semantically equal or almost equal documents, which is the predominant case in the context of a *Change-Oriented Architecture,* the change graph search outperforms the state-of-the-art algorithm by an order of magnitude.

5.6 Discussion

In this chapter, we presented a new algorithm for the constrained weighted tree alignment problem that is more efficient than the state-of-the-art algorithm in [Radzevich, 2006]. The essential idea of our approach is to reduce the tree alignment problem to a search problem. We introduced the notion of change scripts modulo critical tree pairs where the critical tree pairs are the remaining tasks to be solved by the search algorithm. The change graph search proceeds by successively extending limited change scripts modulo by an optimal change script modulo for a critical tree pair in focus. In our context, the optimization problem is a single-source shortest-path-to-goal problem where the change graph can only be dynamically expanded and the length of a path is defined by the change path cost.

We adapted Dijkstra's algorithm for exploring the search space dynamically with a correct and optimal strategy. Since this algorithm is a special case of the A^* algorithm with the constant heuristic $h = 0$, an obvious question is whether there exists an admissible non-constant heuristic. This is not the case, because any critical tree pair might turn out during the search process to be already semantically equal. Thus the optimal change script for these trees would be the empty change script with the minimum edit cost 0. Since an admissible heuristic is not allowed to overestimate the real cost, it follows that the constant heuristic $h = 0$ is the only admissible heuristic. Thus, we cannot further restrict the search space. Since the A^* algorithm is the computationally optimal search strategy with respect to a given heuristic, and since there exist no other admissible heuristics, our adaptation of Dijkstra's algorithm is the computationally optimal search strategy for our reduction of the constrained weighted tree alignment problem to a search problem.

This leads immediately to the interesting question whether there are other possibilities to reduce the constrained weighted tree alignment problem to a search problem. We decided in our approach to start the search at the root of both trees and to proceed in a top-down fashion by comparing the subtrees of these trees. The reason for this decision is mainly the change model: Since we do not support the move operation, we do not need to compare arbitrary subtrees on different layers in the hierarchy of the trees.

Finally, we want to emphasize that we developed an efficient solution for a general class of constrained weighted tree alignment problems, which can be instantiated by a domain- and document-specific similarity specification and a component-specific edit specification. Besides that level of generality, the developed algorithm only depends on the document model, the change model, the semantics of the similarity specification and the semantics of the edit specification.

6 Invertible Grammar Formalism

The *Change-Oriented Architecture* requires a robust method to translate between two different interface documents in both directions. In this chapter we will therefore develop a new formalism for invertible transformation grammars which is based on the introduced notions for the semantic equality and the semantic similarity of documents.

The content diversity of interface documents ranges from serialized data structures to full natural language. Therefore, we have chosen the following two mature grammar models as the foundation of the invertible grammar formalism which we will present in this chapter. As a representative for flexible parser style grammar models we selected *attribute grammars* [Knuth, 1968] which have a long track record in the translation of programming languages. As a representative for flexible generator style grammar models we selected *TGL*, the grammar model of the natural language production system *TG/2* [Busemann, 1996]. Both grammar models are context-sensitive extensions of the *context-free grammar model* [Chomsky, 1957]. We will analyze their differences before we present our derived invertible grammar model. Due to the aim of inverting the grammar for the inverse transformation of an interface document, we have to restrict the expressive power of the context-sensitive extensions in our grammar model. The solution we propose is a compromise between totally invertible content-permuting copy rules and partially invertible full-fledged transformation rules with side-effects.

This chapter begins by introducing the invertible grammar model. This new model significantly differs from the previous models by integrating the notion of semantic equality in the pattern matching method for grammar rules. Furthermore, we augment the grammar rules by unification constraints which need to be satisfied by the constructed parse tree. In addition, the invertible grammar model allows for restructuring the matched input for recursive transformation and it allows for specifying the processing order of the recursive transformations. Since we do not assume a one-to-one correspondence between the contents of two transformed interface documents, we store a transformation trace which we use for the incremental transformation and in particular for the incremental inverted transformation. The idea of the transformation trace is to store the used rules together with their input and output matching mapping, and thus to store in particular those parts of the interface document that are not preserved by the transformation. Then, the transformation trace is used by the inverse transformation process as an oracle for generating content which is as close as possible to the original content. Finally, we will illustrate the presented invertible grammar formalism with a use case.

6.1 Grammar

As proposed by Noam Chomsky in the 1950s, a formal grammar consists of a finite set of symbols and a finite set of production rules. The symbols are partitioned into terminal symbols and nonterminal symbols, where one of the nonterminal symbols is marked as the start symbol of the grammar. A production rule is of the form $A_1 \dots A_n \twoheadrightarrow B_1 \dots B_m$ where all A_i and B_k are symbols and at least one A_i is a non-terminal. A production rule allows for rewriting every occurrence of its left side to its right side.

Chomsky described a containment hierarchy of classes of formal grammars in [Chomsky, 1956] by placing restrictions on the form of the production rules. For the purpose of this thesis, we will focus on the class of *context-free grammars* (*CFGs*). A *CFG* is a formal grammar with the restriction that the left side of every production rule has length one. Thus, a production rule of a *CFG* may only rewrite one nonterminal symbol into a sequence of symbols. This conceptual simplicity leads to an intuitive tree representation of the transformation process, the *context-free parse tree*. Furthermore, this explains the broad acceptance of the *CFG* model as a grammar model for various tasks.

When using *CFGs* to describe programming languages, there is the problem of modeling context-dependent language features like static typing and static scoping. To this purpose, various extensions of the *CFG* model have been developed. A popular example is *attribute grammars* (*AGs*) [Knuth, 1968], developed by Knuth as a *CFG* model extended by an attribute system. Thereby, the nodes in the context-free parse tree are decorated with attribute-value pairs. The attribute system extends each production rule by a set of formulas which define the dependencies between the attributes of a parent node and the attributes of its children. Knuth distinguishes between *inherited attributes*, whose values propagate down the parse tree, and *synthesized attributes*, whose values propagate up the parse tree. In the years after, a family of attribute grammar variant models have been developed, whose members use different kinds of attributes and attribute evaluation methods. For example, the *door attribute grammars* model [Hedin, 1994] extends the context-free parse tree by symbol tables, type environments and collection-valued attributes, which are incrementally evaluated by using visiting procedures and a dependency graph.

In the research of an adequate grammar model for natural language, Chomsky's work inspired the development of several phrase structure grammar models, among others the *generalized phrase structure grammar GPSG* [Gazdar *et al*, 1985], the *lexical functional grammar LFG* [Kaplan & Bresnan, 1981], and the *head-driven phrase structure grammar HPSG* [Pollard & Sag, 1994]. *HPSG* is a generative grammar theory which introduces feature structures. These are nested attribute-value matrices for representing phonological, syntactical and semantic information, from which natural language is generated.

Two orthogonal methods have been pursued for the development of efficient natural language generation systems: (1) reusable, general and linguistically motivated techniques, and (2) simple, task-oriented template-based techniques. The natural language production system *TG/2* [Busemann, 1996] proposes a compromise between these two extreme approaches. With many years of practical application [Busemann, 2005] ranging from the generation of air-quality reports [Busemann & Horacek, 1998] to cross-lingual document summarization [Busemann, 2001], *TG/2* has proven to be well-suited as a flexible shallow approach to the generation of natural language with limited linguistic variations.

For the natural language surface generation, *TG/2* uses production rules specified in its grammar model *TGL*. A rule in *TGL* is a precondition-action pair where the precondition contains the category of a rule, the left side in a *CFG* rule, and tests for the input structure, predicates written in a programming language. The action of a *TGL* rule reflects the right side in a *CFG* rule with the small but significant difference that in *TGL* the nonterminal symbols are declaratively parameterized by the input subtree which they have to transform. On the one hand, multiple nonterminal symbols may transform the same input subtree. On the other hand, the assigned input subtree does not have to be a direct subtree of the input tree, *TGL* allows also for assigning a subtree which is deep in the input tree to simplify the generation of natural language from foreign data structures. Additionally, the rules are augmented by side-effects, functions written in a programming language, and constraints which decorate the nodes in the context-free parse tree with attribute-value pairs, similar to *AG* but restricted to value equations and assignments. We summarize the differences between the production rules used by *AG* and *TGL* in Table 18.

	AG	TGL
Preconditions	- only category check (= left side nonterminal)	- category check - test predicates on input
Actions	- discovering an order-preserving assignment of subtrees to nonterminals is the goal - no multiple assignments - only assignment of the direct subtrees - complete coverage required	- declarative assignment of a subtree to a nonterminal in arbitrary order - multiple assignments of same subtree - assignment of arbitrary deep subtrees - no complete coverage required
Side-Effects	- part of attribute evaluation	- designated functions
Constraints	- arbitrary semantic functions and data structures	- value equations - value assignments
Output	- value of a designated result attribute	- concatenation of all action results

Table 18. Differences between production rules in AG and TGL

In the following, we will analyze these differences with respect to the inversion of a grammar. In the spirit of [Shieber, 1988], we will propose to use a single grammar and a single interpreter as a unified approach for the bidirectional translation of documents.

Preconditions. Although *TGL* supports arbitrary test predicates on the input tree, this grammar feature is in practice mainly used for checking the existence of specific attributes or attribute-value pairs in the input feature structure. Inverting such a test predicate would require the automatic generation or adaptation of an output subtree in order to satisfy the test predicate. Since this task is more difficult than it sounds, we propose an alternative solution. We will consider the right side of a *CFG* rule as a pattern with the nonterminals being variables and the terminals being labeled trees. The input tree or tree sequence will be matched against the pattern using the notion of semantic equality. Thereby, we will introduce positive and negative filters for variables in order to compensate for the missing test predicates. These filters will allow for specifying the kinds of labeled trees which a variable is allowed to match or not allowed to match.

Actions. The envisaged invertible grammar model needs to support the parser style as well as the generator style. Preferring either one of the styles for actions will make the job more difficult for modeling the other style. Therefore, we propose to combine both approaches in the following two step style. The pattern of the rule is first matched against the input tree or tree sequences in parser style, which means every direct subtree of the input is assigned to one and only one symbol in the pattern with respect to the order of the input and the similarity specification. In a second step, the subtransformation tasks are defined by a rule category parameterized by a sequence of labeled trees containing variables from the matched input pattern. Thus, we go even further than the generator style of *TGL* and allow a restructuring of the input tree.

Analogously to the generator style, we do not require that all input subtrees, matched by a pattern variable, have to be processed by a subtransformation. However, the inversion of a rule with such an incomplete coverage requires an oracle to construct the content of all uncovered pattern variables. We will present a method which solves this problem adequately. Does the proposed two step style support both original styles? Is it fully backward compatible? The parser style of the *AG* model can be imported without further adaptations by defining a subtransformation for every pattern variable and parameterizing this subtransformation by the input subtrees matched with that pattern variable. Unfortunately, the generator style of the *TGL* model cannot be imported that easily because a rule may access any part of the input tree. These access points have to be analyzed and either targeted by a pattern variable or transported by using the constraint system.

Side-Effects. Grammars in the *TGL* model use the side-effect function to access a discourse memory. Likewise, grammars in the *AG* model use functions with side-effects in the attributes to access a scoped environment for bounded expressions and type information. Since side-effects are in general to be handled carefully when backtracking is required, we prefer to avoid them completely if possible. Therefore, we will allow for defining specific constraints that can be attached to each grammar rule. These constraints are evaluated partly at the beginning and partly at the end of a rule and they are visible to all subsequent transformations. Furthermore, we will introduce a semantic hash function that maps semantically equal labeled trees to the same hash value. A built-in identity grammar rule returns the hash value of the input tree as an attribute value in the constraints. In combination with meta-variables this can be used to model static scoping.

Constraints. Knuth originally introduced the attribute system as an extension that has to be evaluated in a separate process after the context-free parse tree is completely built. Since we want to use the constraints for type checking and since the *TGL* model uses the constraints for agreement relations between constituents, we will develop an incremental constraint evaluation method and integrate it with the transformation process. As suggested by *definite clause grammars* [Pereira & Warren, 1980], there is in this case no need to distinguish between inherited and synthesized attributes. Since value equations and value assignments satisfy most needs for constraints, we will use our notion of semantic equality as the fundamental equivalence relation for constraints. Additionally, there exists no specific pass-oriented evaluation [op den Akker *et al*, 1990] for our grammar model, because we will allow for defining a rule-specific processing order of subtransformations.

Output. The result construction in generator style is a fairly simple task of concatenating the results of the subtransformations. In contrast to that, the parser style of the *AG* model allows for using arbitrary functions to construct the result. Clearly, we have to restrict the expressive power of the *AG* model in this dimension for the grammar inversion. Therefore, we propose to use a mirrored two step style for specifying the output of a grammar rule as follows. First we define an output pattern containing labeled trees with variables. In a second step, we add to each subtransformation task an output pattern containing variables from the output pattern of the rule. The result of a subtransformation is then matched against its output pattern and the resulting mapping is used to successively construct the output of the rule. If a variable in the output pattern is assigned to multiple subtransformations, then their matched content has to be semantically equal. Altogether, we believe that with the recombination possibilities provided by this mirrored two step style our grammar model inherits the benefits of the *TGL* model for reusing rules.

After this analysis of grammar models and the high-level introduction of the new concepts for the invertible grammar formalism, we will begin with developing the formal theory of our invertible grammar model by first introducing a representation of variables as labeled trees, which will be used in the constraints and patterns of grammar rules.

Definition 6.1.1 (Variable): Let $\mathcal{L}_{VAR}$ be the universal set of variable labels. A *variable* is a labeled tree D with a root label $l \in \mathcal{L}_{VAR}$ and no children. The *set of all variables* is denoted by $\mathcal{V} \coloneqq \{D \in \mathcal{D} | L(D) \in \mathcal{L}_{VAR} \wedge \mathcal{C}(D) = \emptyset\}$. The variables contained in all subtrees of a sequence of labeled trees Y is the set $vars(Y) \coloneqq \{D | D \in \mathcal{S}(Y) \wedge D \in \mathcal{V}\}$. Let $l \in \mathcal{L}_{VAR}$ be the label of a variable, then $var(l)$ returns the corresponding variable tree.

As described in the introduction, we allow for defining filters on variables in order to restrict the possible matching partners of a variable. The following specification allows for defining a positive and negative filter of a variable, that is, the kinds of labeled trees this variable is allowed to match or not to match. Furthermore, we can define whether a variable is only allowed to match exactly one labeled tree or at least one labeled tree.

Definition 6.1.2 (Variable Specification): Let D be a variable with the label $l = L(D)$ and $l \in \mathcal{L}_{VAR}$. The *positive filter* $Z_+(l) \in \mathcal{P}(\mathcal{L}_V)$ is a finite set of node labels. If this set is non-empty, a tree matched by the variable D has to have a root label which is contained in the positive filter. The *negative filter* $Z_-(l) \in \mathcal{P}(\mathcal{L}_V)$ is also a finite set of node labels. A tree matched by the variable D is not allowed to have a root label which is contained in the negative filter. The *matching range* $\Lambda(l) \in \{\top, \bot\}$ is a Boolean value which indicates whether the variable can match one or more labeled trees ($\top$) or only exactly one labeled tree ($\bot$). A *variable specification* is a triple $\Sigma_V = (Z_+, Z_-, \Lambda)$ consisting of a positive filter, a negative filter and a matching range.

Note that the filters and matching range are defined on the labels of variables. This implies that the restrictions hold for all occurrences of a variable with a specific label. By analyzing the occurrences of a specific variable in the patterns of the grammar rules, we will be able to compile additional delimiters for the matching partners of a variable. A serialized representation of a variable specification is given by the following example.

```
\variables{}[
    \positive{ name="X" }[ "+" ],
    \negative{ name="Y" }[ "." ],
    \negative{ name="A" }[ "theorem", "axiom" ],
    \multirange{}[ "X", "Y", "Z" ] ]
```

By default, the positive filter of any variable is defined to be empty. In this example we have defined the positive filter of the variable X to be the set of labels { "+" }. This implies that the variable X can only match trees with the root label " + ". Furthermore, the negative filter of any variable is also empty by default. Here we have defined that the variable Y is not allowed to match a tree with label "." and that the variable A is not allowed to match trees with the label "$theorem$" or "$axiom$". Finally, all variables are by default allowed to match exactly one labeled tree. In this example, we have overridden this default range for the variables X, Y and Z such that they may match one or more labeled trees.

According to the semantics of the variable specification we define the validation of the matching partner of a variable as follows.

Definition 6.1.3 (Valid Variable Matching Partner): Let D_1 and D_2 be two labeled trees with $l_1 = L(D_1)$ and $l_2 = L(D_2)$, and let $\Sigma_V = (Z_+, Z_-, \Lambda)$ be a variable specification. The judgment $D_1 \hookrightarrow_{MAP} D_2$ denotes that D_2 is a *valid variable matching partner* of D_1 with respect to Σ_V. The operational semantics of $\hookrightarrow_{MAP}$ is defined by the following inference rules and reflects the semantics of Σ_V.

$$\frac{D_1 \in \mathcal{V} \wedge D_2 \notin \mathcal{V} \quad l_2 \notin Z_-(l_1) \quad (Z_+(l_1) = \emptyset) \vee \left(l_2 \in Z_+(l_1)\right)}{D_1 \hookrightarrow D_2} \qquad \frac{D_1 \in \mathcal{V} \wedge D_2 \in \mathcal{V} \quad Z_-(l_1) = Z_-(l_2) \quad Z_+(l_1) = Z_+(l_2)}{D_1 \hookrightarrow D_2} \qquad \frac{D_1 \notin \mathcal{V} \wedge D_2 \in \mathcal{V} \quad D_2 \hookrightarrow D_1}{D_1 \hookrightarrow D_2}$$

Table 19. Algorithm MAP

The first rule checks that the label of the non-variable tree D_2 is not in the negative filter of the variable D_1 and that the label of D_2 is in the positive filter if defined. The second rule defines that two variables can be matching partners if their positive and negative filters are equal. The third rule is a simple swapping rule. The evaluation of the judgment $D_1 \hookrightarrow_{MAP} D_2$ terminates because the positive and negative filters are finite sets and both trees can only be swapped once by the third rule.

Note that the matching range of a variable will be checked at another level. If a variable matches multiple labeled trees, then the task of the algorithm $\hookrightarrow_{MAP}$ is to validate each of these labeled trees as a matching partner of that variable.

In the context of grammar processing, we will use variable mappings in the following *application cases*:

1) *Pattern Matching*: D_1 is a linear pattern containing variables with unique occurrences or no variables at all. D_2 does not contain any variable. There is no restriction required for the variable specification Σ_V. In summary, the conditions $\forall v \in vars(D_1). \exists! D \in \mathcal{S}(D_1). (D \approx v)$ and $vars(D_2) = \emptyset$ hold in this case.

2) *Constraint Evaluation*: D_1 and D_2 may contain variables. All variables contained in both trees have an empty positive filter $\left(Z_+^{\emptyset}\right)$, an empty negative filter $\left(Z_-^{\emptyset}\right)$ and match exactly one labeled tree $(\Lambda_\perp)$. In summary, the condition $\Sigma_V = (Z_+^{\emptyset}, Z_-^{\emptyset}, \Lambda_\perp)$ holds in this case.

We extend now the notions of semantic equality and semantic similarity to account for variables as follows.

Definition 6.1.4 (Extended Semantic Equality): Let Σ_S be a similarity specification, let $\Sigma_V = (Z_+, Z_-, \Lambda)$ be a variable specification and let D_1, D_2 be labeled trees which satisfy the conditions of an *application case*. The *extended semantic equality* of the labeled trees D_1 and D_2 with respect to Σ_S and Σ_V, denoted by $D_1 =_{(\Sigma_S, \Sigma_V)} D_2$, is a predicate over pairs of labeled trees defined as follows:

$$D_1 =_{(\Sigma_S, \Sigma_V)} D_2 \quad :\Leftrightarrow \begin{cases} D_1 \hookrightarrow_{MAP} D_2 & if\ D_1 \in \mathcal{V} \vee D_2 \in \mathcal{V} \\ (D_1 \approx D_2) \wedge \left|\mathcal{T}_{(D_1, D_2)}^{=(\Sigma_S, \Sigma_V)}\right| > 0 & if\ D_1, D_2 \notin \mathcal{V} \end{cases}$$

The difference to the standard semantic equality is that we use the notion of valid variable mappings if one of the labeled trees is a variable. Note that the set of tree matching mappings $\mathcal{T}_{(D_1, D_2)}^{=(\Sigma_S, \Sigma_V)}$ in the context $\kappa = (\Sigma_S, \Lambda)$ contains all bijective multi-functions between corresponding children of the labeled trees D_1 and D_2, respecting the subtree partitioning by the layers of the tree and their order.

If for example a variable may match one or more labeled trees because of its matching range, then the tree matching mapping may assign multiple matching partners to this variable. By the definition of $\mathcal{T}_{(D_1, D_2)}^{=(\Sigma_S, \Sigma_V)}$, each of these matching partners has to satisfy the extended semantic equality with respect to this variable. Thus, each matching partner is then validated by the algorithm $\hookrightarrow_{MAP}$.

Definition 6.1.5 (Extended Semantic Similarity): Let Σ_S be a similarity specification, let Σ_V be a variable specification and let D_1, D_2 be labeled trees which satisfy the conditions of an *application case*. The *extended semantic similarity* of the labeled trees D_1 and D_2 with respect to Σ_S, denoted by $D_1 \cong_{(\Sigma_S,\Sigma_V)} D_2$, is a predicate over pairs of labeled trees defined as follows:

$$D_1 \cong_{(\Sigma_S,\Sigma_V)} D_2 \;:\Leftrightarrow \begin{cases} D_1 \hookrightarrow_{MAP} D_2 & if\ D_1 \in \mathcal{V} \vee D_2 \in \mathcal{V} \\ (D_1 \approx D_2) \wedge \left|\mathcal{K}_{(D_1,D_2)}^{=(\Sigma_S,\Sigma_V)}\right| > 0 & if\ D_1, D_2 \notin \mathcal{V} \end{cases}$$

Analogously to semantic equality, the difference to the standard semantic similarity is that we use the notion of valid variable mapping if one of the labeled trees is a variable. Note that the set of tree key matching mappings $\mathcal{K}_{(D_1,D_2)}^{=(\Sigma_S,\Sigma_V)}$ in the context $\kappa = (\Sigma_S, \Lambda)$ contains all bijective multi-functions between corresponding key children of the labeled trees D_1 and D_2, respecting the layers of the tree and their order.

The evaluation of both recursive definitions terminates for all labeled trees because they have a finite amount of nodes and thus a finite amount of subtrees. The leaves of the evaluation tree are either comparisons between equally labeled subtrees which do not have any children in a specific layer, or the evaluations of valid variable mappings which terminate.

For the application case of constraint evaluation, it is important to show that the extended semantic similarity is a reflexive and symmetric relation.

Lemma 6.1.6 (Reflexivity and Symmetry of Extended Semantic Equality): Let $\Sigma_S = (\Sigma_O, \Sigma_K)$ be a similarity specification and let $\Sigma_V = (Z_+^{\emptyset}, Z_-^{\emptyset}, \Lambda_\perp)$ be the variable specification of the extension case for constraint evaluation. The *extended semantic equality* of labeled trees $=_{(\Sigma_S,\Sigma_V)}$ is a reflexive and symmetric relation on the set of labeled trees $\mathcal{D}$ in the extension case for constraint evaluation.

Proof: We have to show the reflexivity and symmetry of $=_{(\Sigma_S,\Sigma_V)}$.

1) *Reflexivity* : $\forall D \in \mathcal{D}.(D =_{(\Sigma_S,\Sigma_V)} D)$

   If $D \in \mathcal{V}$ with $l = L(D)$ holds, we have to show $D \hookrightarrow_{MAP} D$. Clearly, this holds because we know that $Z_+(l) = \emptyset$ and $Z_-(l) = \emptyset$ hold. If $D \notin \mathcal{V}$ holds, the proof is analogous to the proof for the reflexivity of semantic equality in *Lemma 4.1.8 (1)*.

2) *Symmetry* : $\forall D_1, D_2 \in \mathcal{D}.\Big(\big(D_1 =_{(\Sigma_S,\Sigma_V)} D_2\big) \Rightarrow \big(D_2 =_{(\Sigma_S,\Sigma_V)} D_1\big)\Big)$

If $D_1 \in \mathcal{V}$ or $D_2 \in \mathcal{V}$ hold, we have to show that $D_1 \hookrightarrow_{MAP} D_2$ implies $D_2 \hookrightarrow_{MAP} D_1$. Let $l_1 = L(D_1)$ and $l_2 = L(D_2)$ be the labels of these trees. If both trees are variables, this holds because we know that $Z_+(l_1) = Z_+(l_2) = \emptyset$ and $Z_-(l_1) = Z_-(l_2) = \emptyset$. If one tree is not a variable, this follows from the swapping rule of $\hookrightarrow_{MAP}$. If $D_1 \notin \mathcal{V}$ and $D_2 \notin \mathcal{V}$ hold, the proof is analogous to the proof for the symmetry of semantic equality in *Lemma 4.1.8 (2)*.

■

In both application cases, namely pattern matching and constraint evaluation, we are essentially constructing a *substitution* which is a finite set of replacements for variables.

Definition 6.1.7 (Substitution): A *substitution* σ is a function from variables to sequences of labeled trees, thus we have $\sigma \in \mathcal{P}(\mathcal{V} \times \mathcal{P}(\mathcal{D}))$. A substitution σ is valid if it is idempotent and if $\forall x \in dom(\sigma).\Big(\exists!\,(x, Y) \in \sigma \wedge x \notin vars\big(\sigma(x)\big)\Big)$ holds. An idempotent substitution is a substitution which is stable under self-application. We denote the identity substitution by σ_{id}.

The judgment $(D_1, \sigma) \hookrightarrow_{APPLY} D_2$ denotes that the labeled tree D_2 is the result of applying the substitution σ to the labeled tree D_1. We use also the shorter notation $D_2 \coloneqq \sigma(D_1)$ as the usual homomorphic extension. The operational semantics of $\hookrightarrow_{APPLY}$ is defined by the following inference rule.

$$\frac{\Delta \coloneqq [\delta_I(\vec{v}, Y), \delta_E(\vec{v}) | v \in \mathcal{S}(D_1) \wedge (v, Y) \in \sigma]}{(D_1, \sigma) \hookrightarrow \Delta(D_1)}$$

Table 20. Algorithm APPLY

This inference rule describes the application of a substitution in terms of constructing and applying an edit script to the target labeled tree D_1.

For applying a substitution to another substitution we need a method which does not replace the domain of a substitution. The judgment $(\sigma_1, \sigma_2) \hookrightarrow_{SUBAPPLY} \sigma_3$ denotes that the substitution σ_3 is the result of applying the substitution σ_2 to the substitution σ_1. We use the same notation $\sigma_2(\sigma_1) = \sigma_3$ since the semantics can be inferred from the context. The operational semantics of $\hookrightarrow_{SUBAPPLY}$ is defined by the following inference rule.

$$\frac{\Delta \coloneqq [\delta_I(\vec{v}, Y_2), \delta_E(\vec{v}) | (x, Y_1) \in \sigma_1 \wedge v \in \mathcal{S}(Y_1) \wedge (v, Y_2) \in \sigma_2]}{(\sigma_1, \sigma_2) \hookrightarrow \Delta(\sigma_1)}$$

Table 21. Algorithm SUBAPPLY

Although we have not explicitly defined the application of a change script to a sequence of labeled trees, this is a homomorphic extension: Build a new labeled tree without children and add the existing sequence as a specific layer to the new tree. Then use the algorithm $\hookrightarrow_{PATCH}$ as defined and collect the resulting sequence from the specific layer.

In the application case of pattern matching, we have to deal with *grounded* substitutions, which are substitutions γ with $\forall (x, Y) \in \gamma . vars(Y) = \emptyset$. For this case, we define the *subtraction* ($\ominus$) and *addition* ($\oplus$) of two grounded substitutions γ_1 and γ_2 as follows.

$$\gamma_1 \ominus \gamma_2 \coloneqq \{(v_1, Y_1) \in \gamma_1 | \forall (v_2, Y_2) \in \gamma_2 . \neg (v_1 \approx v_2)\}$$
$$\gamma_1 \oplus \gamma_2 \coloneqq (\gamma_1 \ominus \gamma_2) \cup \gamma_2$$

In the non-grounded case, we would have to apply the resulting set of substitutions on itself until it is idempotent, of course always assuming that it is free of cycles. However, in the following we only need the subtraction and addition in the grounded case.

Before we introduce the invertible grammar model, we will present the employed constraint system. In the following we will define the notion of constraint and an algorithm for the incremental evaluation of a set of constraints. The task of the constraint evaluation is to check the satisfiability of the set of constraints by constructing a unifier.

Definition 6.1.8 (Constraint): Let Σ_S be a similarity specification and let $\Sigma_V = (Z_+^{\emptyset}, Z_-^{\emptyset}, \Lambda_\perp)$ be the variable specification of the application case for constraint evaluation. A *constraint* is an ordered pair of two labeled trees (D_1, D_2) with the desired semantics $D_1 =_{(\Sigma_S, \Sigma_V)} D_2$.

Thus, the task of checking the satisfiability of a set of constraints can be interpreted as a unification problem [Robinson, 1965], that is, to find a substitution which, when applied to each constraint, makes the contained trees satisfy the extended semantic equality. Precisely, the semantics of $=_{(\Sigma_S, \Sigma_V)}$ turns the problem into a special case of C-unification which is the unification with a commutativity theory C, usually defined by a set of equations.

In our case, this commutativity theory is implicitly given by the similarity specification $\Sigma_S = (\Sigma_O, \Sigma_K)$. Thereby, the similarity order Σ_O specifies the relevance of the order of tree layers for the equality of labeled trees and thus defines an equality theory for the unification problem. The similarity key Σ_K defines the layer-specific elements that are important for the identification of corresponding labeled trees. Hence, this essentially provides the unification method a priority check before problem decomposition. A unifier is a substitution for a set of constraints, such that - when applied to the constraints - they become syntactically equal modulo the commutativity theory specified by Σ_S.

Definition 6.1.9 (Unifier): Let Σ_S be a similarity specification. The valid substitution θ is a *unifier* for the set of constraints C with respect to Σ_S if it holds that $(C, \theta) \hookrightarrow_{APPLY} C'$ and $\forall (x, y) \in C'. (x =_{\Sigma_S} y)$. The unifier θ is a *most general unifier* for the set of constraints C with respect to Σ_S if it holds that for all unifiers θ_x there exists a substitution σ such that $\theta_x = \sigma \circ \theta$. Thereby, $\sigma \circ \theta$ denotes $\sigma(\theta) \cup \sigma|K$ with $K \coloneqq dom(\sigma) \backslash dom(\theta)$. Since in general a single most general unifier may not exist, we define the *set of most general unifiers* Θ for C as follows:

1) *Correctness*
   All substitutions θ in Θ are unifiers for C.
2) *Completeness*
   For all unifiers θ_x for C there exists a substitution σ and a unifier θ in Θ such that $\theta_x = \sigma \circ \theta$ holds.
3) *Minimality*
   For all unifiers θ_1 and θ_2 both in Θ there does not exist a substitution σ such that $\theta_1 = \sigma \circ \theta_2$ holds.

Jörg Siekmann reports in his survey [Siekmann, 1989] that the unification problem is decidable for the first order case with a commutativity theory. Furthermore, there exist pairs of labeled trees which have more than one most general unifier, but they always have at most finitely many. In our parse scenario, we are only interested in the general satisfiability of a set of constraints, we do not necessarily need to construct a most general unifier.

Since we will integrate the evaluation of constraints with the translation, we ideally want to be able to evaluate the set of constraints incrementally. Therefore, the following algorithm accepts together with the working set of constraints a partial solution of the unification problem. The algorithm computes a unifier by treating a set of constraints as an ordered sequence, which simulates the deterministic behavior of Robinson's unification algorithm [Robinson, 1965].

Definition 6.1.10 (Incremental Constraint Evaluation): Let Σ_S be a similarity specification and let $\Sigma_V = (Z_+^{\emptyset}, Z_-^{\emptyset}, \Lambda_\perp)$ be the variable specification of the application case for constraint evaluation. The judgment $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ denotes the evaluation of the set of constraints C_1 with the valid substitution θ_1 under the assumption that $vars(C_1) \cap dom(\theta_1) = \emptyset$. The result is a new valid substitution θ_2. The operational semantics of $\hookrightarrow_{SAT}$ is defined by the following inference rules.

(constraint reduction)

$$\frac{\begin{array}{c} D_1 \notin \mathcal{V} \wedge D_2 \notin \mathcal{V} \\ D_1 \approx D_2 \\ \mu \in \mathcal{T}^{=(\Sigma_S,\Sigma_V)}_{(D_1,D_2)} \\ (\mu :: C_1, \theta_1) \hookrightarrow \theta_2 \end{array}}{\big((D_1, D_2) \star C_1, \theta_1\big) \hookrightarrow \theta_2}$$

(variable elimination)

$$\frac{\begin{array}{c} D_1 \in \mathcal{V} \\ D_1 \notin vars(D_2) \\ \sigma := \{(D_1, [D_2])\} \\ (\sigma(C_1), \sigma \circ \theta_1) \hookrightarrow \theta_2 \end{array}}{\big((D_1, D_2) \star C_1, \theta_1\big) \hookrightarrow \theta_2}$$

(identity elimination)

$$\frac{\begin{array}{c} D_1 \in \mathcal{V} \wedge D_2 \in \mathcal{V} \\ D_1 \approx D_2 \\ (C_1, \theta_1) \hookrightarrow \theta_2 \end{array}}{\big((D_1, D_2) \star C_1, \theta_1\big) \hookrightarrow \theta_2}$$

(constraint swapping)

$$\frac{\begin{array}{c} D_1 \notin \mathcal{V} \wedge D_2 \in \mathcal{V} \\ \big((D_2, D_1) \star C_1, \theta_1\big) \hookrightarrow \theta_2 \end{array}}{\big((D_1, D_2) \star C_1, \theta_1\big) \hookrightarrow \theta_2}$$

(unifier found)

$$\frac{}{(\emptyset, \theta) \hookrightarrow \theta}$$

Table 22. Algorithm SAT
(adapted from [Martelli & Montanari, 1982])

This algorithm is an adaptation of Martelli & Montanari's unification algorithm [Martelli & Montanari, 1982] to the semantics of $=_{(\Sigma_S,\Sigma_V)}$. We will give a short description of the five rules by referring to them as being numbered from top to bottom and left to right. The first rule replaces a constraint by a set of decomposed constraints obtained from a selected tree matching mapping $\mu \in \mathcal{T}^{=(\Sigma_S,\Sigma_V)}_{(D_1,D_2)}$. Since the tree matching mapping μ is a set of pairs of labeled trees, we can interpret μ as a set of constraints. In the ordered case, we have always at most one matching mapping. If a tree layer is unordered, this may be a choice point for problem decomposition. The second rule transfers a constraint to the substitution after a negative occurs check and by applying it to the remaining constraints and to the current substitution. The third rule removes an identity constraint. The fourth rule swaps a constraint such that it is headed by a variable. Finally, the fifth rule is the only concluding rule.

Note that we totally order the tree matching mappings by the position of the matching partners from left to right. Hence, the alternatives at the choice point are ordered. Since furthermore only one inference of this algorithm is applicable in a specific evaluation state, we can conclude that the algorithm $\hookrightarrow_{SAT}$ and its backtracking mechanism are deterministic.

The termination of the algorithm for constraint evaluation can be shown as follows by analyzing the size of the set of constraints in every evaluation step.

Lemma 6.1.11 (Termination of Constraint Evaluation): Let Σ_S be a similarity specification and let $\Sigma_V = (Z_+^{\emptyset}, Z_-^{\emptyset}, \Lambda_\perp)$ be the variable specification of the application case for constraint evaluation. Then the algorithm $\hookrightarrow_{SAT}$ terminates.

Proof: The application of the fifth rule terminates clearly. Furthermore, the fifth rule is only applicable once because it marks the end of the algorithm $\hookrightarrow_{SAT}$. The fourth rule can only be applied at most once to any constraint in the set of constraints. Since the second rule reduces the amount of different variables in the set of constraints, since the other rules do not introduce new variables, and since the set of constraints contains only finitely many different variables, the second rule can only be applied finitely many times.

Thus, if we can prove that the remaining two rules reduce the set of constraints, the algorithm $\hookrightarrow_{SAT}$ terminates because the fifth rule is applicable when the set of constraints is empty. We can prove this by showing that the size of the set of constraints is reduced by rule one and three. Thereby, the size of a constraint is the sum of the sizes of both labeled trees, and the size of the set of constraints is the sum of the sizes of all contained constraints. Since rule three removes a constraint, this is trivially clear. Rule one replaces a constraint by a set of constraints which contains pairs of direct subtrees as constraints. The size of this set is always smaller than the size of the original constraint because the root nodes are missing. Additionally, the choice point in rule one has only finitely many options of tree matching mappings. It follows that $\hookrightarrow_{SAT}$ terminates, either with returning a new substitution, or because all backtracking options are exhausted and no inference rule is applicable.

■

Since the purpose of the algorithm $\hookrightarrow_{SAT}$ is to check the satisfiability of a set of constraints, it is not important to find a most general unifier. Let x, y be variables and f commutative, then $\hookrightarrow_{SAT}$ computes indeed for $(C_1, \emptyset)$ with $C_1 = \{(f[\boldsymbol{x}, 1], f[1, \boldsymbol{y}])\}$ the resulting unifier $\theta_2 = \{(\boldsymbol{x}, 1), (\boldsymbol{y}, 1)\}$, although $\theta_3 = \{(\boldsymbol{x}, \boldsymbol{y})\}$ is a more general unifier.

In the following, we will prove that the incremental constraint evaluation method $\hookrightarrow_{SAT}$ is both sound and complete.

Lemma 6.1.12 (Soundness of Incremental Constraint Evaluation): Let Σ_S be a similarity specification. Let C_1 be a set of constraints and let θ_1 be a valid substitution with $vars(C_1) \cap dom(\theta_1) = \emptyset$. Furthermore, let $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ hold. Then θ_2 is a unifier for the set of constraints C_1 and θ_2 is a consistent extension of θ_1, that is, there exists a substitution σ with $\theta_2 = \sigma \circ \theta_1$.

Proof: We show this by proving the soundness of every inference rule of the algorithm $\hookrightarrow_{SAT}$. The statement then follows because the derivation tree for $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ is a sequence of inference applications.

1) *Case Rule 1*: By the assumptions of this rule, we know that θ_2 is a unifier for $C_1 \cup \mu$ with $\mu \in \mathcal{T}^{=_{(\Sigma_S,\Sigma_V)}}_{(D_1,D_2)}$ and θ_2 is a consistent extension of θ_1. Let $D_1' \coloneqq \theta_2(D_1)$, $D_2' \coloneqq \theta_2(D_2)$, and $\mu' \coloneqq \theta_2(\mu)$, then we know that $\mu' \in \mathcal{T}^{=_{\Sigma_S}}_{(D_1',D_2')}$. Because of $D_1 \approx D_2$ it follows that $D_1' \approx D_2'$ and hence $D_1' =_{\Sigma_S} D_2'$. Therefore, it follows that θ_2 is a unifier for $\{(D_1, D_2)\}$, hence also for $\{(D_1, D_2)\} \cup C_1$. This shows the soundness of the first rule.

2) *Case Rule 2*: We know that θ_1 is a valid substitution, then $\sigma \circ \theta_1$ is valid too with $\sigma \coloneqq \{(D_1, [D_2])\}$ because $D_1 \notin vars(D_2)$ and $vars(D_1) \cap dom(\theta_1) = \emptyset$. Furthermore, by construction it follows that $\sigma \circ \theta_1$ is a consistent extension of θ_1. Hence, it holds that $vars(\sigma(C_1)) \cap dom(\sigma \circ \theta_1) = \emptyset$. By the assumption of this inference rule, we thus know that θ_2 is a unifier for $\sigma(C_1)$. It remains to be shown that θ_2 is a unifier for $\{(D_1, D_2)\} \cup C_1$. Since θ_2 is a consistent extension of $\sigma \circ \theta_1$, it follows that there exists a substitution σ' such that $\theta_2 \coloneqq \sigma' \circ \sigma \circ \theta_1$ holds. From $vars(D_1) \cap dom(\theta_1) = \emptyset$ and the definition of σ we can derive:

$$\theta_2(D_1) = \sigma' \circ \sigma \circ \theta_1(D_1) = \sigma' \circ \sigma(D_1) = \sigma'(D_2)$$
$$\theta_2(D_2) = \sigma' \circ \sigma \circ \theta_1(D_2) = \sigma' \circ \sigma(D_2) = \sigma'(D_2)$$

Therefore, it follows that θ_2 is a unifier for $\{(D_1, D_2)\}$.
From $vars(C_1) \cap dom(\theta_1) = \emptyset$ we can derive furthermore:

$$\theta_2(\sigma(C_1)) = \sigma' \circ \sigma \circ \theta_1(\sigma(C_1)) = \sigma' \circ \sigma(\sigma(C_1)) = \sigma' \circ \sigma(C_1)$$
$$\theta_2(C_1) = \sigma' \circ \sigma \circ \theta_1(C_1) = \sigma' \circ \sigma(C_1)$$

Hence, because θ_2 is a unifier for $\sigma(C_1)$ it is also a unifier for C_1 and thus for $\{(D_1, D_2)\} \cup C_1$. This shows the soundness of the second rule.

3) *Case Rule 3*: Then we consider the case of the third rule. By the assumptions of this inference rule, we know that θ_2 is a unifier for C_1 and θ_2 is a consistent extension of θ_1. Because of $D_1, D_2 \in \mathcal{V}$ and $D_1 \approx D_2$ it follows immediately that $D_1 =_{(\Sigma_S,\Sigma_V)} D_2$ because $=_{(\Sigma_S,\Sigma_V)}$ is reflexive by *Lemma 6.1.6*. Therefore, it follows that θ_2 is a unifier for $\{(D_1, D_2)\} \cup C_1$. This shows the soundness of the third rule.

4) *Case Rule 4*: We continue with the fourth rule. By the assumptions of this inference rule, we know that θ_2 is a unifier for $\{(D_2, D_1)\} \cup C_1$ and θ_2 is a consistent extension of θ_1. Since $=_{(\Sigma_S,\Sigma_V)}$ is symmetric by *Lemma 6.1.6*, it follows that θ_2 is a unifier for $\{(D_1, D_2)\} \cup C_1$, too. This shows the soundness of the fourth rule.

5) *Case Rule 5*: Finally, we have the case of the fifth rule. Clearly, any substitution θ is a unifier for the empty set of constraints $\emptyset$. This shows the soundness of the fifth rule.

■

It remains to be shown that if there exists a unifier θ_3 for C_1 which is a consistent extension of θ_1, then the judgment $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ can be derived.

Lemma 6.1.13 (Completeness of Incremental Constraint Evaluation): Let Σ_S be a similarity specification. Let C_1 be a set of constraints and let θ_1 be a valid substitution with $vars(C_1) \cap dom(\theta_1) = \emptyset$. If there exists a unifier θ_3 for C_1 which is a consistent extension of θ_1, then there exists a valid substitution θ_2 such that $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ holds.

Proof: We will prove this statement by contradiction. We assume that there exists a unifier θ_3 for C_1 which is a consistent extension of θ_1, but we cannot derive the judgment $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ for any valid substitution θ_2.

Since the fifth rule is not applicable, the set of constraints contains at least one constraint (D_1, D_2).

1) *Case* $D_1 \notin \mathcal{V} \wedge D_2 \notin \mathcal{V}$: If $\neg(D_1 \approx D_2)$ were the case, there cannot exist a unifier, which contradicts the assumption. Thus $D_1 \approx D_2$ must hold. In case of $\mathcal{T}^{=(\Sigma_S,\Sigma_V)}_{(D_1,D_2)} = \emptyset$, there cannot exist a unifier. Thus there must exist a $\mu \in \mathcal{T}^{=(\Sigma_S,\Sigma_V)}_{(D_1,D_2)}$. The first rule is applicable and can be used to decompose all such tree matching mappings.

2) *Case* $D_1 \notin \mathcal{V} \wedge D_2 \in \mathcal{V}$: The fourth rule would be applicable.

3) *Case* $D_1 \in \mathcal{V} \wedge D_2 \notin \mathcal{V}$: If $D_1 \in vars(D_2)$ were the case, there could not exist a unifier. Thus $D_1 \notin vars(D_2)$ must hold and the second rule would be applicable.

4) *Case* $D_1 \in \mathcal{V} \wedge D_2 \in \mathcal{V}$: If $D_1 \approx D_2$ were the case, the third rule would be applicable. Thus $\neg(D_1 \approx D_2)$ must hold, hence we have $D_1 \notin vars(D_2)$ and the second rule would be applicable.

We conclude that, as long as the set of constraints contains a constraint (D_1, D_2), at least one inference rule is applicable. If the algorithm $\hookrightarrow_{SAT}$ does not directly select a correct tree matching mapping $\mu \in \mathcal{T}^{=(\Sigma_S,\Sigma_V)}_{(D_1,D_2)}$ at the choice point in the application of the first rule, we know that it will try all other alternatives by backtracking. Since there exists a unifier θ_3, this implies that $\theta_3(D_1) =_{\Sigma_S} \theta_3(D_2)$ holds and that there exists a tree matching mapping $\mu' \in \mathcal{T}^{=\Sigma_S}_{(\theta_3(D_1),\theta_3(D_2))}$. Hence there must also exist at least one correct tree matching mapping $\mu \in \mathcal{T}^{=(\Sigma_S,\Sigma_V)}_{(D_1,D_2)}$ with $\mu' = \theta_3(\mu)$. Since the algorithm $\hookrightarrow_{SAT}$ terminates by *Lemma 6.1.11*, we can conclude that the set of constraints will be reduced until the fifth rule is applicable. Thus we have constructed a derivation tree for $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ which contradicts our assumption.

■

We have shown that, if a unifier θ_3 exists for C_1 which is a consistent extension of θ_1, the judgment $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ can always be derived for some θ_2. Furthermore, θ_2 is a unifier for the set of constraints C_1 and θ_2 is a consistent extension of θ_1. Altogether, we have shown the soundness and completeness of the presented incremental constraint evaluation method.

Note that the algorithm $\hookrightarrow_{SAT}$ computes a unifier θ_2 for C_1 that is a consistent extension of θ_1. Thus, if we first derive the judgment $(C_1, \theta_1) \hookrightarrow_{SAT} \theta_2$ and if we then extend the set of constraints by a new set of constraints C_2, trying to derive $(C_2, \theta_2) \hookrightarrow_{SAT} \theta_3$, this may not succeed although $C_1 \cup C_2$ is satisfiable. The reason is that the decisions taken at the choice points of the construction for θ_2 might not be the correct decisions for $(C_1 \cup C_2, \theta_1) \hookrightarrow_{SAT} \theta_3$. For the practical application, this implies that the similarity order Σ_O should be used very carefully for the constraint system, because it possibly leads to backtracking.

Before we present in the following the formal definition of an invertible grammar, we would like to give an intuitive introduction by discussing examples of invertible rules.

FORALL-MULTI	
L	TERM
P_{IN}	{ dir="r" }["\forall" **c** "," **n** ":" **t** "." **f**]
M	[(k,[**c** ":" **t**],[**v**],VARINTRO), (m,["\forall" **n** ":" **t** "." **f**],[**w**],TERM)]
P_{OUT}	[\F{"forall"}[\B[**v**] , **w**]]
C_{PRE}	{ m.type = "bool", ME.type = "bool" }
C_{POST}	{ }
i	{ false }

This example shows a grammar rule called FORALL-MULTI which is part of a set of grammar rules that transform a syntactic linear-style mathematical formula into a semantic tree-style representation. The grammar rule is serialised as a labeled tree, where the layers of the tree are the rows in the given example, except from the first row L which contains the label of the root node of the grammar rule, which is also the category of that grammar rule. The name FORALL-MULTI has no technical significance. We will use this name to refer to a specific rule in the textual description.

The second row P_{IN} contains the input pattern, against which the input will be matched, and a matching direction. The matching direction of a pattern is by default to the right, otherwise the direction can be specified as shown for the input pattern in the example rule. Thereby, dir="r" denotes the matching direction to the right, and dir="l" denotes the matching direction to the left. Since there exists often more than one matching mapping because some variables might match more than one partner, we order the matching mappings by the amount of matching partners for variables from high to low, and from left to right (dir="l") or from right to left (dir="r"). The matching direction indicates in which order the matching mappings should be selected.

The third row M contains the invocations of other rules. This example demonstrates how the input pattern can be recombined for two recursive rule invocations. The first invocation with the label *k* processes the first quantified typed variable, whereas the second invocation *m* transforms the remaining linear-style formula. The second parameter of the invocation is the input pattern for this rule invocation, whose variables will be substituted by the matching mapping of the input pattern of the current rule. The output of the rule invocation will be matched against the pattern in the third parameter of the invocation. The fourth parameter is the category of the rule that can be invoked.

The fourth row P_{OUT} contains the output pattern which will be completed by the matching mappings of the output patterns of the recursive rule invocations.

The fifth row $\boldsymbol{C_{PRE}}$ and the sixth row $\boldsymbol{C_{POST}}$ contain constraints which are written as equations. Constraint-variables are written in attribute-style notation by separating the variable and its attribute with a dot. The special variable *ME* refers to the invocation of the current grammar rule. This is in line with the idea of attaching attributes to the invocation nodes in the parse tree. We will illustrate this concept after the introduction of a second grammar rule. The difference between both rows is that the constraints in C_{PRE} are checked for satisfiability before the recursive rule invocations and the constraints in C_{POST} after them.

The last row $\boldsymbol{i}$ indicates whether the environment of the invoked rules is inherited (⊤) or not (⊥). The environment is by default inherited. The concept of an environment is part of our solution for static scoping and will be explained in the following.

As a second example, let us consider the grammar rule `TYPED-VAR` which is of the category `VARINTRO` and potentially used by the invocation *k* in the previous rule.

TYPED-VAR	
$\boldsymbol{L}$	`VARINTRO`
$\boldsymbol{P_{IN}}$	`[ c ":" t ]`
$\boldsymbol{M}$	`[ (k,[c],[v],IDI),` `(m,[t],[w],TYPE) ]`
$\boldsymbol{P_{OUT}}$	`[ \V{ v }[ w ] ]`
$\boldsymbol{C_{PRE}}$	`{ }`
$\boldsymbol{C_{POST}}$	`{ ME.type = m.type , (k.index).type = m.type }`
$\boldsymbol{i}$	`{ true }`

The first subrule *k* in this second example calls the special identity transformation rule class `IDI`, which passes input to output and returns the semantic hash value *#1* of the input tree as the value of the attribute *index*. This index value *k.index* is in turn used to assign a type to this particular variable with the meta-variable *(k.index).type*. Meta-variables are evaluated by replacing the constraint-variable with the label of its value. In this case, *(k.index).type* is evaluated to *#1.type*.

The semantic hash function returns the same hash value for semantically equal input trees. Since we will use this feature in particular for assigning a unique identifier to variables in mathematical formulas, we need to take special care about scoping. The role of the environment is to map the hash value to the correct identifier in the current scope. A local scope is always bounded by a single grammar rule which does not inherit the environments of its recursive rule invocations. In this case, the environment at the beginning of the rule processing is returned again.

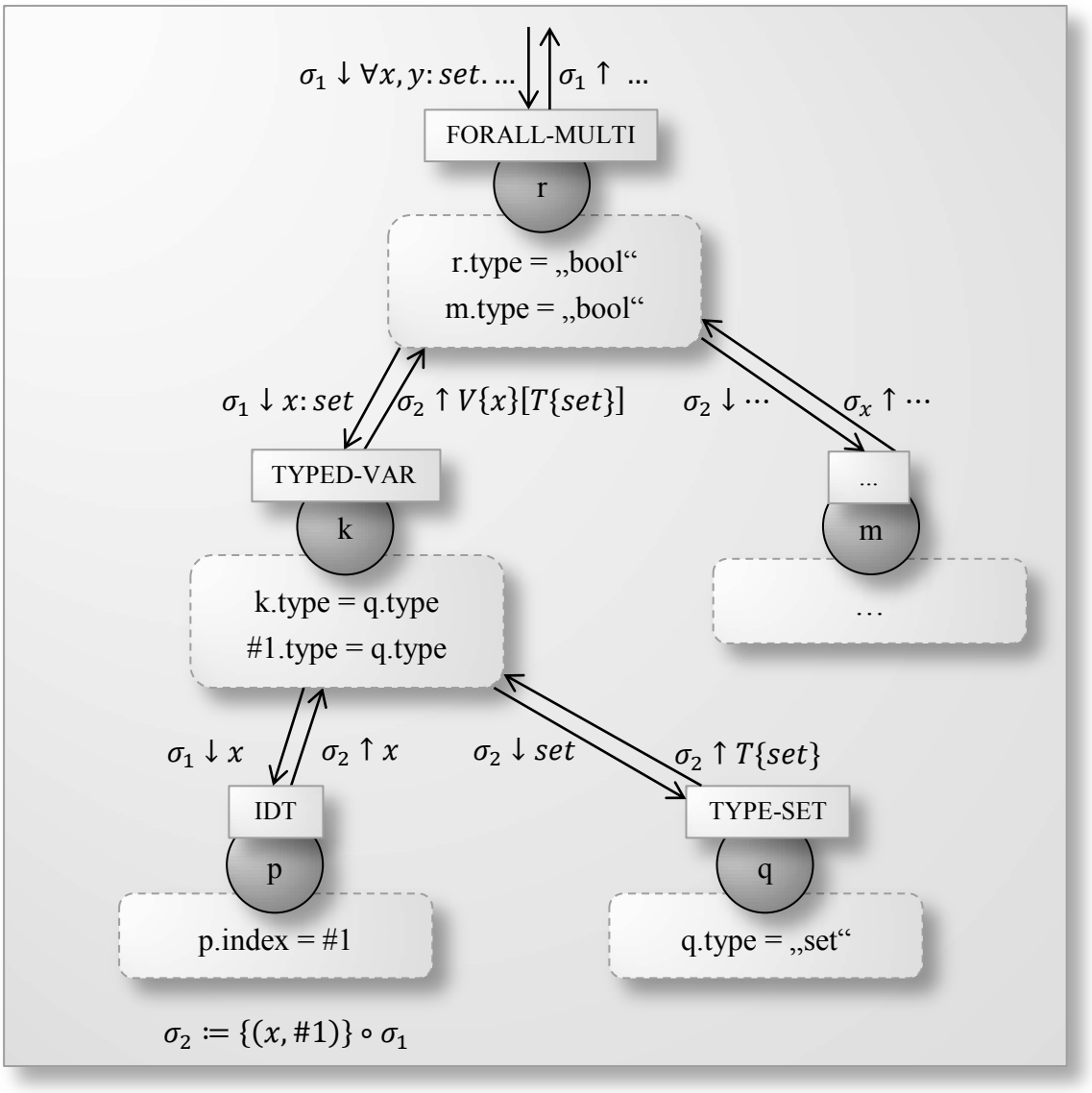


Figure 10. Example Parse Tree

A (partial) parse tree for the mathematical formula $\forall x, y: set. \ldots$ is shown in Figure 10. The rule invocations are represented by nodes containing the invocation label. The node with the label *r* is the root of this example parse tree. The box label above a node, for example `FORALL-MULTI`, denotes the rule that is invoked at that node. Note that the invocation labels defined in the rule `TYPED-VAR` have changed from *k* and *m* to *p* and *q*. The reason is that every invocation label in the parse tree has to be unique. Therefore, a grammar rule needs to be instantiated first with fresh invocation labels. Thereby, the label *ME* is replaced by the current invocation label of that grammar rule in the parse tree. The box below a node contains the constraints which are added by that rule invocation. On the left hand side of a downward edge, we have written the input environment and input sequence of a rule invocation. On the right hand side of an upward edge, the ouput environment and the output sequence is shown.

Our invertible grammar model will be extensively based on the concept of pattern matching. Therefore we introduce now the notion of a pattern.

Definition 6.1.14 (Pattern): A *pattern* is a sequence of labeled trees with uniquely occurring variables. A pattern has a direction which is either left or right. Let $\mathcal{L}_T$ be a set of text labels, the *lexicon*. Then the set of all patterns with respect to $\mathcal{L}_T$ is denoted by $\mathcal{Q}(\mathcal{L}_T)$ and contains all pattern whose text subtrees have a label in $\mathcal{L}_T$.

We totally order the matching mappings between a pattern and a variable-free sequence of labeled trees by the amount of labeled trees matched with the pattern variables. Thereby, we order the pattern variables by the traversal order of depth-first search. If the pattern direction is left, the mappings are ordered from many to few matching partners of the variables from left to right. If the pattern direction is right, this order is inverted. The matching mappings are then additionally ordered by the position of the matching partners. The described ordering of matching mappings allows for a deterministic pattern matching. Furthermore, the pattern direction helps defining whether a rule is associative to the left or to the right. We introduce now the new invertible grammar model as a combination of the grammar models *AG* and *TGL* as follows.

Definition 6.1.15 (Invertible Grammar): Let $\mathcal{L}_T$ be a lexicon, let $\mathcal{L}_R$ be the set of rule labels and let $\mathcal{L}_M$ be the set of subrule labels with $\mathcal{L}_R \cap \mathcal{L}_M = \emptyset$. A *rule* r of the set of rules $\mathcal{R}$ consists of

- a rule label $l_r \in \mathcal{L}_R$,
- an input pattern $P_{in} \in \mathcal{Q}(\mathcal{L}_T)$,
- a sequence of invocations $M \subset \mathcal{M}$,
- an output pattern $P_{out} \in \mathcal{Q}(\mathcal{L}_T)$,
- a set of precondition constraints $C_{pre} \in \mathcal{P}(\mathcal{D} \times \mathcal{D})$,
- a set of postcondition constraints $C_{post} \in \mathcal{P}(\mathcal{D} \times \mathcal{D})$,
- and an environment inheritance flag $i \in \{\top, \bot\}$.

We denote a rule by $r = (l_r, P_{in}, M, P_{out}, C_{pre}, C_{post}, i)$ and model it as a labeled tree where l_r is the root label and the other components form a layer of the tree. Thus, we have $L(r) = l_r$. An *invocation* m of the set of rule invocations $\mathcal{M}$ consists of

- an invocation label $l_m \in \mathcal{L}_M$,
- an input pattern $P_i \in \mathcal{Q}(\mathcal{L}_T)$,
- an output pattern $P_o \in \mathcal{Q}(\mathcal{L}_T)$,
- and a rule label $l_r \in \mathcal{L}_R$.

We denote a subrule by $m = (l_m, P_i, P_o, l_r)$ and model it as a labeled tree with root label l_m and the other components forming a layer of the tree. Thus, we have $L(m) = l_m$.

Finally, an *invertible grammar* $I = (\mathcal{L}_T, R, l_0, \Sigma_S^I, \Sigma_S^O, \Sigma_S^C, \Sigma_V)$ contains a lexicon $\mathcal{L}_T$, a sequence of rules R with a designated start rule label $l_0 \in \mathcal{L}_R$, a similarity specification Σ_S^I for the input, a similarity specification Σ_S^O for the output, a similarity specification Σ_S^C for the constraints and a variable specification Σ_V. Altogether, an invertible grammar is represented as a labeled tree.

A grammar rule will be interpreted by first evaluating the set of precondition constraints. Then we compute a matching mapping between the input pattern of the rule and the current input sequence. Note that the matching mapping is computed with respect to the input similarity specification, which may account for example for ordering variants of terminals in the input pattern. After that, the matching mapping is used to recursively interpret the rule invocations of the grammar rule. Thereby, the input of a rule invocation can be defined by arbitrary input patterns which can access the variable matching mapping partners of the variables in the input pattern of the grammar rule. The resulting output sequence of a rule invocation is in turn matched with the output pattern. The variable matching mapping partners of the variables contained in all invocation output patterns are used to compute the output of the grammar rule from its output pattern. Finally, the set of postcondition constraints is evaluated. The formal semantics of the invertible grammar will be defined in the next section with the incremental interpreter.

We need to define additional conditions that have to be satisfied by an invertible grammar in order to be a valid grammar for the invertible transformation of labeled trees.

Definition 6.1.16 (Valid Invertible Grammar): An *invertible grammar* $I = (\mathcal{L}_T, R, l_0, \Sigma_S^I, \Sigma_S^O, \Sigma_S^C, \Sigma_V)$ is valid if the following conditions hold. For all rules $r \in R$ with $r = (l_r, P_{in}, M, P_{out}, C_{pre}, C_{post}, i)$ it holds that

1) *Disjoint variables for patterns and constraints*
   a) $vars(P_{in}) \cap vars(P_{out}) = \emptyset$
   b) $\big(vars(P_{in}) \cup vars(P_{out})\big) \cap \big(vars\big(C_{pre}\big) \cup vars(C_{post})\big) = \emptyset$
2) *Unique variable occurrences*
   a) $\forall v \in vars(P_{in}). \exists! D \in \mathcal{S}(P_{in}). (D \approx v)$
   b) $\forall v \in vars(P_{out}). \exists! D \in \mathcal{S}(P_{out}). (D \approx v)$
3) *Unique rule invocation labels*

   $\forall m_1 \in M. \forall m_2 \in M. \Big((m_1 \neq m_2) \Rightarrow \big(L(m_1) \neq L(m_2)\big)\Big)$

Furthermore, for all rule invocations $m \in M$ with $m = (l_m, P_i, P_o, l_r)$ it holds that

1) *Subset of pattern variables*
   a) $vars(P_i) \subseteq vars(P_{in})$
   b) $vars(P_o) \subseteq vars(P_{out})$
2) *Unique variable occurrences*
   a) $\forall v \in vars(P_i). \exists! D \in \mathcal{S}(P_i). (D \approx v)$
   b) $\forall v \in vars(P_o). \exists! D \in \mathcal{S}(P_o). (D \approx v)$
3) *Smaller pattern size*
   a) $|\mathcal{S}(P_{in})| > |\mathcal{S}(P_i)|$
   b) $|\mathcal{S}(P_{out})| > |\mathcal{S}(P_o)|$

The major changes of the presented invertible grammar formalism with respect to the grammar models *AG* and *TGL* can be summarized as follows:

1) the integration of the similarity specification with the rule pattern matching,
2) the input and output recombination for the recursive rule processing,
3) the incremental unification-based constraint system, and
4) the automated inversion of the grammar.

Let us now take a closer look at the constraints and the support for static scoping. Our solution to this problem is based on the following three parts:

1) a semantic hash indexing function which will be used by identity rules to pass a hash value of the input tree to the constraints,
2) meta-variables in the constraints which will be evaluated to constraint-variables,
3) an environment which is maintained along the parse tree, which can be inherited or not from recursive rule invocations, and which is used to locally map the result of the semantic hash indexing function to different values.

We begin by introducing the semantic hash indexing function.

Definition 6.1.17 (Semantic Hash Indexing Function): Let Σ_S be a similarity specification. A *semantic hash indexing function* $\mathcal{H}$ assigns to every labeled tree D a *hash value* $D_x \in \mathcal{V}_I$, denoted by $\mathcal{H}(D) = D_x$. Let $\mathcal{L}_I$ be the set of hash value labels, then we define the set of all *hash-variables* by $\mathcal{V}_I \coloneqq \{D \in \mathcal{V} | L(D) \in \mathcal{L}_I\}$. The semantic hash indexing function satisfies the following property with respect to Σ_S:

$$\forall D_1, D_2. \Big(\big(D_1 =_{\Sigma_S} D_2 \big) \Leftrightarrow \big(\mathcal{H}(D_1) \approx \mathcal{H}(D_2) \big) \Big)$$

For our purpose of semantic hash indexing, we use the following simple implementation of maximal structure sharing with respect to the similarity specification Σ_S:

1) If the tree has no children or only hash values as children, test for semantic equality with indexed labeled trees. If there is a tree found, return its hash value. If there is no tree found, store the tree with a new hash value and return this hash value. In all other cases, proceed with step (2).
2) Replace all leaves D_i by the corresponding hash value $\mathcal{H}(D_i) = D_x$. Replace all subtrees D_k except of the tree itself, which have only hash values as children, by the corresponding hash value $\mathcal{H}(D_k) = D_y$. Proceed with step (1).

Since $=_{\Sigma_S}$ is an equivalence relation by *Lemma 4.1.8*, it is sufficient to test for semantic equality with one labeled tree in the equivalence classes of labeled trees defined by $=_{\Sigma_S}$. Thus, the required property of the semantic hash indexing function follows immediately from the construction steps by induction over the structure of labeled trees.

Next, we will take a closer look at the constraint-variables and meta-variables which can be used in the constraints of a rule.

Definition 6.1.18 (Constraint- and Meta-Variables): Let $\mathcal{L}_S$ be a set of custom labels. A *constraint-variable* $v \in \mathcal{V}$ is a variable with a label of the set of constraint labels $\mathcal{L}_C$. A constraint label $L(v) = (l_n, l_a)$ is a pair consisting of a sublabel $l_n \in \mathcal{L}_M \uplus \{l_{ME}\} \uplus \mathcal{L}_I \uplus \mathcal{L}_S$, and an attribute label $l_a \in \mathcal{L}_A$. The set of all constraint-variables is defined by $\mathcal{V}_C := \{D \in \mathcal{V} | L(D) \in \mathcal{L}_C\}$. A *meta-variable* $v \in \mathcal{V}$ is a variable with a label of the set of meta labels $\mathcal{L}_E$. A meta label $L(v) = (l_c, l_a)$ is a pair consisting of a constraint label $l_c \in \mathcal{L}_C$, and an attribute label $l_a \in \mathcal{L}_A$. The set of all meta-variables is defined by $\mathcal{V}_E := \{D \in \mathcal{V} | L(D) \in \mathcal{L}_E\}$. Note that the sets of labels $\mathcal{L}_I$, $\mathcal{L}_S$, $\mathcal{L}_C$ and $\mathcal{L}_E$ have to be pairwise disjoint.

In summary, we have four types of variables which can be used in the constraints: (1) *constraint-variables* like *k.index*, (2) *meta-variables* like *(k.index).type*, (3) *hash-variables* like *#1.type*, and (4) *custom-variables* like *A.type*.

In order to set the constraints of rules in a parse tree into relation, we need to instantiate a rule by replacing the rule invocation labels consistently with fresh labels. Assume we have an infinite amount of fresh labels $\mathcal{L}_M^{FRESH} \subseteq \mathcal{L}_M$ where fresh means unused in a specific parse tree of a document translation. Then we define an instance $r(l_x)$ with $l_x \in \mathcal{L}_M$ of a rule $r = (l, P_{in}, M, P_{out}, C_{pre}, C_{post}, i)$ by $r(l_x) = (l, P_{in}, M', P_{out}, C'_{pre}, C'_{post}, i)$ where all subrule labels in M', C'_{pre}, C'_{post} are consistently replaced by fresh subrule labels. The special label l_{ME} is thereby replaced by the label l_x, which establishes a connection to the parent rule in the parse tree.

Then we proceed by introducing the method for the evaluation of a meta-variable.

Definition 6.1.19 (Meta-Variable Evaluation): Let D_1 be a labeled tree and let θ be a substitution. The judgment $(D_1, \theta) \hookrightarrow_{META} D_2$ denotes that the evaluation of meta-variables in D_1 by θ results in the labeled tree D_2. The operational semantics of $\hookrightarrow_{META}$ is defined by the following inference rule. Thereby, we use the following additional notation. Let $v \in \mathcal{V}_E$ be a meta-variable with $L(v) = (l_c, l_a)$. The corresponding constraint-variable is $\tilde{v} \in \mathcal{V}_C$ with $L(\tilde{v}) = l_c$. If $(\tilde{v}, [D]) \in \theta$ and $l_x = L(D)$ hold, we define the evaluated meta-variable by $v(\theta)$ with $v(\theta) \in \mathcal{V}_C$ and $L\big(v(\theta)\big) = (l_x, l_2)$.

$$\frac{\Delta \coloneqq \left[\delta_R\big(\vec{v}, v(\theta)\big) \middle| v \in \mathcal{S}(D_1) \wedge v \in \mathcal{V}_E\right] \quad (D_1, \Delta) \hookrightarrow_{PATCH} D_2 \quad vars(D_2) \cap \mathcal{V}_E = \emptyset}{(D_1, \theta) \hookrightarrow D_2}$$

Table 23. Algorithm META

Let *(k.index).type* be a meta-variable and let the substitution θ assign the hash value *#1* to the constraint-variable *k.index*. Then the meta-variable *(k.index).type* is evaluated to the hash-variable *#1.type*.

Our motivation for augmenting the semantic hash indexing function by an environment is to provide means to locally assign different hash values to semantically equal input trees. The reason is that we want to support for example the introduction of a variable inside the scope of an equally named variable. The new introduced variable should then be mapped to a fresh hash value as well as all variable occurrences in its scope.

Definition 6.1.20 (Environment): An environment σ is a function from the set of hash-variables $\mathcal{V}_I$ to $\mathcal{V}_I$. The identity environment is denoted by σ_{id}.

6.2 Incremental Interpreter

We introduced the invertible grammar together with an intuitive description of its semantics. In the following, the formal semantics will be defined with the introduction of the incremental grammar interpreter. With an incremental processing model we want to investigate how the transformation process can benefit from the information collected during the transformation of a previous version of an interface document. The primary reason is not efficiency but the possibility to reuse cached information in the inverse direction in order to generate document parts that are lost in the transformation process. The following list is a non-exhaustive enumeration of general techniques for incremental processing:

- Compiling incremental programs [Yellin & Strom, 1988]
- Exploiting the transformation result [Liu & Teitelbaum, 1995b]
- Discovering auxiliary information [Liu *et al*, 1996]
- Caching intermediate results [Liu & Teitelbaum, 1995a]

The first technique can be used to compile a program written in a special language automatically into an incremental program. Transferred to our problem setting, this technique would be interesting if we would compile a parser from an invertible grammar. Since we expect frequent changes to a grammar, for example because of the definition of new notation, a direct interpretation of the grammar is a more suitable processing model.

The second technique transforms a program into an incremental version with respect to possible input changes. The basic idea is to reuse as much subtransformations as possible by caching transformation results. This technique is applicable to our grammar interpreter, in particular in cases where the set of constraints of subproblems are independent from the parent set of constraints in the parse tree. However, since the conditions for the reuse of results needs to be checked, a gain of efficiency cannot be guaranteed in all cases.

The third technique computes embedding relations and analyses forward dependencies in the spirit of a binding-time analysis for partial evaluation. Thus, candidate auxiliary information is collected which is used to generate an incremental version of a program. This technique would be useful if we allowed in the constraints the same expressive power as attribute grammars.

The last technique extends a program in such a way that all intermediate results are returned which are useful for the incremental computation. We will adapt this technique to our setting by designing the grammar interpreter in such a way that it returns a sequence of *transformation traces* of the grammar rules and matching mappings used to construct the parse tree. The incremental interpreter then exploits these intermediate results.

In order to be able to check the conditions for the reuse of intermediate results, we need means to identify the preserved subtrees between two versions of the same interface document. Figure 11 shows such a maximal tree alignment between two documents. In this example, the order of the layers of the tree is not relevant. We call this maximal tree alignment an *incremental matching mapping*. Note that a subtree is called preserved even if some descendant trees are not *preserved*, for example A is preserved while F is not.

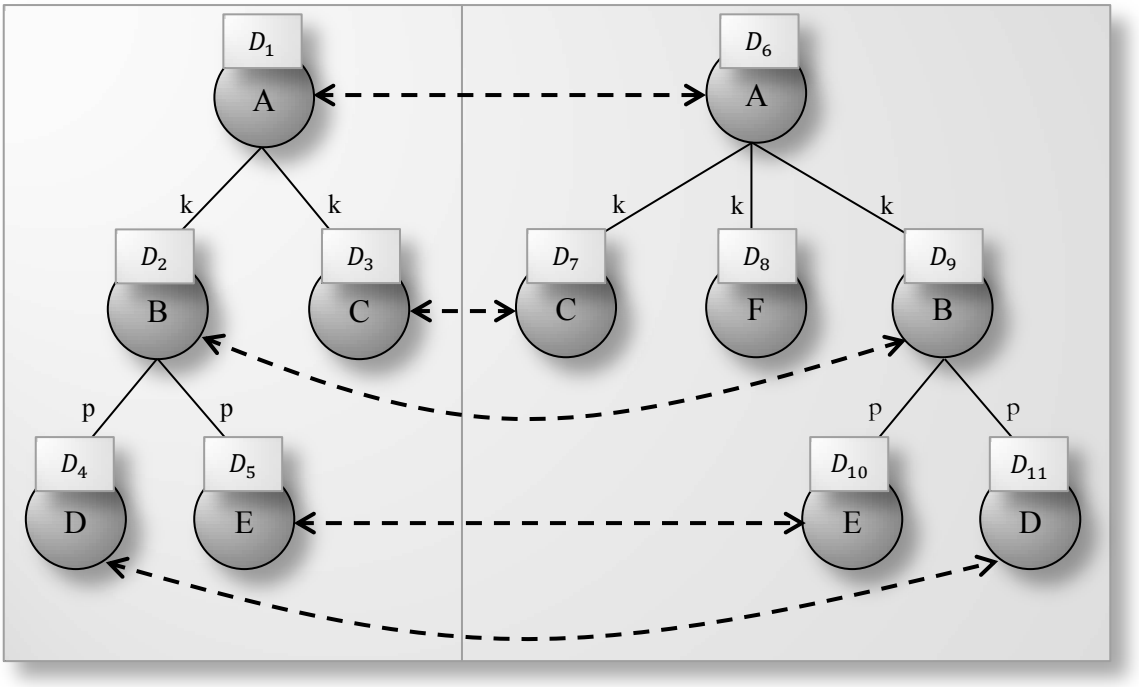


Figure 11. Incremental Matching Mapping between two Documents

The algorithm for change graph search, which we presented in Chapter 5, computes an optimal change script between two documents. Besides, it solves the dual problem of computing an optimal tree alignment between these documents. Therefore, we exploit the information contained in the computed change graph by the following algorithm to compute an incremental matching mapping between two interface documents.

Definition 6.2.1 (Incremental Matching Mapping): Let D_{old} and D_{new} be labeled trees, let Σ_S be a similarity specification and let Σ_E be an edit specification. Then we can construct an *incremental matching mapping* of the preserved subtrees of D_{old} in D_{new} by deriving the judgment $(D_{old}, D_{new}) \hookrightarrow_{INCMAP} \alpha$. The mapping α is an incremental matching mapping between D_{old} and D_{new} with respect to Σ_S. The operational semantics of $\hookrightarrow_{INCMAP}$ is defined by the following inference rules.

$$\frac{(D_{old}, D_{new}) \hookrightarrow_{DIFF} (\Delta, \mathrm{C}) \quad p \in \mathcal{Y}_C(\emptyset) \quad \Delta(p) = \overline{\Delta}_C(\emptyset) \quad A_C(p) \hookrightarrow_{SUBINC} \alpha}{(D_{old}, D_{new}) \hookrightarrow_{INCMAP} \alpha}$$

$$\frac{\Theta \backslash \Theta' = \{(D_x, D_y)\} \quad \forall D_z \in \Psi_{\preceq}^{\Delta}.\, \overrightarrow{D_z} \neq \overrightarrow{D_x} \quad A \hookrightarrow_{SUBINC} \alpha}{\{(\Theta, \Theta', \Delta)\} \cup A \hookrightarrow_{SUBINC} \{(\overrightarrow{D_x}, \overrightarrow{D_y})\} \cup \alpha}$$

$$\frac{}{\emptyset \hookrightarrow_{SUBINC} \emptyset}$$

$$\frac{\Theta \backslash \Theta' = \{(D_x, D_y)\} \quad \exists D_z \in \Psi_{\preceq}^{\Delta}.\, \overrightarrow{D_z} = \overrightarrow{D_x} \quad A \hookrightarrow_{SUBINC} \alpha}{\{(\Theta, \Theta', \Delta)\} \cup A \hookrightarrow_{SUBINC} \alpha}$$

Table 24. Algorithms INCMAP and SUBINC

The algorithm begins with computing the change graph C for D_{old} and D_{new}. Furthermore, let $p \in \mathcal{Y}_C(\emptyset)$ be a path to the goal node $\emptyset$ in the change graph C with an optimal corresponding change path script $\Delta(p) = \overline{\Delta}_C(\emptyset)$. The idea of this algorithm is that the critical tree pairs which are expanded on the path in the change graph are preserved subtrees, except of the case that a tree of a critical tree pair is removed by the change script. Note that a node in the change graph represents a set of critical tree pairs Θ. A node Θ is expanded to another node Θ' by reducing a critical tree pair $\Theta \backslash \Theta' = (D_x, D_y)$. Since a path contains finitely many nodes, this algorithm is terminating.

Definition 6.2.2 (Incremental Semantic Equality): Let Σ_S be a similarity specification and let α be an incremental matching mapping between two labeled trees D_1 and D_2 with respect to Σ_S. The *incremental semantic equality* of the labeled trees D_x and D_y with respect to Σ_S and α, denoted by $D_x \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_y$, is a predicate over pairs of labeled trees defined as follows:

$$D_x \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_y \;:\Leftrightarrow \begin{cases} (\overrightarrow{D_x}, \overrightarrow{D_y}) \in \alpha & if\ \overrightarrow{D_x} \in dom(\alpha) \\ (D_x \approx D_y) \wedge \left| \mathcal{T}_{(D_x,D_y)}^{\overset{=}{\equiv}_{(\Sigma_S,\alpha)}} \right| > 0 & if\ \overrightarrow{D_x} \notin dom(\alpha) \end{cases}$$

The difference to the standard semantic equality is that all preserved subtrees of D_1 which are contained in D_x must have a matching partner in D_y in conformance with the incremental matching mapping α.

Regarding our example in Figure 11, we can derive the following statements about preserved subtrees which satisfy the incremental semantic equality: $D_1 \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_6$, $D_2 \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_9$, $D_3 \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_7$, $D_4 \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_{11}$, and $D_5 \overset{=}{\equiv}_{(\Sigma_S,\alpha)} D_{10}$.

The second pillar of our incremental interpreter will be the caching of intermediate transformation results. We will collect the input and output mappings of all successfully invoked rules during the transformation process and we will call this information the *transformation trace*. The transformation process will then return the output document together with a sequence of transformation traces, denoted by Φ.

The incremental interpreter will operate on specific nodes in the parse tree. Since the parse tree is augmented by a constraint system, we will denote the current unifier for the set of constraints by θ. Furthermore, we will maintain an environment σ for modelling static scoping. We will denote the state of the *parse tree extensions* by $\mathcal{X} = (\theta, \sigma, \Phi)$.

We are now ready to introduce the incremental interpreter for our invertible grammar model. In the spirit of [Pereira & Warren, 1983], we will consider parsing as a deductive process. Therefore we will define the operational semantics of the incremental interpreter by inference rules which are employed by a backward reasoning process to construct the parse tree. We will introduce the interpreter from high-level to low-level algorithms using forward references in the description. All of the following algorithms operate in the context $\kappa = (I, \Phi, \alpha)$, consisting of an invertible grammar I, a sequence of transformation traces Φ of a previous transformation, and an incremental matching mapping α. Note that before the actual translation process begins, all text subtrees in the input document are tokenized with respect to the lexicon of the specified invertible grammar.

Definition 6.2.3 (Translation): Let I be a valid invertible grammar, let l_0 be the label of the start rule and let D_1 be a labeled tree. The judgment $D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi)$ denotes the *translation* of D_1 to the labeled tree D_2 with the sequence of transformation traces Φ with respect to the default context $\kappa = (I, \emptyset, \emptyset)$, unless another context is defined. The operational semantics of $\hookrightarrow_{TRANSLATE}$ is defined by the following inference rule.

$$\frac{([D_1], \mathcal{X}_1, l_0, l_{root}) \hookrightarrow_{PROCESS} ([D_2], \mathcal{X}_2)}{D_1 \hookrightarrow (D_2, \Phi_2)} \quad \begin{array}{l} \textit{with} \\ \mathcal{X}_1 \coloneqq (\emptyset, \sigma_{id}, \emptyset)\,, \\ \mathcal{X}_2 \coloneqq (\theta_2, \sigma_2, \Phi_2) \end{array}$$

Table 25. Algorithm TRANSLATE

The document D_1 is translated by calling the algorithm $\hookrightarrow_{PROCESS}$ with the initial parse tree extensions $\mathcal{X}_1 \coloneqq (\emptyset, \sigma_{id}, \emptyset)$, with the start rule label l_0 and an initial invocation label l_{root} which will be used as the root of the parse tree. Then the algorithm $\hookrightarrow_{PROCESS}$ transforms the sequence $[D_1]$ into $[D_2]$ and returns the final state of parse tree extensions $\mathcal{X}_2 = (\theta_2, \sigma_2, \Phi_2)$ which contains the sequence of transformation traces Φ_2.

An incremental translation of a new version of an interface document then calls the translation algorithm with a different context $\kappa = (I, \Phi, \alpha)$. This context contains the sequence of transformation traces of the translation of an old version of that document and an incremental matching mapping between the old version and the new version of that document.

Definition 6.2.4 (Incremental Translation): Let I be a valid invertible grammar, let D_{old} be the old labeled tree, let D_{new} be the new labeled tree and let Φ_1 be the sequence of transformation traces of the previous translation of D_{old}. The judgment $(D_{new}, D_{old}, \Phi_1) \hookrightarrow_{INCTRANSLATE} (D_2, \Phi_2)$ denotes the *incremental translation* of D_{new} to the labeled tree D_2 and the sequence of transformation traces Φ_2 with respect to the old labeled tree D_{old} and its sequence of transformation traces Φ_1. The operational semantics of $\hookrightarrow_{INCTRANSLATE}$ is defined by the following inference rule.

$$\frac{(D_{old}, D_{new}) \hookrightarrow_{INCMAP} \alpha \qquad D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi_2)}{(D_{new}, D_{old}, \Phi_1) \hookrightarrow (D_2, \Phi_2)} \quad \textit{with} \quad \kappa = (I, \Phi_1, \alpha)$$

Table 26. Algorithm INCTRANSLATE

First of all, an incremental matching mapping α between the old labeled tree D_{old} and the new labeled tree D_{new} is computed with respect to the input similarity specification Σ_S^I and the default edit specification Σ_E^d which assigns uniform edit weights and which does not limit the granularity of the edit operations. Together with the old sequence of transformation traces Φ_1, this information forms the context $\kappa = (I, \Phi_1, \alpha)$ for the translation of D_1 into D_2. This context will support the reuse of intermediate transformation results.

Next we will introduce the central algorithm of the incremental interpreter, the algorithm for the processing of a grammar rule.

Definition 6.2.5 (Rule Processing): Let $\kappa = (I, \Phi, \alpha)$ be a context, let $\mathcal{X}_1$ be parse tree extensions, let l_r be a rule label and let l_v be a subrule label. The judgment $(Y_1, \mathcal{X}_1, l_r, l_v) \hookrightarrow_{PROCESS} (Y_2, \mathcal{X}_2)$ denotes the *rule processing* which translates the input sequence Y_1 of labeled trees into the output sequence Y_2 of labeled trees and the new parse tree extensions $\mathcal{X}_2$. The operational semantics of $\hookrightarrow_{PROCESS}$ is defined by the following inference rule.

$$\frac{\begin{array}{c}(Y_1, \sigma_1, l_r) \hookrightarrow_{MATCH} (r, \mu, \gamma, \sigma_2) \\ r(l_v) = (l_r, P_{in}, M, P_{out}, C_{pre}, C_{post}, i) \\ (C_{pre}, \theta_1) \hookrightarrow_{EVAL} \theta_2 \\ (M, \mu, \emptyset, \mathcal{X}_2) \hookrightarrow_{INVOKE} (\gamma_2, \mathcal{X}_3) \\ (C_{post}, \theta_3) \hookrightarrow_{EVAL} \theta_4 \\ (\gamma, \gamma_2, P_{out}) \hookrightarrow_{COMBINE} (\gamma_3, Y_2) \\ \Phi_4 \coloneqq (l_r, r, Y_1, \mu, \gamma_3, Y_2) \star \Phi_3\end{array}}{(Y_1, \mathcal{X}_1, l_r, l_v) \hookrightarrow_{PROCESS} (Y_2, \mathcal{X}_4)}$$

with
$\mathcal{X}_1 = (\theta_1, \sigma_1, \Phi_1)$,
$\mathcal{X}_2 \coloneqq (\theta_2, \sigma_2, \Phi_1)$,
$\mathcal{X}_3 = (\theta_3, \sigma_3, \Phi_3)$,
$\mathcal{X}_4 \coloneqq (\theta_4, \sigma_4, \Phi_4)$,
$\sigma_4 \coloneqq i \; ? \; \sigma_3 : \; \sigma_1$

Table 27. Algorithm PROCESS

First, the algorithm $\hookrightarrow_{PROCESS}$ has to identify a grammar rule with the label l_r whose input pattern matches the input sequence Y_1. This task is handled by calling the algorithm $\hookrightarrow_{MATCH}$ which returns such a grammar rule, a matching mapping μ between the input pattern and the input sequence, and possibly a reusable matching mapping γ between the output pattern and the output sequence of the previous transformation. After that, the grammar rule is instantiated by the current rule invocation label l_v in the parse tree. Then, the set of precondition constraints C_{pre} of the grammar rule is incrementally evaluated with the current unifier θ_1 to the new unifier θ_2 by calling $\hookrightarrow_{EVAL}$.

The recursive rule invocations are then processed by calling the algorithm $\hookrightarrow_{INVOKE}$ which successively computes the substitution γ_2 for the output pattern, together with new parse tree extensions $\mathcal{X}_3$. Then, the set of postcondition constraints C_{post} of the grammar rule is incrementally evaluated with the current unifier θ_3 to the new unifier θ_4 by calling $\hookrightarrow_{EVAL}$. The (possibly empty) matching mapping γ, which we reuse from a previous transformation, is then overwritten by the computed matching mapping γ_2 by calling $\hookrightarrow_{COMBINE}$. In consequence, we recover the non-overwritten parts of γ, which are those parts of the original document that have not been translated. Thus, the matching mapping γ serves as the default substitution and as an oracle for parts of the tree which have been lost during the two-way transformation process. Furthermore, the algorithm $\hookrightarrow_{COMBINE}$ computes the resulting output sequence Y_2.

The resulting environment σ_4 is either the environment σ_3 inherited from the recursive rule invocations if $i = \top$ holds, or the environment σ_1 from the begin of this rule processing if $i = \bot$ holds. Finally, the transformation trace of this grammar rule is then collected by $\Phi_4 \coloneqq (l_r, r, Y_1, \mu, \gamma_3, Y_2) \star \Phi_3$. The algorithm $\hookrightarrow_{PROCESS}$ returns the computed output sequence Y_2, together with the new parse tree extensions $\mathcal{X}_4$.

The idea of the transformation trace is to store all information which is required to reproduce the transformation process, in particular to generate as an oracle those parts of the output which have been lost during the inverse transformation process.

Definition 6.2.6 (Transformation Trace): A *transformation trace* $(l_r, r, Y_1, \mu, \gamma, Y_2)$ contains the label l_r of a rule, the rule r itself, the input sequence Y_1, the input matching mapping μ, the output matching mapping γ, and the output sequence Y_2.

The space for storing transformation traces can be reduced by storing a reference to the rule instead of the rule itself, and by replacing the subtrees in the input and output sequence with references to subtrees in the input and output of the global transformation process. Finally, we can also use references in the matching mappings μ and γ. We will give an illustrating example with the following grammar rule which translates a specific step in a mathematical proof into its semantic representation.

FACT-2	
L	FACT
P_{IN}	[**z** "it" "holds" "that" **x** "."]
M	[(k,[**x**],[**v**],FORMS)]
P_{OUT}	[\fact[**v**]]
C_{PRE}	{ }
C_{POST}	{ }
i	{ true }

We assume the following input sequence Y_1 for this grammar rule.

$$Y_1 := [\ "Hence"\ "it"\ "holds"\ "that"\ "\$"\ "x"\ "\backslash in"\ "A"\ "\$"\ "."\ ]$$

Then the matching mapping μ for the input pattern P_{in} is computed as follows.

$$\mu := \{(\boldsymbol{z}, [\ "Hence"\ ]), (\boldsymbol{x}, [\ "\$"\ "x"\ "\backslash in"\ "A"\ "\$"\ ])\}$$

Assume that the recursive rule invocation leads to the following matching mapping γ.

$$\gamma := \{(\boldsymbol{v}, [\ \backslash F\{"in"\}[\backslash V\{"x"\}, \backslash V\{"A"\}\ ]\ ])\}$$

Then the rule processing terminates with the following output sequence Y_2.

$$Y_2 := [\ \backslash fact[\ \backslash F\{"in"\}[\backslash V\{"x"\}, \backslash V\{"A"\}\ ]\ ]$$

Hence, the transformation trace $(l_r, r, Y_1, \mu, \gamma, Y_2)$ is stored.

Now let us consider the inverse direction. The former output sequence Y_2 is now the new input sequence and the input matching mapping is γ, which allows to reuse the former matching mapping μ as a new default output matching mapping. Indeed, thanks to the transformation trace we can reconstruct the valid matching partner "$Hence$" for the pattern variable $\boldsymbol{z}$. Otherwise, this grammar rule is not applicable in the inverse direction.

The first step in the processing of grammar rules is the identification of a grammar rule with the label l_r whose input pattern matches the input sequence Y_1. This task is handled by the following algorithm $\hookrightarrow_{MATCH}$.

Definition 6.2.7 (Grammar Rule Matching): Let $\kappa = (I, \Phi, \alpha)$ be a context, let σ_1 be an environment and let l_r be a rule label. The judgment $(Y, \sigma_1, l_r) \hookrightarrow_{MATCH} (r, \mu, \gamma, \sigma_2)$ denotes that the input sequence Y matches the input pattern of the grammar rule r with the matching mapping μ and that the matching mapping γ can be reused from a previous transformation to complete the output substitution of the grammar rule. Furthermore, the new environment σ_2 is returned. The operational semantics of $\hookrightarrow_{MATCH}$ is defined by the following inference rules, where $D_{fresh} \in \mathcal{V}_I$ denotes a fresh semantic hash value and $v_{index} \coloneqq newvar(l_{ME}, l_{INDEX})$ denotes the constructed index constraint-variable.

(identity with introduction)

$$\frac{\sigma_2 \coloneqq \{(\mathcal{H}(D), D_{fresh})\} \circ \sigma_1 \quad C_{post} \coloneqq \{(v_{index}, D_{fresh})\} \quad r_{idi} \coloneqq (l_{idi}, [D], \emptyset, [D], \emptyset, C_{post}, \top)}{([D], \sigma_1, l_{idi}) \hookrightarrow (r_{idi}, \emptyset, \emptyset, \sigma_2)}$$

(identity with recognition)

$$\frac{C_{post} \coloneqq \{(v_{index}, \sigma(\mathcal{H}(D)))\} \quad r_{idr} \coloneqq (l_{idr}, [D], \emptyset, [D], \emptyset, C_{post}, \top)}{([D], \sigma, l_{idr}) \hookrightarrow (r_{idr}, \emptyset, \emptyset, \sigma)}$$

(identity)

$$\frac{r_{id} \coloneqq (l_{id}, Y, \emptyset, Y, \emptyset, \emptyset, \top)}{(Y, \sigma, l_{id}) \hookrightarrow (r_{id}, \emptyset, \emptyset, \sigma)}$$

(trace reuse with $\kappa = (I, \Phi, \alpha)$)

$$\frac{l_r \notin \{l_{id}, l_{idi}, l_{idr}\} \quad (l_r, r, Y_1, \mu, \gamma, Y_2) \in \Phi \quad r \in R \quad |\Omega^{\vec{=}(\Sigma_S, \alpha)}_{Y_1 \leftrightarrow Y}| > 0 \quad r = (l_r, P_{in}, M, P_{out}, C_{pre}, C_{post}, i) \quad (P_{in}, Y) \hookrightarrow_{VARMAP} \mu}{(Y, \sigma, l_r) \hookrightarrow (r, \mu, \gamma, \sigma)}$$

(standard rule matching)

$$\frac{l_r \notin \{l_{id}, l_{idi}, l_{idr}\} \quad r \in R \quad r = (l_r, P_{in}, M, P_{out}, C_{pre}, C_{post}, i) \quad (P_{in}, Y) \hookrightarrow_{VARMAP} \mu}{(Y, \sigma, l_r) \hookrightarrow (r, \mu, \emptyset, \sigma)}$$

Table 28. Algorithm MATCH

We will give a short description of the five inference rules by referring to them as being numbered from top to bottom and left to right. The first three inference rules are built-in identity rules, the only rules that are able to pass labeled trees from input to output, and as a semantic hash value to the constraints. This aspect is very important for the inversion of the grammar, because the built-in identity rules are invertible by construction. Note that the input and output pattern of the identity rules are equal and contain no variables.

The first inference rule for the rule label l_{idi} is an identity rule that additionally computes the semantic hash value of the input tree $\mathcal{H}(D)$ in the context Σ_S^I. Then the environment is modified to assign a fresh index value D_{fresh} to this semantic hash value. The fresh value is returned as the value of the constraint-variable v_{index}.

The second inference rule for the rule label l_{idr} is similar to the first rule, but it does not assign a fresh index value to the semantic hash value. Instead, it returns the value currently assigned to the semantic hash value by the environment as the value of the constraint-variable v_{index}.

The third inference rule for the rule label l_{id} is also an identity rule but it just returns the input sequence as the output sequence, without indexing the input tree.

The fourth inference rule tries to exploit the information contained in the translation context $\kappa = (I, \Phi, \alpha)$. It looks for a transformation trace with a rule $r \in R$ labeled by l_r. If a tree matching mapping exists between the stored input sequence Y_1 and the current input sequence Y with respect to incremental semantic equality, that is, $|\Omega_{Y_1 \leftrightarrow Y}^{\rightleftharpoons(\Sigma_{S,\alpha})}| > 0$ holds, then the inference rule verifies whether the stored input matching mapping μ is a variable mapping between the input pattern P_{in} and the current input sequence Y by calling $\hookrightarrow_{VARMAP}$ for forward reasoning. Note that all transformation traces are ordered, thus the choice of an alternative trace during backtracking is deterministic.

The fifth inference rule is the standard non-identity inference rule which selects a rule $r \in R$ having the rule label l_r and computes an input matching mapping μ between the input pattern P_{in} and the current input sequence Y by calling the algorithm $\hookrightarrow_{VARMAP}$ with the context (Σ_S^I, Σ_V) for backward reasoning. All rules in the grammar are ordered, thus the choice of an alternative rule during backtracking is deterministic. Note that the input pattern is always ordered on top level.

Additionally, we define the following processing priority between the fourth and the fifth inference rule: All application alternatives of the fourth inference rule have to be tried first before the fifth inference rule is used. Thus, first we try to reuse the information contained in the sequence of transformation traces before we use the standard rule matching. This allows for a deterministic processing of the algorithm $\hookrightarrow_{MATCH}$ and its backtracking mechanism.

The following algorithm $\hookrightarrow_{VARMAP}$ computes or verifies a variable mapping between a pattern and a sequence of labeled trees. A variable mapping is a mapping of the variables contained in the pattern to subtrees contained in the sequence of labeled trees.

Definition 6.2.8 (Variable Mapping): Let Σ_S be a similarity specification and let $\Sigma_V = (Z_+, Z_-, \Lambda)$ be a variable specification. The judgment $(P, Y) \hookrightarrow_{VARMAP} \mu$ denotes that μ is a variable mapping between the pattern P and the sequence of labeled trees Y. The operational semantics of $\hookrightarrow_{VARMAP}$ is defined by the following inference rules.

$$\frac{f \in \Omega_{P \leftrightarrow Y}^{=(\Sigma_S, \Sigma_V)} \quad f \hookrightarrow_{SUBMAP} f'}{(P, Y) \hookrightarrow_{VARMAP} f'} \qquad \frac{x \notin \mathcal{V} \quad f_R \in \mathcal{T}_{(x,y)}^{=(\Sigma_S, \Sigma_V)} \quad f_R \cup f \hookrightarrow_{SUBMAP} f'}{\{(x, y)\} \cup f \hookrightarrow_{SUBMAP} f'}$$

$$\frac{x \in \mathcal{V} \quad f \hookrightarrow_{SUBMAP} f'}{\{(x, y)\} \cup f \hookrightarrow_{SUBMAP} \{(x, y)\} \cup f'} \qquad \frac{}{\emptyset \hookrightarrow_{SUBMAP} \emptyset}$$

Table 29. Algorithms VARMAP and SUBMAP

This algorithm generates a matching mapping $f \in \Omega_{P \leftrightarrow Y}^{=(\Sigma_S, \Sigma_V)}$ between the pattern and the sequence, and recursively for all layers in matched non-variable trees by the algorithm $\hookrightarrow_{SUBMAP}$. The resulting mapping is restricted to variables and finally returned. Since we totally order the matching mappings between a pattern and a variable-free sequence of labeled trees by the amount of labeled trees matched with the pattern variables, the algorithm $\hookrightarrow_{VARMAP}$ and its backtracking mechanism are deterministic.

The second step in the processing of a grammar rule is the satisfiability check of the set of precondition constraints C_{pre} with respect to the current unifier θ_1 of the set of constraints in the parse tree C_1. The satisfiability check by the algorithm $\hookrightarrow_{SAT}$ terminates with a new unifier θ_2 for the new set of constraints $C_{pre} \cup C_1$.

Definition 6.2.9 (Constraint Evaluation): Let Σ_S be a similarity specification. The judgment $(C_0, \theta_1) \hookrightarrow_{EVAL} \theta_2$ denotes the *constraint evaluation* of the set of constraints C_0 with respect to the current unifier θ_1 to the new unifier θ_2 which is a consistent extension of θ_1. The operational semantics of $\hookrightarrow_{EVAL}$ is defined by the following inference rule.

$$\frac{(C_1', \theta_1) \hookrightarrow_{SAT} \theta_2}{(C_0, \theta_1) \hookrightarrow \theta_2} \qquad \text{with} \quad C_1' \coloneqq \theta_1(\theta_1[C_0])$$

Table 30. Algorithm EVAL

First, the substitution θ_1 is used to evaluate the meta-variables in C_0, denoted by $\theta_1[C_0]$. Then the substitution θ_1 is applied to the resulting set of constraints. Finally, the result C_1' is incrementally evaluated by calling the algorithm $\hookrightarrow_{SAT}$ with the substitution θ_1. Since θ_1 is a valid substitution and $vars(C_1') \cap dom(\theta_1) = \emptyset$ holds, the requirements of $\hookrightarrow_{SAT}$ are satisfied. The resulting unifier θ_2 is finally returned.

Note in case that the similarity order Σ_O has been used in the similarity specification for the constraint system, the incremental constraint evaluation might fail although the overall set of constraints is satisfiable. Since we use meta-variables in the set of constraints, we cannot just compute a new unifier for the current set of constraints in the parse tree. Thus, the backtracking method of the incremental grammar interpreter has to explore all alternatives in the choice points of the incremental constraint evaluation. We suggest waiving the use of the similarity order (when possible) in the constraints for the benefit of having a method for constraint evaluation without the need for backtracking.

The third step in the processing of a grammar rule is the recursive rule invocation.

Definition 6.2.10 (Recursive Rule Invocation): Let $\kappa = (I, \Phi, \alpha)$ be a context, let M be a sequence of rule invocations, let μ be an input matching mapping, let γ_1 be the collected output matching mapping and let $\mathcal{X}_1$ be the current parse tree extensions. The judgment $(M, \mu, \gamma_1, \mathcal{X}_1) \hookrightarrow_{INVOKE} (\gamma_2, \mathcal{X}_2)$ denotes the *recursive rule invocation* of the ordered sequence M to compute an output matching mapping γ_2 and new parse tree extensions $\mathcal{X}_2$. The operational semantics of $\hookrightarrow_{INVOKE}$ is defined by the following inference rules.

$$\frac{\begin{array}{c}(\mu(P_i), \mathcal{X}_1, l_r, l_v) \hookrightarrow_{PROCESS} (Y_2, \mathcal{X}_2) \\ (\gamma_1(P_o), Y_2) \hookrightarrow_{VARMAP} \gamma_2 \\ (M, \mu, \gamma_1 \oplus \gamma_2, \mathcal{X}_2) \hookrightarrow (\gamma_4, \mathcal{X}_3)\end{array}}{\big((l_v, P_i, P_o, l_r) \star M, \mu, \gamma_1, \mathcal{X}_1\big) \hookrightarrow (\gamma_4, \mathcal{X}_3)} \qquad \frac{}{(\emptyset, \mu, \gamma, \mathcal{X}) \hookrightarrow (\gamma, \mathcal{X})}$$

Table 31. Algorithm INVOKE

The algorithm $\hookrightarrow_{INVOKE}$ processes all recursive rule invocations contained in M in the order of the sequence. An invocation $m = (l_v, P_i, P_o, l_r)$ is processed by first substituting the input variables in the input pattern of the rule invocation by the substitution μ. The resulting invocation input sequence is then recursively processed by calling the algorithm $\hookrightarrow_{PROCESS}$ with the rule label l_r and the subrule label l_v of the invocation m. Hence, we obtain an invocation output sequence Y_2 and new parse tree extensions $\mathcal{X}_2$. Next we apply the already computed partial output substitution γ_1 to the output pattern P_0.

Finally, a variable matching mapping γ_2 is computed between the resulting pattern and the computed invocation output sequence Y_2. The new partial output substitution is then $\gamma_1 \oplus \gamma_2$. And the algorithm $\hookrightarrow_{INVOKE}$ proceeds with the next rule invocation.

The fourth step in the processing of a grammar rule is the satisfiability check of the set of postcondition constraints with respect to the current *constraint evaluation context*. For this purpose, we use again the algorithm $\hookrightarrow_{EVAL}$. Finally, in the fifth processing step the collected output matching mapping is extended by the information from the sequence of transformation traces. All information is combined to construct the output sequence.

Definition 6.2.11 (Result Combination): Let γ_1 and γ_2 be output matching mappings and let P_{out} be the current parse tree extensions. The judgment $(\gamma_1, \gamma_2, P_{out}) \hookrightarrow_{COMBINE} (\gamma_3, Y)$ denotes the *result combination* of both output matching mappings and the output pattern to a complete output matching mapping γ_3 and an output sequence Y. The operational semantics of $\hookrightarrow_{COMBINE}$ is defined by the following inference rule.

$$\frac{\begin{array}{c}\gamma_3 \coloneqq \gamma_1 \oplus \gamma_2 \\ Y \coloneqq \gamma_3(P_{out}) \\ vars(Y) = \emptyset\end{array}}{(\gamma_1, \gamma_2, P_{out}) \hookrightarrow (\gamma_3, Y)}$$

Table 32. Algorithm COMBINE

This algorithm finally extends the collected output matching mapping γ_2 by the information recovered from the output matching mapping γ_1 contained in a transformation trace. The resulting substitution γ_3 is applied to the output pattern P_{out} of the grammar rule. We require that the resulting output sequence does not contain any variable.

All presented algorithms of the incremental interpreter are deterministic. For all algorithms with a choice point we defined an order for the processing of alternatives. The only choice points are in fact in the algorithms $\hookrightarrow_{MATCH}$ for finding a matching grammar rule, $\hookrightarrow_{VARMAP}$ for finding a variable matching mapping for patterns, and $\hookrightarrow_{SAT}$ for the incremental constraint evaluation. Altogether, we presented an incremental interpreter for our invertible grammar model that is composed of multiple deterministic algorithms with a deterministic backtracking mechanism. In consequence, we do not have to remember failed alternatives during the parsing process, which reduces the space complexity of the incremental interpreter.

We will now prove the termination of the incremental interpreter, beginning with the termination of the processing of grammar rules.

Lemma 6.2.12 (Termination of the Rule Processing): Let Y_1 be a sequence of labeled trees, let $\mathcal{X}_1$ be parse tree extensions, let l_r be a rule label and let l_v be a rule invocation label. Then it holds that $(Y_1, \mathcal{X}_1, l_r, l_v) \hookrightarrow_{PROCESS} (Y_2, \mathcal{X}_2)$ terminates.

Proof: The algorithm $\hookrightarrow_{PROCESS}$ terminates because all substitutions are finite and because the called algorithms $\hookrightarrow_{MATCH}$, $\hookrightarrow_{EVAL}$, $\hookrightarrow_{INVOKE}$ and $\hookrightarrow_{COMBINE}$ terminate, as we will show. The algorithm $\hookrightarrow_{EVAL}$ terminates because the algorithm $\hookrightarrow_{SAT}$ terminates by *Lemma 6.1.11* and because the algorithms $\hookrightarrow_{META}$ and $\hookrightarrow_{APPLY}$ terminate. The algorithm $\hookrightarrow_{INVOKE}$ terminates because the sequence of rule invocations M is reduced in every step and because the algorithm $\hookrightarrow_{VARMAP}$ terminates, as we will show. The mutual recursion between the algorithms $\hookrightarrow_{PROCESS}$ and $\hookrightarrow_{INVOKE}$ terminates because $\hookrightarrow_{INVOKE}$ calls $\hookrightarrow_{PROCESS}$ always with a smaller input sequence because of the smaller size property of subrules as required by *Definition 6.1.16* for a valid invertible grammar. The algorithm $\hookrightarrow_{COMBINE}$ terminates because $\hookrightarrow_{APPLY}$ terminates. It remains to be shown that the algorithms with choice points terminate.

The algorithm $\hookrightarrow_{MATCH}$ terminates in the cases of identity rules because the semantic hash indexing terminates. In the more general cases, the algorithm $\hookrightarrow_{MATCH}$ terminates because the algorithm $\hookrightarrow_{VARMAP}$ terminates. The backtracking mechanism terminates for $\hookrightarrow_{MATCH}$ because there are only finitely many alternatives, either applicable transformation traces or applicable rules in the invertible grammar. Finally, the algorithm $\hookrightarrow_{VARMAP}$ terminates because it calls itself recursively with a smaller argument size. Furthermore, the backtracking mechanisms for $\hookrightarrow_{VARMAP}$ and $\hookrightarrow_{SAT}$ terminate because there are only finitely many alternative matching mappings. Altogether, we have shown that $(Y_1, \mathcal{X}_1, l_r, l_v) \hookrightarrow_{PROCESS} (Y_2, \mathcal{X}_2)$ terminates.

■

The termination of the translation of an interface document follows immediately.

Theorem 6.2.13 (Termination of the Translation): Let D_1 be a labeled tree. Then it holds that $D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi_1)$ terminates.

Proof: The algorithm $\hookrightarrow_{TRANSLATE}$ terminates because the constructed input sequence is finite and the algorithm $\hookrightarrow_{PROCESS}$ terminates by *Lemma 6.2.12*.

■

Finally, the termination of the incremental translation can be proven.

Theorem 6.2.14 (Termination of the Incremental Translation): Let D_{new} be a labeled tree, let D_{old} be a labeled tree and let Φ_1 be a sequence of transformation traces from a previous translation of D_{old}. Then it follows that the derivation of $(D_{new}, D_{old}, \Phi_1) \hookrightarrow_{INCTRANSLATE} (D_4, \Phi_2)$ terminates.

Proof: The algorithm $\hookrightarrow_{INCTRANSLATE}$ terminates because $\hookrightarrow_{DIFF}$ terminates by *Theorem 5.3.12*, because the constructed input sequence is finite and because $\hookrightarrow_{PROCESS}$ terminates by *Lemma 6.2.12*.

■

Besides the termination, we are also interested in the consistency of the results of the full translation algorithm and the incremental translation algorithm. This means that the incremental translation of an interface document should produce the same result as the plain translation. Let us analyze the requirements for this consistency property.

Both translation algorithms call the algorithm $\hookrightarrow_{PROCESS}$. The only difference is that the incremental translation passes a context $\kappa_1 = (I, \Phi_1, \alpha)$ whereas the standard translation uses the default context $\kappa_2 = (I, \emptyset, \emptyset)$. Since the incremental interpreter operates completely deterministic, we have to analyze the impact of Φ_1 and α on the different algorithms. The only algorithm which uses this information of the context is the rule matching algorithm $\hookrightarrow_{MATCH}$. The processing priority is defined in such a way that the sequence of transformation traces Φ_1 is first exploited before the grammar rules are checked in the standard rule matching case.

Observation 6.2.15 (Consistency Criterion): In order to guarantee the consistency property, we have to exclude the possibility that the algorithm $\hookrightarrow_{MATCH}$ selects a transformation trace which produces a wrong but valid output. Since we cannot predict the effect of a choice, we have to guarantee that at most one transformation trace is a valid choice for the algorithm $\hookrightarrow_{MATCH}$.

Before we will present some possibilities to guarantee this consistency criterion, we will now prove the consistency of the incremental interpreter under the assumption of this consistency criterion.

Theorem 6.2.16 (Consistency of the Incremental Interpreter): Let I be a valid invertible grammar and let D_1 be a labeled tree. If we have $D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi_1)$ and the consistency criterion holds, then it follows that $(D_1, D_1, \Phi_1) \hookrightarrow_{INCTRANSLATE} (D_2, \Phi_1)$.

Proof: The consistency criterion guarantees that there is always at most one matching transformation trace in the set Φ_1 for the grammar rule matching algorithm $\hookrightarrow_{MATCH}$. Thus in contrast to the original transformation, the incremental transformation selects at this choice point directly the rule which led to the final parse tree in the original transformation. By employing the sequence of transformation traces we only bypass the steps which have been deterministically backtracked anyway by selecting an alternative grammar rule in the algorithm $\hookrightarrow_{MATCH}$. Hence the incremental transformation returns the same result and we have $(D_1, D_1, \Phi_1) \hookrightarrow_{INCTRANSLATE} (D_2, \Phi_1)$.

■

In the following, we will describe possibilities to check the consistency criterion of the incremental interpreter both statically and dynamically for a given translation context $\kappa = (I, \Phi, \alpha)$.

Dynamic Consistency Check. First of all, one can check this criterion dynamically for every translation turn by verifying for every pair of transformation traces $(l_r, r, Y_1, \mu, \gamma, Y_2)$ and $(l_r, r', Y_1', \mu', \gamma', Y_2')$ in Φ with equal rule label l_r that $\left|\Omega^{\rightleftharpoons_{(\Sigma_S,\alpha)}}_{Y_1 \leftrightarrow Y_1'}\right| = 0$ holds. By the symmetry and transitivity of $\rightleftharpoons_{(\Sigma_S,\alpha)}$ it follows that at most one transformation trace is a valid choice for $\hookrightarrow_{MATCH}$.

Static Consistency Check. Another option for checking the consistency criterion, is to statically check for additional restrictive properties of the invertible grammar I that implicitly guarantee that the dynamic check always succeeds. One possibility is to require for every grammar rule that all invocation input patterns contain only disjunctive subtrees of the input pattern. Since the invocation patterns must have a smaller size than the original pattern, the input pattern of all processed rules contain only preserved subtrees and are thus unique with respect to $\rightleftharpoons_{(\Sigma_S,\alpha)}$. Hence, the dynamic check for the consistency criterion will always succeed. Note that the static check does not consider the effects of the constraint system. There may well exist invertible grammars which do not pass the static check but satisfy the consistency criterion because the evaluation of the constraints only succeeds with one of the matching transformation traces in all cases.

6.3 Inversion

The main motivation for the introduction of the invertible grammar formalism was the ability to invert the grammar in order to transform between the interface documents of two components in both directions. A particular problem of this scenario is that both interface documents contain related information with a different level of detail and in different representations. In the following, we will describe how the inversion of the grammar can be automated and used for the inverse transformation.

A related approach in the context of attribute grammars has been presented in [Yellin & Mueckstein, 1985]. Yellin proposes to restrict attribute grammars by requiring the exclusive use of token-permuting functions in the attribute system. These functions are not allowed to modify or delete any of its arguments. Thus, this approach is only suitable for transformations which permute parts of the interface document. Our scenario does not satisfy these strict requirements.

We have designed the invertible grammar model with the primary aim of automating the inversion of the grammar. The principle idea of the inversion is to swap the input and output patterns in the grammar rules and their rule invocations, as shown in Figure 12. Thus, the inversion of an invertible grammar is first of all based on the inversion of the built-in identity rules. This is guaranteed because the input and output pattern of the identity rules are equal by construction. Hence, the inversion of an identity rule results in the same identity rule.

$$r = \left(l_r, \boldsymbol{P_{in}}, M, \boldsymbol{P_{out}}, C_{pre}, C_{post}, i\right)$$

$$with \quad M = [(l_m, \boldsymbol{P_i}, \boldsymbol{P_o}, l_r), \ldots, (l'_m, \boldsymbol{P'_i}, \boldsymbol{P'_o}, l'_r)]$$

Figure 12. Principle Idea of the Inversion

This approach would be already sufficient for inverting simple content permuting rules as used by [Yellin & Mueckstein, 1985]. But in our scenario, not all parts of the interface document are translated. Therefore, we collect the sequence of transformation traces during the translation process. A transformation trace contains among other information the input and output variable matching mappings, thus in particular the matching partners of variables which are not used in the translation process. This information can be exploited by the inverse translation process to generate the non-translated parts of an interface document, as we will outline in the correctness proof for the inverse translation.

First of all, we will present methods for the automated inversion of the grammar and the automated inversion of a sequence of transformation traces. With these methods, we will then model the inverse transformation of an interface document as an incremental transformation with the inverted grammar and the inverted sequence of transformation traces. Thereby, we will analyze the requirements for a consistent inverse transformation. We begin with presenting a method for inverting rule invocations.

Definition 6.3.1 (Rule Invocation Inversion): Let M be a sequence of rule invocations. The judgment $M \hookrightarrow_{INVINVOCATIONS} M^{-1}$ denotes that M is inverted to the sequence of rule invocations M^{-1}. The operational semantics of $\hookrightarrow_{INVINVOCATIONS}$ is defined by the following inference rules.

$$\frac{M \hookrightarrow M^{-1}}{(l_m, P_i, P_o, l_r) \star M \hookrightarrow (l_m, P_o, P_i, l_r) \star M^{-1}} \qquad \frac{}{[\,] \hookrightarrow [\,]}$$

Table 33. Algorithm INVINVOCATIONS

A rule invocation is inverted by swapping its input and output pattern. A sequence of rule invocations is inverted by inverting every rule invocation in that sequence.

We continue by presenting the inversion method for a sequence of grammar rules.

Definition 6.3.2 (Rule Inversion): Let R be a sequence of grammar rules. The judgment $R \hookrightarrow_{INVRULES} R^{-1}$ denotes that R is inverted to the sequence of grammar rules R^{-1}. The operational semantics of $\hookrightarrow_{INVRULES}$ is defined by the following inference rule.

$$\frac{\begin{array}{c} M \hookrightarrow_{INVINVOCATIONS} M^{-1} \\ R \hookrightarrow R^{-1} \end{array}}{(l_r, P_{in}, M, P_{out}, C_{pre}, C_{post}, i) \star R \hookrightarrow (l_r, P_{out}, M^{-1}, P_{in}, C_{pre}, C_{post}, i) \star R^{-1}} \quad \frac{}{[\,] \hookrightarrow [\,]}$$

Table 34. Algorithm INVRULES

A grammar rule is inverted by swapping its input and output pattern, and by inverting its sequence of rule invocations. A sequence of grammar rules is inverted by inverting every rule in that sequence.

Then, we can define the method for inverting a complete grammar as follows.

Definition 6.3.3 (Grammar Inversion): Let I be an invertible grammar. The judgment $I \hookrightarrow_{INVGRAMMAR} I^{-1}$ denotes that I is inverted to the grammar I^{-1}. The operational semantics of $\hookrightarrow_{INVGRAMMAR}$ is defined by the following inference rule.

$$\frac{R \hookrightarrow_{INVRULES} R^{-1}}{(\mathcal{L}_T, R, l_0, \Sigma_S^I, \Sigma_S^O, \Sigma_S^C, \Sigma_V) \hookrightarrow (\mathcal{L}_T, R^{-1}, l_0, \Sigma_S^O, \Sigma_S^I, \Sigma_S^C, \Sigma_V)}$$

Table 35. Algorithm INVGRAMMAR

A grammar is inverted by inverting the sequence of rules and swapping the input and output similarity specifications.

Finally, we introduce the method for inverting a sequence of transformation traces.

Definition 6.3.4 (Trace Inversion): Let Φ be a sequence of transformation traces. The judgment $\Phi \hookrightarrow_{INVTRACES} \Phi^{-1}$ denotes that Φ is inverted to the sequence of transformation traces Φ^{-1}. The operational semantics of $\hookrightarrow_{INVTRACES}$ is defined by the following inference rules.

$$\frac{r \hookrightarrow_{INVRULE} r^{-1} \quad \Phi \hookrightarrow \Phi^{-1}}{(l_r, r, Y_1, \mu, \gamma, Y_2) \star \Phi \hookrightarrow (l_r, r^{-1}, Y_2, \gamma, \mu, Y_1) \star \Phi^{-1}} \qquad \frac{}{[\,] \hookrightarrow [\,]}$$

Table 36. Algorithm INVTRACES

A sequence of transformation traces is inverted by inverting its rule, swapping the input and output sequence, and swapping the input and output matching mapping. A sequence of transformation traces is inverted by inverting every contained transformation trace.

Altogether, we have presented terminating and deterministic methods for the automated inversion of a grammar and the automated inversion of a sequence of transformation traces. The principle idea of these methods is to swap the patterns, mappings and specifications between input and output.

In the following, we will discuss how an inverted grammar and an inverted sequence of transformation traces can be used to translate a labeled tree in the inverse direction. First of all, we will state that the inversion of a valid invertible grammar produces a valid invertible grammar.

Lemma 6.3.5 (Validity of Inverted Valid Grammars): Let I be a valid invertible grammar and let $I \hookrightarrow_{INVGRAMMAR} I^{-1}$ hold. It follows that I^{-1} is a valid invertible grammar.

Proof: Since the inversion only swaps the input and output patterns of the grammar rules and their rule invocations, this follows from the properties for validity in *Definition 6.1.16* which are defined equally for the input and output patterns of a grammar rule and its rule invocations.

■

Therefore, we can model the inverse translation as an incremental translation with the inverted valid grammar of the original translation.

Definition 6.3.6 (Inverse Translation): Let I be a valid invertible grammar and let D_1, D_2, D_3 and D_4 be labeled trees. If we have the translation $D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi_1)$ then we define the *inverse translation* by $(D_3, D_2, \Phi_1^{-1}) \hookrightarrow_{INCTRANSLATE} (D_4, \Phi_2^{-1})$ with $\Phi_1 \hookrightarrow_{INVTRACES} \Phi_1^{-1}$. The inverse translation then calls the algorithm $\hookrightarrow_{TRANSLATE}$ with the translation context $\kappa = (I^{-1}, \Phi_1^{-1}, \alpha)$ where $I \hookrightarrow_{INVGRAMMAR} I^{-1}$ holds.

The termination of the inverse translation follows immediately.

Theorem 6.3.7 (Termination of Inverse Translation): Let I be a valid invertible grammar and $I \hookrightarrow_{INVGRAMMAR} I^{-1}$, let D_{new} be a labeled tree, let D_{old} be a labeled tree and let Φ_1^{-1} be a sequence of transformation traces from a previous translation of D_{old} with the grammar I^{-1}. Then it follows that $(D_{new}, D_{old}, \Phi_1^{-1}) \hookrightarrow_{INCTRANSLATE} (D_4, \Phi_2^{-1})$ terminates with the grammar I^{-1}.

Proof: Since I^{-1} is a valid invertible grammar by *Lemma 6.3.5*, this follows directly from the termination of the incremental translation in *Theorem 6.2.14*.

■

In the following we will show that the inverse translation of a translated interface document results in an interface document which is semantically equal to the original one. We can only prove this statement by assuming the consistency criterion stated in *Observation 6.2.15*. The reason is that we need to guarantee that we can always exploit the correct transformation trace for generating content which has not been translated. To achieve this, we need to extend our proposed checks for the consistency criterion to the inversion case. In particular, we have to duplicate all requirements for the input patterns to the output patterns, since input and output are inverted.

Dynamic Inverse Consistency Check. We can check the *inverse consistency criterion* dynamically for every translation turn by verifying for every pair of transformation traces $(l_r, r, Y_1, \mu, \gamma, Y_2)$ and $(l_r, r', Y_1', \mu', \gamma', Y_2')$ in Φ with equal rule label l_r that both $\left|\Omega_{Y_1 \leftrightarrow Y_1'}^{\rightleftharpoons_{(\Sigma_S,\alpha)}}\right| = 0$ and $\left|\Omega_{Y_2 \leftrightarrow Y_2'}^{\rightleftharpoons_{(\Sigma_S,\alpha)}}\right| = 0$ hold. By the symmetry and transitivity of $\rightleftharpoons_{(\Sigma_S,\alpha)}$ it follows that at most one trace is a valid choice for $\hookrightarrow_{MATCH}$ in both translation directions.

Static Inverse Consistency Check. We can also statically check additional restrictive properties of the invertible grammar I that implicitly guarantee that the dynamic check always succeeds. One possibility is to require for every grammar rule that all invocation input patterns contain only disjunctive subtrees of the input pattern, and that all invocation output patterns contain only disjunctive subtrees of the output pattern. Since the invocation patterns must have a smaller size than the original pattern, the input and output patterns of all processed rules contain only preserved subtrees and are thus unique with respect to $\rightleftharpoons_{(\Sigma_S,\alpha)}$. Hence, the inverse consistency criterion is satisfied in both directions.

We will now prove the correctness of the inverse translation under the assumption that the inverse consistency criterion is satisfied. We begin by proving the correctness of the inverse rule processing which intuitively means that if we translate a sequence Y_1 into Y_2, the inverse translation of Y_2 will produce Y_1' which is semantically equal to Y_1.

Lemma 6.3.8 (Correctness of the Inverse Rule Processing): Let I be a valid invertible grammar, let $I \hookrightarrow_{INVGRAMMAR} I^{-1}$ hold and let the inverse consistency criterion be satisfied. Let Y_1 and Y_2 be sequences of labeled trees, let $\mathcal{X}_1$ and $\mathcal{X}_2$ be parse tree extensions, and let Φ be a sequence of transformation traces. If we have derived the judgment $(Y_1, \mathcal{X}_1, l_r, l_v) \hookrightarrow_{PROCESS} (Y_2, \mathcal{X}_2)$, then $(Y_2, \mathcal{X}_1', l_r, l_v) \hookrightarrow_{PROCESS} (Y_1', \mathcal{X}_2')$ follows with $\left|\Omega_{Y_1 \leftrightarrow Y_1'}^{=_{\Sigma_S^I}}\right| > 0$ under the context $\kappa = (I^{-1}, \Phi^{-1}, \alpha)$ where $\mathcal{X}_2 = (\theta, \sigma, \Phi)$ and $\Phi \hookrightarrow_{INVTRACES} \Phi^{-1}$ holds. Thereby, the parse tree extensions $\mathcal{X}_1'$ and $\mathcal{X}_2'$ are equal to $\mathcal{X}_1$ and $\mathcal{X}_2$ respectively modulo the sequence of transformation traces.

Proof: We prove this statement by induction over the derivation tree for $(Y_2, \mathcal{X}_1', l_r, l_v) \hookrightarrow_{PROCESS} (Y_1', \mathcal{X}_2')$ with the grammar I^{-1}. For all cases, we know that $(Y_1, \sigma_1, l_r) \hookrightarrow_{MATCH} (r, \mu, \gamma, \sigma_2)$ holds. The inverse consistency criterion guarantees at most one matching transformation trace for the algorithm $\hookrightarrow_{MATCH}$. Since furthermore the algorithm $\hookrightarrow_{PROCESS}$ stores the output sequence Y_2 in the transformation trace, it follows that the transformation trace created by $\hookrightarrow_{PROCESS}$ in the first translation turn is matched by the inverse rule processing and it holds that $(Y_2, \sigma_1, l_r) \hookrightarrow_{MATCH} (r^{-1}, \gamma, \mu, \sigma_2)$ with $r \hookrightarrow_{INVRULE} r^{-1}$.

Furthermore, note that the processing of the constraint evaluation context is completely independent of the overall translation direction. In particular, the built-in identity rules compute an equal semantic hash value for the given labeled tree, regardless of whether it stems from the input or output pattern of this rules. Therefore, we will take this aspect of the proof for granted.

The base case for the induction is the case where the matched grammar rule r^{-1} does not have any rule invocations, and the step case is the case where r^{-1} does have at least one rule invocation.

(Base Case) In this case, the rule r^{-1} is either one of the built-in identity grammar rules or an inverted original grammar rule. However, the rule r^{-1} does not have any rule invocations, hence the algorithm $\hookrightarrow_{INVOKE}$ returns an empty substitution. The substitution μ from the matched transformation trace is then combined with this empty substitution by the algorithm $\hookrightarrow_{COMBINE}$, and the result μ is applied to the output pattern of the inverted grammar rule, which is the former input pattern of the original grammar rule. Note that the substitution μ contains in particular the matching partners of those variables that have not been used in the recursive translation process. Thus, we obtain Y_1' as the output sequence together with the set containing the inverted transformation trace of the current grammar rule. Altogether $(Y_2, \mathcal{X}_1', l_r, l_v) \hookrightarrow_{PROCESS} (Y_1', \mathcal{X}_2')$ holds in this case and from $(Y_1, \sigma_1, l_r) \hookrightarrow_{MATCH} (r, \mu, \gamma, \sigma_2)$ and in particular $(P_{in}, Y_1) \hookrightarrow_{VARMAP} \mu$ it follows that $\left|\Omega^{=_{\Sigma_S^I}}_{Y_1 \leftrightarrow Y_1'}\right| > 0$ holds additionally.

(Step Case) In this case, the rule r^{-1} is an inverted original grammar rule with at least one rule invocation. The algorithm $\hookrightarrow_{INVOKE}$ recursively calls $\hookrightarrow_{PROCESS}$ for every rule invocation with an invocation input sequence constructed by applying the substitution γ to the invocation input pattern. By the induction hypothesis it follows that the processing of these inverted rule invocations produce outputs which are semantically equal to the original invocation input sequences. Therefore, the algorithm $\hookrightarrow_{INVOKE}$ returns a substitution

that contains semantically equal variable mapping partners. By combining the matching mapping μ from the matched transformation trace with the computed substitution, the algorithm $\hookrightarrow_{COMBINE}$ essentially extends the computed substitution by the original matching partners of the variables which have not been translated. Finally, By applying this substitution we hence obtain an output sequence Y_1' with $\left|\Omega^{=_{\Sigma_S^I}}_{Y_1 \leftrightarrow Y_1'}\right| > 0$. Altogether $(Y_2, \mathcal{X}_1{}', l_r, l_v) \hookrightarrow_{PROCESS} (Y_1', \mathcal{X}_2')$ holds in this case.

■

The correctness of the inverse translation of an interface document follows immediately.

Theorem 6.3.9 (Correctness of the Inverse Translation): Let I be a valid invertible grammar, let $I \hookrightarrow_{INVGRAMMAR} I^{-1}$ hold and let the inverse consistency criterion be satisfied. Furthermore, let D_1 be a labeled tree. If we have $D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi_1)$ then it follows that $(D_2, D_2, \Phi_1^{-1}) \hookrightarrow_{INCTRANSLATE} (D_1', \Phi_2^{-1})$ hold with $D_1 =_{\Sigma_S^I} D_1'$. Thereby, the inverse translation calls the algorithm $\hookrightarrow_{TRANSLATE}$ with the translation context $\kappa = (I^{-1}, \Phi_1^{-1}, \alpha)$.

Proof: The algorithm $\hookrightarrow_{INCTRANSLATE}$ constructs the canonical incremental matching mapping α of D_2, the input tree of the inverse translation. From $D_1 \hookrightarrow_{TRANSLATE} (D_2, \Phi_1)$ we know that $([D_1], \mathcal{X}_1, l_0, l_{root}) \hookrightarrow_{PROCESS} ([D_2], \mathcal{X}_2)$ holds. By the correctness of the inverse rule processing shown in *Lemma 6.3.8*, it follows directly that $([D_2], \mathcal{X}_1', l_0, l_{root}) \hookrightarrow_{PROCESS} ([D_1'], \mathcal{X}_2')$ with $\left|\Omega^{=_{\Sigma_S^I}}_{[D_1] \leftrightarrow [D_1']}\right| > 0$. Hence $D_1 =_{\Sigma_S^I} D_1'$ holds and we can derive $(D_2, D_2, \Phi_1^{-1}) \hookrightarrow_{INCTRANSLATE} (D_1', \Phi_2^{-1})$.

■

Altogether, we have shown that the inverse translation terminates and that the translation round-trip is correct for valid invertible grammars under the assumption of the inverse consistency criterion.

In practice, the interface documents are continuously modified by the connected service components. Thereby, our aim is to preserve as much as possible of the original interface documents during the two-way translation by exploiting the sequence of transformation traces. Note that the contained input/output matching mappings store the information about parts of the interface documents that are not translated. In the following use case, we will discuss to which extent our approach solves this incremental problem.

6.4 Use Case

In this chapter, we will discuss an example from the mathematical domain as a practical application of the invertible grammar formalism. We will illustrate the following authoring workflow where every step is accomplished automatically by the methods of the invertible grammar formalism:

1) Translating a text-editor document into its semantic representation
2) Incrementally translating a modified version of the text-editor document into a corresponding modified semantic representation
3) Inverse translating a modified semantic representation into a modified version of the text-editor document

Before we jump into the translation process, we need to define the invertible grammar for our use case. The mathematical document will consist in our example of a theorem and a proof with some proof steps. Therefore, we will define grammar rules for the high-level structure of the document like the sentences in theorems and proofs, and for the low-level parts of the document like mathematical formulas.

Since the constraint system will not be used by the high-level grammar rules and since these rules do not use the recombination of patterns for the recursive rule invocations, it is reasonable to introduce a more compact way of representing grammar rules. Table 37 shows an example grammar in this new compact representation, in which a row represents a grammar rule. In this representation, the *head* value is the rule label, the *production* is the input pattern and the *creation* is the output pattern of a rule. Note that the input and output pattern of a rule use the same variables in the compact representation.

Head	Production	Creation
DOC	[(***TEO***|***PRF***)*]	
TEO	[\theorem{ **x:*NAME*** }[**y:*ALTC***]]	[\theorem{ **x** }[**y**]]
ALTC	["It" "holds" "that" **x:*FORMS*** "."]	[**x**]
PRF	[\proof{}[**x:*STEPS***]]	[\proof{}[**x**]]
STEPS	[((***STEP STEPS***?)|***TRIV***)]	
STEP	[(***ASS***|***FACT***)]	
TRIV	["Trivial" "."]	[\trivial[]]
ASS	["We" "assume" **x:*FORMS*** "."]	[\assumption[**x**]]
ASS	[**z** "let" **x:*FORMS*** "."]	[\assumption[**x**]]
FACT	["It" "follows" "that" **x:*FORMS*** "."]	[\fact[**x**]]
FACT	[**z** "it" "holds" "that" **x:*FORMS*** "."]	[\fact[**x**]]
NAME	[***ID***]	

Table 37. Example Grammar in Compact Representation

This compact representation allows additionally the use of EBNF notation for rules which do not have any terminals in the pattern like the start rule DOC of this example grammar. In this case, the output pattern is equal to the input pattern of the rule, except that all occurrences of pattern variables are removed. On the one hand, the compact representation provides a simple and intuitive access to the invertible grammar formalism. On the other hand, it is a shortcut for writing grammar rules which do not make use of pattern recombination for recursive rule invocations or the constraint system.

Grammar rules which are written in the compact representation are automatically transformed into the extended original representation. Non-EBNF rules are transformed by adding a rule invocation for every variable in the input pattern in DFS traversal order in pattern direction. The second rule with the label ASS is for example transformed into the following grammar rule, where the pattern variable x2 is automatically generated with the same variable specification as the pattern variable x.

ASS-2	
L	ASS
P_{IN}	[**z** "let" **x** "."]
M	[(k,[**x**],[**x2**],FORMS)]
P_{OUT}	[\assumption[**x2**]]
C_{IN}	{ }
C_{OUT}	{ }
i	{ true }

For our use case, we define the following variable specification which does not allow the pattern variable z to match "$" or ".", thus preventing this pattern variable to match a formula or a whole sentence. Furthermore, the pattern variable z is allowed to match one or more labeled trees. The other pattern variables will be mainly used by the grammar rules for the mathematical formulas, which we will introduce in the following.

```
\variables{}[
    \negative{ name="z" }[ "$", "." ],
    \negative{ name="a" }[ "$" ],
    \negative{ name="b" }[ "$" ],
    \negative{ name="t" }[ ":", "." ],
    \negative{ name="c" }[ ":" ],
    \negative{ name="n" }[ ":" ],
    \multirange{}[ "z" , "x" , "y" , "a" , "b" , "f" , "t" , "n" ]
]
```

In order to convert the EBNF rules into non-EBNF rules we apply the following standard rewrite rules until the grammar rules are stable:

1) Convert every repetition $E*$ to a fresh non-terminal X and add the grammar rules $X \twoheadrightarrow [\,]$ and $X \twoheadrightarrow [\,X\ E\,]$.
2) Convert every repetition $E+$ to a fresh non-terminal X and add the grammar rules $X \twoheadrightarrow [\,E\,]$ and $X \twoheadrightarrow [\,X\ E\,]$.
3) Convert every option $E?$ to a fresh non-terminal X and add the grammar rules $X \twoheadrightarrow [\,]$ and $X \twoheadrightarrow [\,E\,]$.
4) Convert every group (E) to a fresh non-terminal X and add the grammar rule $X \twoheadrightarrow [\,E\,]$.
5) Convert every alternative $E_1|\ldots|E_n$ to a fresh non-terminal X and add multiple grammar rules $X \twoheadrightarrow [\,E_1\,]$, …, $X \twoheadrightarrow [\,E_n\,]$.

Additionally, we apply the following standard techniques for grammar hygiene:

1) *Eliminate unproductive rules*:
   A rule is productive if it is an identity rule or if all its invoked rules are productive. We remove all rules which are not productive.
2) *Eliminate unreachable rules*:
   A rule is reachable if it is a rule with the start label of the grammar or if it is invoked by a reachable rule. We remove all rules which are not reachable.
3) *Eliminate epsilon rules*:
   An epsilon rule is a rule with empty input and empty output pattern. Let `A` be the label of an epsilon rule. Whenever `A` occurs in the rule invocations of another rule with a single variable input and output invocation pattern, then we add a rule without this rule invocation and where the pattern variables of the former rule invocation are removed from the input and output patterns of the rule if they are not used by other rule invocations. We repeat this procedure for all occurrences of epsilon rules. This process may generate new epsilon rules but it will loop to a fixpoint because there are only finitely many rules. Finally, all epsilon rules can be removed safely except epsilon rules with the start label of the grammar. Finally, this process may generate unproductive rules which we need to remove again.

At the end of this process, we completely converted an invertible grammar in compact representation into the extended original representation of invertible grammars.

Note that there may still be some unit rules in the grammar, which are rules with an input or output pattern that contains only one variable. These rules conflict with the property of valid invertible grammars that requires a smaller size input and output for rule invocations. The reason for this property is a termination guarantee. We will ignore this property in the practical examples because we assume that the grammar author checks for termination.

Having defined the grammar rules for the high-level structure of the document, we will now present the grammar rules for translating LaTeX–style mathematical formulas. Thereby, we will employ the constraint system for type checking.

The following rule translates a conjunction of two formulas. The constraints verify that both formulas have the type "*bool*".

FORMS-AND	
L	FORMS
P_{IN}	["$" **a** "$" "and" "$" **b** "$"]
M	[(k,[**a**],[**v**],TERM), (m,[**b**],[**w**],TERM)]
P_{OUT}	[\F{"and"}[**v, w**]]
C_{PRE}	{ k.type = "bool", m.type = "bool" }
C_{POST}	{ }
i	{ true }

The next rule is a unit rule which translates a single formula. We guarantee that there is no cycle between rules with the label `TERM` and rules with the label `FORMS`. The constraint verifies that the formula has the type "*bool*".

FORMS-SINGLE	
L	FORMS
P_{IN}	["$" **a** "$"]
M	[(k,[**a**],[**v**],TERM)]
P_{OUT}	[**v**]
C_{PRE}	{ k.type = "bool" }
C_{POST}	{ }
i	{ true }

The following rule translates a quantified formula by decomposition. Since the scope of the variable bounded by this quantifier is limited to the enclosed formula, this grammar rule does not inherit the environment of the recursive rule invocations. The constraints verify that the enclosed formula has the type "*bool*", and they define the type of this quantified formula to be "*bool*".

FORALL-SINGLE	
L	TERM
P_{IN}	{ dir="r" }["\forall" **c** ":" **t** "." **f**]
M	[(k,[**c** ":" **t**],[**v**],VARINTRO), (m,[**f**],[**w**],TERM)]
P_{OUT}	[\F{"forall"}[\B[**v**] , **w**]]
C_{PRE}	{ m.type = "bool", ME.type = "bool" }
C_{POST}	{ }
i	{ false }

A formula may also be quantified over multiple variables. The following rule recombines the input sequence for the recursive rule invocation. The second rule invocation may be processed for example by using the rule FORALL-MULTI or FORALL-SINGLE. The environment resulting from the recursive rule invocations is not inherited because the scope of the quantified variable is limited to the enclosed formula.

FORALL-MULTI	
L	TERM
P_{IN}	{ dir="r" }["\forall" **c** "," **n** ":" **t** "." **f**]
M	[(k,[**c** ":" **t**],[**v**],VARINTRO), (m,["\forall" **n** ":" **t** "." **f**],[**w**],TERM)]
P_{OUT}	[\F{"forall"}[\B[**v**] , **w**]]
C_{PRE}	{ m.type = "bool", ME.type = "bool" }
C_{POST}	{ }
i	{ false }

The following rule translates the intersection of sets in a term. Note that the constraints are used to verify the type "*set*" of the terms and to define the type of this intersection to be "*set*".

INTERSECT	
L	TERM
P_{IN}	[**x** "\cap" **y**]
M	[(k,[**x**],[**v**],TERM), (m,[**y**],[**w**],TERM)]
P_{OUT}	[\F{"isect"}[**v** , **w**]]
C_{PRE}	{ k.type = "set", m.type = "set", ME.type = "set" }
C_{POST}	{ }
i	{ true }

Analogously, the next rule translates the union of sets in a term.

UNION	
L	TERM
P_{IN}	[**x** "\cup" **y**]
M	[(k,[**x**],[**v**],TERM), (m,[**y**],[**w**],TERM)]
P_{OUT}	[\F{"union"}[**v** , **w**]]
C_{PRE}	{ k.type = "set", m.type = "set", ME.type = "set" }
C_{POST}	{ }
i	{ true }

The following grammar rule translates the set membership with a weak type checking by the constraints of this rule.

ELEM	
L	TERM
P_{IN}	[**x** "\in" **y**]
M	[(k,[**x**],[**v**],TERM), (m,[**y**],[**w**],TERM)]
P_{OUT}	[\F{"in"}[**v** , **w**]]
C_{PRE}	{ m.type = "set", ME.type = "bool" }
C_{POST}	{ }
i	{ true }

A single variable is translated by the following rule. This rule calls the identity translation rule `IDR` as a recursive rule invocation which returns the recognized semantic hash index value of the input variable as the value of the attribute `k.index`. Since the meta-variable `(k.index).type` can only be evaluated after the recursive rule invocations, we define the type of this variable occurrence in the set of postcondition constraints.

VAR	
L	TERM
P_{IN}	[**c**]
M	[(k,[**c**],[**v**],IDR)]
P_{OUT}	[\V{ **v** }]
C_{PRE}	{ }
C_{POST}	{ ME.type = (k.index).type }
i	{ true }

The following rule translates the introduction of a typed variable. In addition to the last rule, the type is processed by a rule invocation. The input variable is translated by calling the identity translation rule `IDI` which assigns a fresh semantic hash index value to the input variable and returns it as the value of the attribute `k.index`.

TYPED-VAR	
L	VARINTRO
P_{IN}	[**c** "**:**" **t**]
M	[(k,[**c**],[**v**],IDI), (m,[**t**],[**w**],TYPE)]
P_{OUT}	[\V{ **v** }[**w**]]
C_{PRE}	{ }
C_{POST}	{ ME.type = m.type, (k.index).type = m.type }
i	{ true }

The following rule translates the equality of terms. Thereby, the type equality of the terms is checked by the constraint `k.type = m.type`. The type of the equality is defined in the constraints to be "*bool*".

EQUAL-TERMS	
L	TERM
P_{IN}	[**x** "**=**" **y**]
M	[(k,[**x**],[**v**],TERM), (m,[**y**],[**w**],TERM)]
P_{OUT}	[\F{"equal"}[**v** , **w**]]
C_{PRE}	{ k.type = m.type, ME.type = "bool" }
C_{POST}	{ }
i	{ true }

Finally, the following rule translates the predefined type `set` and defines this also as the type in the constraints.

SET	
L	TERM
P_{IN}	["set"]
M	[ ]
P_{OUT}	[\T{"set"}]
C_{PRE}	{ ME.type = "set" }
C_{POST}	{ }
i	{ true }

With these grammar rules we have introduced a basic grammar for translating a mathematical document written in a text-editor into a semantic representation for a proof assistance system. This basic grammar is by no means complete but it satisfies the illustrative purpose of this use case.

We will now illustrate the first step of the workflow in this use case, the translation of a text-editor document into its semantic representation. The following example document in the text-editor contains a theorem about the commutativity of ∩ in the theory of simple sets. We have deliberately chosen such a simple mathematical example because we want to focus on the translation aspects instead of mathematical problem solving capabilities.

```
[ \theorem{ "Commutativity" "of" "\cap" }
    [ "It" "holds" "that" "$" "\forall" "A" "," "B" ":" "set"
      "." "A" "\cap" "B" "=" "B" "\cap" "A" "$" "." ]
]
```

The first processing steps of the algorithm $\hookrightarrow_{TRANSLATE}$ do not have any alternatives. The translation tree in Figure 13 shows the rules invoked in the parse tree. Let `#A` be the semantic hash value of the equally named variable, then the first `TYPED-VAR` rule extends the set of constraints by `#A.type = "set"`. Let `#B` be the index of the equally named variable, then the second `TYPED-VAR` rule extends the constraints by `#B.type = "set"`.

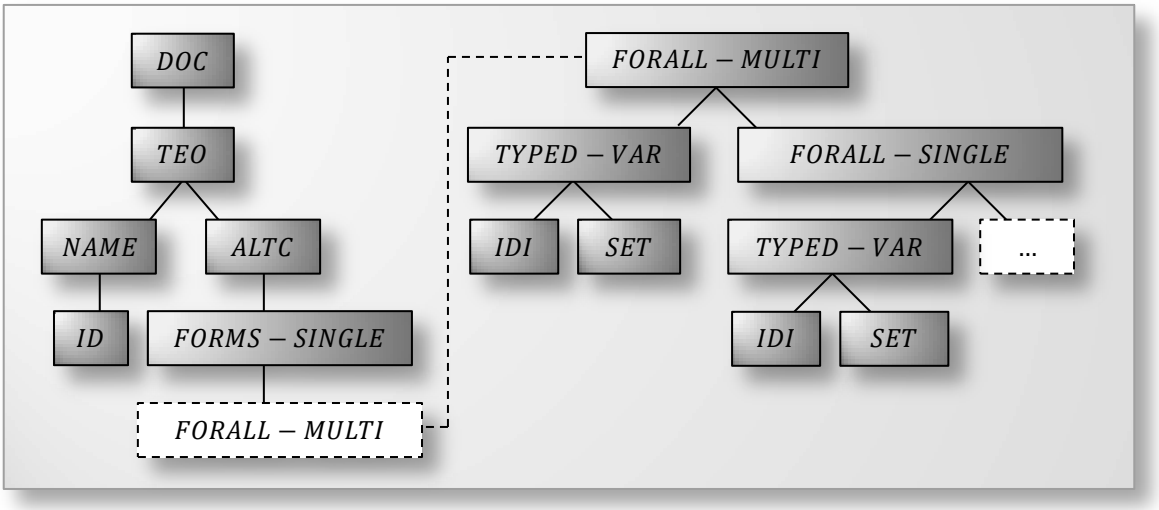


Figure 13. Translation tree (Part 1)

The translation of the quantified part of the formula, denoted by "…" in the translation tree, is a more complex process where backtracking is required. The algorithm $\hookrightarrow_{MATCH}$ first returns the grammar rule `INTERSECT` with the mapping `x->["A"]` and `y->["B" "=" "B" "\cap" "A"]`. Then, the processing of the first rule invocation succeeds. The second rule invocation matches first the grammar rule `INTERSECT` with the mapping `x->["B" "=" "B"]` and `y->["A"]`.

In turn, the first rule invocation of `INTERSECT` matches the grammar rule `EQUAL-TERMS`. This rule produces a type conflict in the evaluation of constraints with the algorithm $\hookrightarrow_{EVAL}$ because the type `"bool"` does not unify with the type `"set"`. There are no alternative mappings available. Thus, we have to backtrack two levels and instead of the rule `INTERSECT`, the rule `EQUAL-TERMS` is matched with the mapping `x->["B"]` and `y->["B" "\cap" "A"]`. But the result type `"bool"` of the rule `EQUAL-TERMS` produces a type conflict with the required type `"set"`. After some further processing steps, the backtracking reaches the top grammar rule `INTERSECT`.

An alternative mapping `x->["A" "\cap" "B" "=" "B"]` and `y->["A"]` is selected, which leads to similar type conflicts. Finally, we backtrack and the algorithm $\hookrightarrow_{MATCH}$ returns the grammar rule `EQUAL-TERMS` with the mapping `x->["A" "\cap" "B"]` and `y->["B" "\cap" "A"]`. The translation process continues without any conflicts in the constraint evaluation and produces the translation subtree shown in Figure 14. Note that the formula variables are recognized by the rule `IDR` and the type of these variable occurrences is set in relation to the introduced type of this variable via the constraint system.

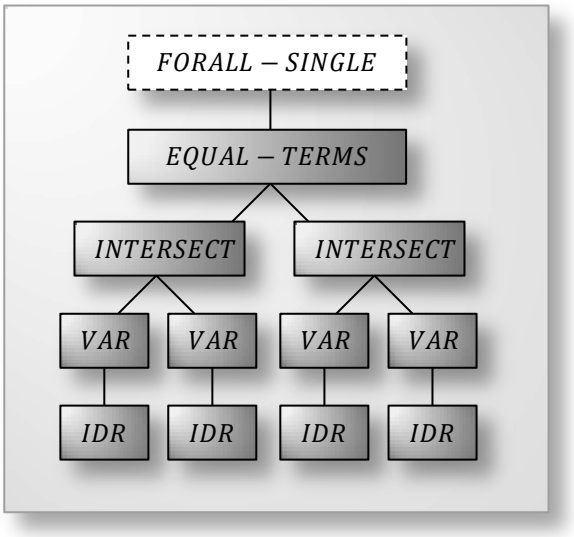


Figure 14. Translation tree (Part 2)

The result of the translation process is the following semantic representation.

```
[ \theorem{ "Commutativity" "of" "\cap" }
    [ \F{"forall"}[ \B[ \V{"A"}[\T{"set"}] ],
       \F{"forall"}[ \B[ \V{"B"}[\T{"set"}] ],
        \F{"equal"}[ \F{"isect"}[ \V{"A"}, \V{"B"} ],
                     \F{"isect"}[ \V{"B"}, \V{"A"} ]]]]
]
```

The second part of the workflow, which we want to illustrate, consists of incrementally translating a modified version of the document in the text-editor into a corresponding modified version of the semantic representation. We modify the example document by adding a partial proof for the theorem as follows.

```
[ \theorem{ "Commutativity" "of" "\cap" }
    [ "It" "holds" "that" "$" "\forall" "A" "," "B" ":" "set"
      "." "A" "\cap" "B" "=" "B" "\cap" "A" "$" "." ],
  \proof{}
    [ "First" "," "let" "$" "x" "\in" "A" "\cap" "B" "$" "."
      "It" "follows" "that" "$" "x" "\in" "A" "$" "." ]
]
```

The algorithm $\hookrightarrow_{INCTRANSLATE}$ first computes the incremental matching mapping between the new and the old document using the algorithm $\hookrightarrow_{INCMAP}$. The computed mapping maps all preserved subtrees to their canonical matching partners. Since the proof subtree is inserted, it does not have any matching partner in the incremental matching mapping. Therefore, the incremental translation process can reuse those parts of the translation tree which are related to the translation of the theorem subtree using the information stored in the transformation traces.

The new proof subtree is then processed by the rule `PRF`. The first proof step matches the grammar rule `ASS-2` with the mapping `z->["First" ","]` and `x->["$" "x" "\in" "A" "\cap" "B" "$"]`. Note that the pattern variable `z` is not used by the sub-rules of `ASS-2` but the information about the matching partners is stored in the transformation trace of that rule application. Then the rule `FORMS-SINGLE` is invoked with the mapping `a->["x" "\in" "A" "\cap" "B"]`.

In the recursive rule invocation of `FORMS-SINGLE`, the algorithm $\hookrightarrow_{MATCH}$ first returns the grammar rule `INTERSECT` with the mapping `x->["x" "\in" "A"]` and `y->["B"]`. The processing of the first rule invocation of `INTERSECT` fails because of a type conflict with the only compatible grammar rule `ELEM`. The backtracking mechanism then selects the grammar rule `ELEM` instead of `INTERSECT` which leads to a valid translation tree.

After completing the processing of this branch of the translation tree, the second proof step matches the rule `FACT-1` with the input matching mapping `x->["$" "x" "\in" "A" "$"]`. Then the rule `FORMS-SINGLE` is invoked with the input mapping `a->["x" "\in" "A"]`. which in turn invokes the grammar rule `ELEM`. Finally, the translation process succeeds with the incremental translation tree shown in Figure 15 that extends the original translation tree.

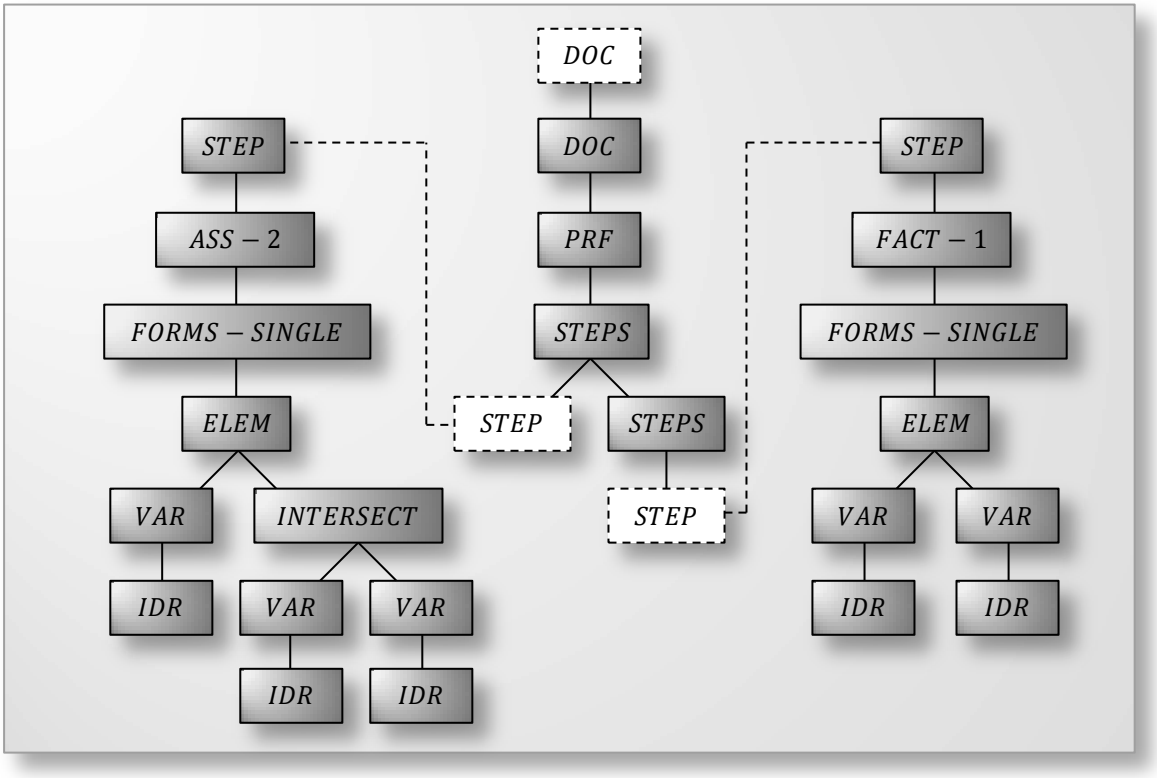


Figure 15. Incremental translation tree

The result of the incremental translation process is the following semantic representation.

```
[ \theorem{ "Commutativity" "of" "\cap" }
    [ \F{"forall"}[ \B[ \V{"A"}[\T{"set"}] ],
       \F{"forall"}[ \B[ \V{"B"}[\T{"set"}] ],
        \F{"equal"}[ \F{"isect"}[ \V{"A"}, \V{"B"} ],
                     \F{"isect"}[ \V{"B"}, \V{"A"} ]]]],
  \proof{}
    [ \assumption
       [ \F{"in"}[ \V{"x"}, \F{"isect"}[ \V{"A"}, \V{"B"} ]]],
      \fact
       [ \F{"in"}[ \V{"x"}, \V{"A"} ]]]
]
```

Note that the scope of the formula variables `A` and `B` , which have been introduced in the theorem, ends formally with the scope of the binding quantifier in the theorem. Thus, the semantic hash values returned for the variable occurrences in the proof are different. A possible solution could be to define a special binding sentence like "*Let* $A, B : set$." As part of the theorem content, whose scope ends with the proof of this theorem.

The third and final part of the workflow in this use case consists of the inverse translation of a modified version of the semantic representation into a modified version of the text-editor document. In this example, the proof assistance system has modified the semantic representation by replacing the formula in the second proof step, the fact subtree, with a formula which contains more derived information. The modified semantic representation is shown in the following.

```
[ \theorem{ "Commutativity" "of" "\cap" }
    [ \F{"forall"}[ \B[ \V{"A"}[\T{"set"}] ],
       \F{"forall"}[ \B[ \V{"B"}[\T{"set"}] ],
        \F{"equal"}[ \F{"isect"}[ \V{"A"}, \V{"B"} ],
                     \F{"isect"}[ \V{"B"}, \V{"A"} ]]]],
  \proof{}
    [ \assumption
       [ \F{"in"}[ \V{"x"}, \F{"isect"}[ \V{"A"}, \V{"B"} ]]],
      \fact
       [ \F{"and"}[ \F{"in"}[ \V{"x"}, \V{"A"} ],
                    \F{"in"}[ \V{"x"}, \V{"B"} ]]]
]
```

For the inverse translation, the grammar is first inverted using the algorithm $\hookrightarrow_{INVGRAMMAR}$ by swapping the input and output pattern of the rule and its recursive rule invocations. Furthermore, the sequence of transformation traces is inverted using the algorithm $\hookrightarrow_{INVTRACES}$ by swapping the stored input and output pattern and matching mappings. The inverse translation is then performed by the algorithm $\hookrightarrow_{INCTRANSLATE}$.

First, the incremental matching mapping between the new and the old semantic representation is computed using the algorithm $\hookrightarrow_{INCMAP}$. The computed mapping maps all preserved subtrees to their canonical matching partners. In this case, this applies to all subtrees except of the subtrees of the formula in the fact proof step. Therefore, the inverse translation process can reuse all parts of the translation tree which are related to the translation of any subtree except of the fact subtree using the information stored in the transformation traces.

Note in particular that the translation of the first proof step with the grammar rule `ASS-2` has access to the stored output matching mapping `z->["First" ","]` and `x->["$" "x" "\in" "A" "\cap" "B" "$"]` in the corresponding inverted transformation trace. Therefore, the original input can be reconstructed by the inverse translation process.

The only part of the semantic representation, for which there is no matching transformation trace available, is the modified fact subtree. Hence, the inverse translation has to process this part without hints. The algorithm $\hookrightarrow_{MATCH}$ returns the grammar rule `FACT-1` with the matching mapping `x2->[\F{"AND"}[\F{"IN"}[\V{"x"}, \V{"A"} ], \F{"IN"}[ \V{"x"}, \V{"B"}]]]`. Then, the processing of the first rule invocation matches the grammar rule `FORMS-AND` with the input matching mapping `v->[\F{"IN"}[\V{"x"},\V{"A"}]] , w->[\F{"IN"}[\V{"x"},\V{"B"}]]]`. Furthermore, both recursive rule invocations are successfully processed with the matching grammar rule `ELEM`. Thereby, the type of the variables `A` and `B` is checked by the constraint system. Finally, the inverse translation process succeeds with the incremental inverse translation tree shown in Figure 16 that replaces the original translation tree at the position corresponding to the second proof step.

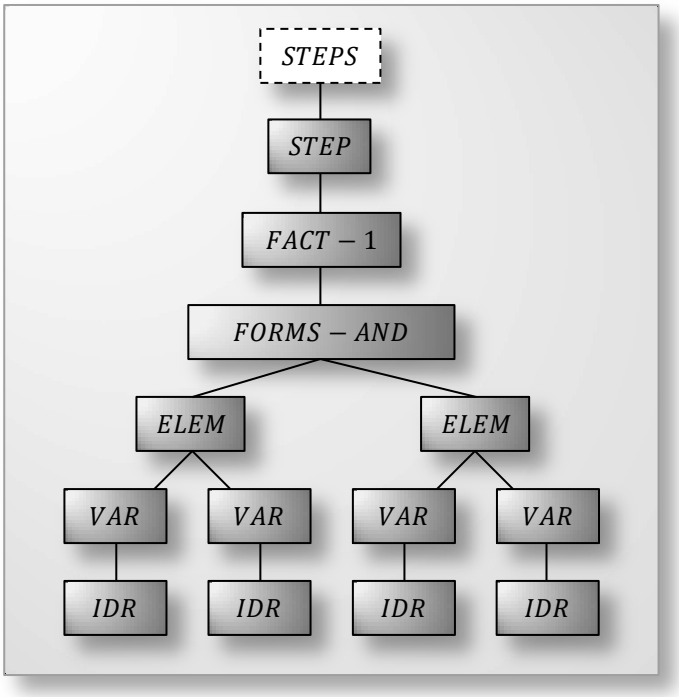


Figure 16. Inverse translation tree

The result of the inverse translation process is the following document in the text-editor.

```
[ \theorem{ "Commutativity" "of" "\cap" }
    [ "It" "holds" "that" "$" "\forall" "A" "," "B" ":" "set"
      "." "A" "\cap" "B" "=" "B" "\cap" "A" "$" "." ],
  \proof{}
    [ "First" "," "let" "$" "x" "\in" "A" "\cap" "B" "$" "."
      "It" "follows" "that" "$" "x" "\in" "A" "$"
      "and" "$" "x" "\in" "B" "$" "." ] ]
```

In this use case, we illustrated how the invertible grammar formalism can be employed to incrementally translate between two interface documents in both directions. We want to emphasize the main benefits of the invertible grammar formalism in a short summary:

1) *Workflow Automation*
   Based on the invertible grammar formalism, we presented methods for the complete automation of the translation of interface documents, the incremental translation, and the inverse translation.
2) *Hints for the Incremental Translation*
   The sequence of transformation traces which is stored during the translation serves as hints for the incremental translation. By exploiting the change graph between the old and the new version of an interface document, we compute the set of preserved subtrees. This information is then used to identify a reusable transformation trace which guides the incremental translation process.
3) *Oracle for the Inversion*
   Since the transformation trace stores the matching partners of the variables in the input and output pattern of all invoked grammar rules, we can exploit this information in the two-way translation process. If the conditions for reusing a transformation trace are satisfied, we can complete the computed partial variable mappings by the stored information. Thus we use the transformation trace as an oracle for generating parts of the interface document that are lost during translation. Since the input of the current rule and the input of the candidate transformation trace has to be semantically equal with respect to the preserved subtrees, we suggest wherever possible to model grammar rules for the content of tree layers by the recursive form $A \twoheadrightarrow BA$. This increases the possibility for reusing a transformation trace in the invocation of grammar rules for B.
4) *Type Checking and Static Scoping*
   We have demonstrated how the constraint system can be used for the weak type checking of mathematical formulas. Furthermore, we illustrated how the combination of a semantic hash indexing function, meta-variables and a managed environment allows for the modeling of static scoping.
5) *Support for Alternatives*
   In this use case, we have not exploited all possibilities of the similarity specification with respect to the pattern matching of the grammar rule and input alternatives. These features are more suitable for the two-way translation between two different semantic representations.

6.5 Discussion

In this chapter, we developed an *invertible grammar formalism* by combining concepts from the parser-oriented *attribute grammars* and from the generator-oriented *TGL*. The main motivation for the development of a new grammar formalism was the need for adequate methods to transform between interface documents in both directions. Thereby, we focused on the one hand on the complete automation of the inversion process, and on the other hand on a technique which allows for preserving as much as possible of the non-translated content during the continuous transformation round-trip.

Our contributions are a method for the automation of the grammar inversion, and an incremental interpreter which can be used with both, a grammar and its inverted grammar. By integrating the notion of semantic equality into the pattern matching method of the grammar rule processing, we essentially transform an interface document modulo a commutativity theory. In comparison to the two-way translators proposed by [Yellin & Mueckstein, 1985], our invertible grammar formalism allows for more than just the permutation of content. We support the recombination of parts of the input in a two-step processing style with grammar rules and recursive rule invocations.

Besides that, there exists a subclass of attribute grammars called *ordered attribute grammars* [Kastens, 1980] with the property that for all grammar rules a partial order of the constraints can be given such that - in any context - the constraints can be evaluated in an order which includes that partial order. Since we use unification constraints as attributes, the order of attribute evaluation is not as important as for a standard attribute grammar. Indeed, we presented an incremental method for constraint evaluation. Nevertheless, the specific processing order of recursively invoked rules is declaratively specified in every rule of an invertible grammar.

With the invertible grammar formalism we presented a method to translate the change script for one interface document into a change script for its connected interface document. The translation of change scripts is a very general problem with many application areas. The approach of [Greenwald *et al*, 2003] proposes for example a language for primitive tree manipulation to describe the translations, called *lenses*. This approach assumes that one tree is an abstraction of the other tree. In our scenario, both interface documents may contain detailed information which is not available in the other one. We cannot require semantic tree inclusion. Another approach [Abiteboul *et al*, 2002] introduces a declarative language for describing correspondences between parts of the trees in a data forest. These correspondence rules can be used to translate one tree format into another one through non-deterministic computation. However, this approach assumes an isomorphism between the trees, which is not a realistic requirement in our scenario.

The approach we presented in this chapter uses the concept of a *transformation trace* to store intermediate results of the translation process. The inverse translation process then exploits the sequence of transformation traces in order to generate those parts of the original input interface document which have not been translated. The important property for the correctness of the inverse translation is the *inverse consistency criterion* that requires that there is at most one matching transformation trace when the rule matching method exploits the sequence of previous transformation traces in both translation directions. We presented dynamic and static methods for checking whether the inverse consistency criterion is satisfied.

Additional transformation aspects are methods for identifying and correcting errors. One usually distinguishes between simple recovery techniques which use primitive edit operations, phrase level recovery techniques which discard or replace sequences of tokens, and scope recovery techniques which insert for example closing fragments. For example, for diagnosing the source of errors in the evaluation of constraints there exists the method of inverse currying transformation for attribute grammars [Wilhelm, 1984]. In our scenario with unification constraints, the method of source-tracking unification [Choppella & Haynes, 2005] would be more suitable. Furthermore, one can integrate error correction methods that use the primitive edit operations to add, delete or replace words [Mellish, 1989] or skip words for robustness [Lavie & Tomita, 1993].

In the context of bidirectional transformations, the *problem of logical-form equivalence* [Shieber, 1993] is always mentioned. Given a *logical form*, that is, a semantic representation, a translation process must then generate a string with that meaning, that is, a string whose canonical logical form *means the same* as the given one.

String	Canonical Logical Form
$\boldsymbol{A \cup B \cap C}$	\F{"union"}[\V{"A"},\F{"isect"}[\V{"B"},\V{"C"}]]
$\boldsymbol{A \cup (B \cap C)}$	\F{"union"}[\V{"A"},\F{"isect"}[\V{"B"},\V{"C"}]]
$\boldsymbol{(A \cup B) \cap C}$	\F{"isect"}[\F{"union"}[\V{"A"},\V{"B"}],\V{"C"}]

Table 38. Examples of the Logical-Form Equivalence Problem

An example for this problem in our scenario is shown in Table 38. The inverse translation of the third logical form should not generate the formula $A \cup B \cap C$ because its canonical logical form does not mean the same as the original logical form. In practice, meaning equivalence can be approximated by logical equivalence. We can solve the logical-form equivalence problem for this example by translating the arguments of ∪ and ∩ with an additional rule category whose default rule generates brackets around the term if the term is not a single variable. Nevertheless, the logical-form equivalence problem includes additionally the problem that there does not exist a formal definition of meaning equivalence and that it is widely assumed that meaning equivalence is not computable.

III
Mathematical Authoring Assistance

In this part, we illustrate a practical application of the *Change-Oriented Architecture* to *Mathematical Authoring Assistance* using the proof assistance system ΩMEGA. In a mathematical course scenario, we will use the concepts and methods of the COA for the integration of the verification services of ΩMEGA with text-editors to provide assistance during the authoring process. In this setting, we will present solutions for the following authoring assistance cases in the course scenario.

1) How can we assist the lecturer with the authoring of lecture notes?
2) How can we assist the student with the authoring of exercise solutions?

To use the verification services of ΩMEGA, the content of a mathematical document in a text-editor has to be transformed into the semantic representation required by ΩMEGA. The results of verification services, like for example corrections or completions, need to be inversely transformed into modifications of the document in the text-editor. For this purpose, we use the invertible grammar formalism. Furthermore, the semantic change scripts are computed with respect to the document-specific weights and processing requirements of ΩMEGA. The solutions to the authoring cases share a common ground of invertible transformation grammars, similarity and edit specifications. The presentation of the solutions is organized as follows.

Before we start to address the authoring assistance cases, we will introduce the proof assistance system ΩMEGA, its semantic representation of the mathematical document and its verification services in Chapter 7. Furthermore, we will analyze the requirements of *Mathematical Authoring Assistance* with an exploratory study conducted with the students of a mathematics course.

In Chapter 8 we will address the first authoring assistance case for lecture notes. Besides introducing the transformation pipeline between text-editor and proof assistance system, the focus of this chapter is on the dynamic and extensible support of notation. We will present a method that allows for exploiting the notation defined by the author in lecture notes in order to automate the formalization of the formulas in a mathematical document, and to adapt these formulas on notational changes.

The second authoring assistance case for exercise solutions in Chapter 9 sets the focus on mathematical proofs. We will illustrate the process of incrementally creating proof obligations for ΩMEGA from the proof steps in the exercise solutions written in a controlled mathematical language. Furthermore, we will demonstrate how the transformation pipeline can be used to propagate feedback for proof steps and to interact with the verification services of the proof assistance system ΩMEGA within the text-editors TEXMACS and MS WORD.

7 Application Scenario

The vision of a powerful mathematical assistance environment, providing computer-based support for most tasks of a mathematician, has stimulated the development of the proof assistance system ΩMEGA. A mathematical assistance system that really supports mathematicians in their daily work has to be highly user oriented. The mathematician is used to formulate his problems with pen and paper, and we want to extend this traditional workflow by providing authoring assistance for encoding the solution of a problem and refining the details with a text-editor like the TEXMACS system [van der Hoeven, 2001].

Our aim is to build document-centric services that support the author in the text-editor while preparing a document in a publishable format. In order to use the services of the proof assistance system ΩMEGA, the content of a mathematical document in a text-editor has first to be transformed into the semantic representation required by ΩMEGA. To analyze the complexity of the linguistic aspects of this transformation, we conducted an exploratory study in the mathematical domain. Although the goal of 90% coverage of common mathematical documents is clearly out of scope for this thesis, the study indicates the relative importance of some linguistic features with respect to the coverage. Therefore, we collected and analyzed in an exploratory study a *corpus* of sample documents.

The real-time behavior of a proof assistance system in the specific domain is very important because we need a quick response to user input. Regarding the capabilities of state-of-the-art proof assistance systems like ISABELLE, COQ or ΩMEGA, it seems more realistic to provide the authoring assistance to a first year mathematics student rather than to a mathematician working on the cutting edge of research. Therefore we decided to conduct the exploratory study with the first year mathematics course *Mathematik für Informatiker* by Prof. John at Saarland University.

This chapter is organized as follows: First, we introduce the proof assistance system ΩMEGA, its semantic representation of a mathematical document and its verification services. Then, the course scenario will be described together with our main research questions for authoring assistance. Furthermore, the architecture of the exploratory study is described, which integrates TEXMACS and an exercise manager via a mediation module. After that, we give a detailed report on a corpus analysis with respect to the authoring behavior and selected linguistic aspects. Finally, we discuss potential added values before we summarize the results.

7.1 Introduction to ΩMEGA

The long-term goal of the ΩMEGA system ([Siekmann *et al*, 2006], [Benzmüller *et al*, 2006]) is the development of a large, integrated assistance system supporting different mathematical tasks and a wide range of typical research, publication and knowledge management activities. Examples of such tasks and activities are proving, verifying, explaining, and many others. Mathematical assistance systems are knowledge-based systems in which different kinds of knowledge must be maintained. Among others, this includes

1) *theory knowledge* like axioms and theorems organized in structured theories,
2) *procedural knowledge* like a symbol ordering or proof planning methods, and
3) *notational knowledge* used in the user interfaces of the system.

In our application scenario for *Mathematical Authoring Assistance*, we provide a part of the theory knowledge via the semantic representation of the mathematical content of the document in the text-editor. The ΩMEGA system then synthesizes additional procedural knowledge from the provided theory knowledge and tries to verify proof sketches by constructing a valid proof plan using proof search techniques. Thereby, the notational knowledge is not immediately relevant for the proof verification service.

In the following, we will first describe ΩMEGA*'s* main components as shown in Figure 17. Then, we describe the semantic document representation required by the text-editor service interface of ΩMEGA*'s* service mediator PLATΩ. Furthermore, we give an overview of the mathematical assistance services offered by ΩMEGA via PLATΩ.

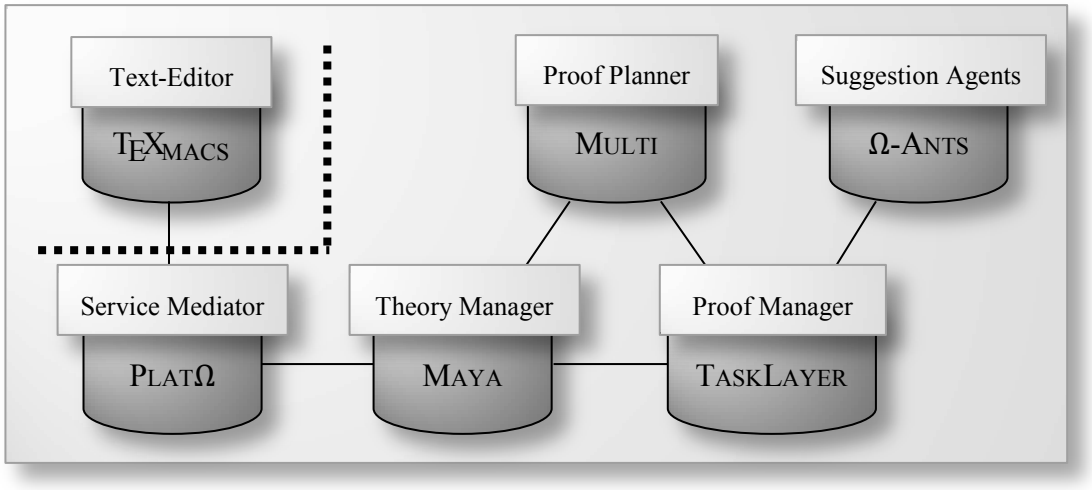


Figure 17. Components of the ΩMEGA *system*

MAYA – The Theory Manager. Theory knowledge such as axioms, definitions, lemmas, and theorems – collectively called *assertions* – are organized in structured theories and maintained in MAYA [Autexier & Hutter, 2005]. These theories are built on top of each other by importing knowledge from lower theories via theory morphisms. MAYA internally organizes the theories based on the notion of *development graphs* ([Hutter, 2000], [Mossakowski *et al*, 2006]). Development graphs are used by MAYA to dynamically control which knowledge is available for which proof.

MAYA maintains the information on how global proof obligations are decomposed into local proof obligations and which axioms, lemmas and theorems have been used in which proofs. These dependency relations are exploited to provide an efficient support for change operations which can modify the graph structure of the theories as well as their content.

TASKLAYER – The Proof Manager. The TASKLAYER [Dietrich, 2006] provides the primary methods to represent, manipulate and maintain proofs in the ΩMEGA system. Its main tasks are the construction of proofs and the maintenance of the current states of proof attempts with their open goals.

The basic proof construction operator in the TASKLAYER is called *inference*. Inferences are either operational representations of the axioms, lemmas and theorems provided by MAYA, or they encode domain or problem specific mathematical methods, possibly including the invocation of specialized external systems.

The states of proof attempts maintained by the TASKLAYER contain so-called *tasks*, which are Gentzen-style multi-conclusion sequents [Gentzen, 1969], augmented by means to define multiple foci of attention on subformulas. The goal of the proof construction is to recursively reduce each task to a possibly empty set of subtasks by one of the following proof construction steps:

1) the introduction of a proof sketch ([Autexier *et al*, 2004], [Wiedijk, 2003]),
2) deep structural rules for weakening and decomposition of subformulas,
3) the application of a postulated lemma [Autexier & Dietrich, 2006],
4) the substitution of meta-variables, and
5) the application of an inference.

The operationalization of mathematical knowledge into inferences paired with the possibility to apply inferences to subformulas results in natural, human-oriented proofs where each step is justified by an *assertion*, such as a definition, axiom, theorem or lemma.

MULTI – The Proof Planner. The multi-strategy proof planner MULTI [Melis *et al*, 2008] is used by the TASKLAYER to perform a heuristically guided search using the proof strategies which it receives from MAYA. A proof strategy describes a set of inferences and control rules. Thereby, the control rules define how to proceed at choice points in the proof planning process, for example they define the order in which inferences are applied to tackle the next subgoal. Thus, the search space is restricted by the control rules, certain search paths are preferred and others are pruned. MULTI can flexibly interleave different strategies in a proof attempt.

Inferences can also encode some abstract-level proof ideas rather than low-level calculus rules and thus the proof plans delivered by MULTI may fail. However, abstract proof plans can be recursively expanded to a logic level proof within a verifiable calculus. If this expansion succeeds, a valid proof plan and a corresponding checkable proof on the calculus level have been found. If the expansion fails, the proof plan remains invalidated.

Ω-ANTS – The Suggestion Agents. The Ω-ANTS component ([Benzmüller & Sorge, 1998], [Benzmüller & Sorge, 2000]) supports interactive proof construction by generating a ranked list of bids of potentially applicable inferences in each proof state. In this process, all inferences are uniformly viewed with respect to their arguments, that is, their premises, conclusions, and additional parameters. An inference rule is applicable if a sufficiently complete instantiation for its arguments has been computed. Hence, the goal of Ω-ANTS is to determine the applicability of inference rules by incrementally computing argument instantiations. These applicability checks are performed by separate processes, that is, software agents which compute and report bids.

PLATΩ – The Service Mediator. The goal of the PLATΩ component [Wagner *et al*, 2006] is to make ΩMEGA's functionalities available in different application scenarios. One application scenario is to support the writing of scientific publications in the WYSIWYG text-editor TEXMACS [van der Hoeven, 2001] with the ΩMEGA system running in the background. Thereby, PLATΩ expects the mathematical content being provided by the text-editor in a semantic representation. The PLATΩ component then establishes and guarantees the consistency between the semantic representation at its interface and the theory knowledge and global state of the ΩMEGA system. Finally, PLATΩ offers automated verification, completion and explanation services for proofs and proof steps provided by ΩMEGA.

In the following, we will introduce the language for the semantic representation of a mathematical document required by the services of the PLATΩ component. This language has been primarily designed as an annotation language for mathematical documents and consists of different modules for theories, proofs, references, definitions and formulas. The development of this semantic representation language has been inspired by the standard for semantic mathematical documents OMDOC [Kohlhase, 2000] and the standard for semantic mathematical formulas OPENMATH [Davenport, 2000].

The reason for developing the semantic representation language of PLATΩ has been twofold: On the one hand, we needed a more compact representation with a minimal syntactic processing overhead for real-time usage. On the other hand, the model of proof steps in OMDOC did not fit well to the model used by the ΩMEGA system. Nevertheless, you will notice that the following semantic representation language shares the basic aspects of the above standards but with less verbosity.

We will present the XML grammar for the semantic representation language in EBNF notation, partitioned into different sublanguages. Note that every non-terminal in the grammar has to be treated as an XML element. Furthermore, the symbol `text` is a terminal which matches an arbitrary string. Curly brackets in the content indicate that the elements can be ordered arbitrarily.

Element	Args	Content
DOCUMENT		`THEORY*`
THEORY	`NAME?`	`(CONTEXT|DEFINITION|AXIOM|THEOREM|PROOF|text)*`
CONTEXT		`(REFERENCE|text)*`
AXIOM	`NAME?`	`{PRECONDITION*;CONCLUSION?;text*}`
THEOREM	`NAME?`	`{PRECONDITION*;CONCLUSION?;text*}`
PRECONDITION		`{FORMULA;text*}`
CONCLUSION		`{FORMULA;text*}`

Table 39. Semantic Theory Language

Table 39 shows the grammar for representing a mathematical theory with definitions, axioms, theorems and proofs. In the spirit of the notion of theory inheritance in development graphs, the context of a theory allows for including the knowledge of other existing theories in the document, but in this case without a theory morphism. Note that the terminal `text` occurs in the content of almost all elements. This allows us to use this language as a document annotation language.

Mathematical proofs can be annotated by the language shown in Table 40. This grammar contains multiple semantically different annotations for proof steps. We will discuss the semantics of these different types of proof steps after the presentation of the remaining sublanguages.

Element	Args	Content
PROOF	FOR	(SET|FACT|DECOMPOSE|GOAL|SUBGOALS|ASSUMPTION| CASES|COMPLEX|text)*;TRIVIAL?
SET		{FORMULA}
FACT		{FORMULA+;BY*;FROM*;text*}
DECOMPOSE		{ASSUME;OBTAIN;BY*;FROM*;text*}
GOAL		{FORMULA;BY*;FROM*;text*}
SUBGOALS		{FORMULA+;PROOF*;BY*;FROM*;text*}
ASSUMPTION		{FORMULA;BY*;FROM*;text*}
CASES		{FORMULA+;PROOF*;BY*;FROM*;text*}
COMPLEX		{COMP+;PROOF*;BY*;FROM*;text*}
TRIVIAL		(BY|FROM|text)*
BY		(REFERENCE|text)*
FROM		(REFERENCE|text)*
ASSUME		{FORMULA+;text*}
OBTAIN		{FORMULA+;text*}
COMP		{ASSUME;OBTAIN;text*}

Table 40. Semantic Proof Language

The grammar for references is shown in Table 41. We distinguish between local references (L) which point to elements inside of the same theory, and global references (R) which point to elements in another specific theory.

Element	Args	Content
REFERENCE		(R|L)
R	NAME?	THEORY
L	NAME	

Table 41. Semantic Reference Language

A definition may introduce either a new symbol or a new type. The grammar for definitions is shown in Table 42. The only predefined name of a simple type (T) is the name of the Boolean type `bool`. To specify the type of a new symbol, one may also use other types defined in the actual theory or in the transitive closure of its context. Functional types can be specified either in the Cartesian style (TX) or in the curried style (TF). Predefined symbol names are *false*, *true*, =, *impl*, *or*, *eqv*, *and*, *not*, *pi*.

$a_1 \times a_2 \times \ldots \times a_n$ is encoded as `\TX[ a1 a2 ... an ]`

$a_1 \to a_2 \to \cdots \to a_n$ is encoded as `\TF[ a1 a2 ... an ]`

To give an example, a new symbol *union* with type $set \times set \to bool$ is encoded as

```
\S{"union"}[
   \TF[\TX[\T{"set"},\T{"set"}],
       \T{"bool"}]]
```

Element	Args	Content
DEFINITION	NAME?	text*;(SYMBOL|TYPE);text*
SYMBOL	NAME	(T|TF|TX)
TYPE	NAME	
T	NAME, THEORY?	
TF		(T|TF|TX);(T|TF|TX)+
TX		(T|TF|TX);(T|TF|TX)+

Table 42. Semantic Definition Language

Finally, the sublanguage for formulas is shown in Table 43. A formula is either a function application (F), a variable (V), or a symbol (S). Predefined function names are *forall*, *exists*, *lambda*, =, *eqv*, *impl*, *not*, *and*, *or*. The bounded variables of quantifiers are grouped by (B). One may also use other symbols or functions defined in the actual theory or in any theory in the transitive closure of the context. In case one uses a symbol or function which is defined multiple times, the theory has to be indicated in order to resolve the ambiguity.

Element	Args	Content
FORMULA	NAME?	(F|V|S)
F	NAME, THEORY?	B?;(F|V|S)*
B		V+
V	NAME	(T|TF|TX)?
S	NAME, THEORY?	
T	NAME, THEORY?	
TF		(T|TF|TX);(T|TF|TX)+
TX		(T|TF|TX);(T|TF|TX)+

Table 43. Semantic Formula Language

A new variable x with the type $bool$ is encoded as `\V{"x"}[\T{"bool"}]`. Some other examples of formulas with their encodings are given in Table 44.

$\forall x.x \in A \Rightarrow x \in B$	$x \in A \cap (B \cup C)$	$f(x) = x^2$

`\F{"forall"}[`
`\B[`
`\V{"x"}],`
`\F{"impl"}[`
`\F{"in"}[`
`\V{"x"},`
`\V{"A"}],`
`\F{"in"}[`
`\V{"x"},`
`\V{"B"}]]]`
	`\F{"in"}[`
	`\V{"x"},`
	`\F{"intersect"}[`
	`\V{"A"},`
	`\F{"union"}[`
	`\V{"B"},`
	`\V{"C"}]]]`
		`\F{"f"}[`
		`\V{"x"},`
		`\F{"square"}[`
		`\V{"x"}]]`

Table 44. Examples for Semantic Representations of Formulas

We will now discuss the semantics of the different types of proof steps supported by ΩMEGA. This will allow us to describe how proofs are processed by the proof assistance system. In the ΩMEGA system, proofs are constructed by the TASKLAYER that uses an instance of the generic *proof data structure* ($\mathcal{PDS}$) [Autexier *et al*, 2005] to represent proofs.

Task. At the TASKLAYER, the main entity is a *task* T, a Gentzen-style multi-conclusion sequent $F_1, \ldots, F_i \vdash G_1, \ldots, G_k$. Each formula in a task can be named by assigning a label l to the formula.

Agenda. A proof attempt is represented by an *agenda*. It maintains a set of tasks, which are the subproblems to be solved, and a global substitution which instantiates meta-variables. Formally, an agenda is a pair $\langle T_1, \ldots, T_{i-1}, \boldsymbol{T_i}, T_{i+1}, \ldots, T_n; \sigma \rangle$ where $T_1, \ldots, T_n$ are tasks, σ is a substitution, and $\boldsymbol{T_i}$ is the task the system is currently working on.

In the following, we describe informally for each member of our semantic proof representation language how this type of proof step is processed by ΩMEGA. The detailed formal modeling of these methods is described by Dominik Dietrich in [Dietrich, 2006], [Dietrich & Buckley, 2007] and [Dietrich *et al*, 2008].

Fact. The command *fact* derives a new formula φ with label l from the current proof context. ΩMEGA tries to justify the new fact by the application of the inference *name* specified in *by* to term positions in the formulas with labels $l_1, \ldots, l_n$ in the current task, specified in *from*. If the information in *by* and *from* is underspecified, all inferences are matched against all admissible term positions, and the first one which delivers the desired formula φ is applied. If the above check fails, ΩMEGA generates a new lemma with a new proof tree, containing the assumptions of the current task and the newly stated fact φ as the goal (see Figure 18). It then tries to automatically close the lemma.

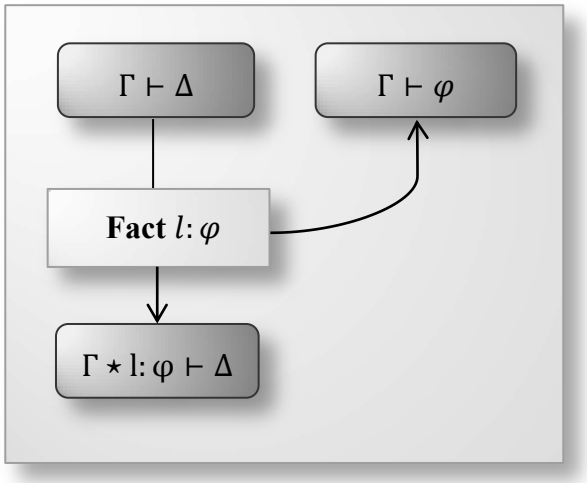


Figure 18. Fact Repair

Goal. The command *goal* reduces the current set of goals to a new formula φ with label l. ΩMEGA tries to justify the new goal by the application of the inference *name* in *by* to term positions in *from* in the formulas with labels $l_1, \ldots, l_n$ in the current task. In the case that *by* and *from* are underspecified, all inferences are matched against all admissible term positions, and the first one which delivers the desired formula φ is applied. If the above check fails, ΩMEGA generates a new lemma with a new proof tree, containing the assumptions and the goals of the current task and the newly stated goal φ as an assumption (see Figure 19). It then tries to automatically close the lemma and continues analogously to *fact*.

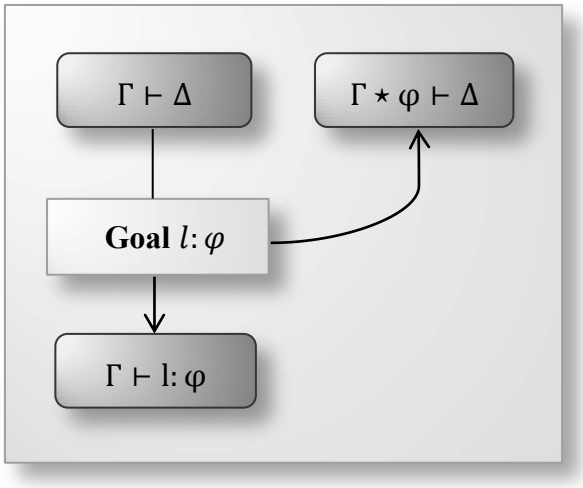


Figure 19. Goal Repair

Assumption. The command *assumption* introduces a new assumption φ on the left hand side of the current task. ΩMEGA checks whether one of the following situations occurs, each of which can be justified by an inference application:

- Δ contains $\varphi \Rightarrow \psi$. The implication is decomposed, l: φ is added to the left side of the task and ψ remains in right side of the task (see Figure 20).
- Δ contains $\neg\varphi$. Then l: φ is added to the left side.
- Δ contains $\psi \Rightarrow \neg\varphi$. Then l: φ is added to the left side and $\neg\psi$ to the right side of the task.

If this check fails, ΩMEGA tries to derive one of the above situations by applying inferences to the goal of the current task. Thereby, the hypotheses of the task remain untouched.

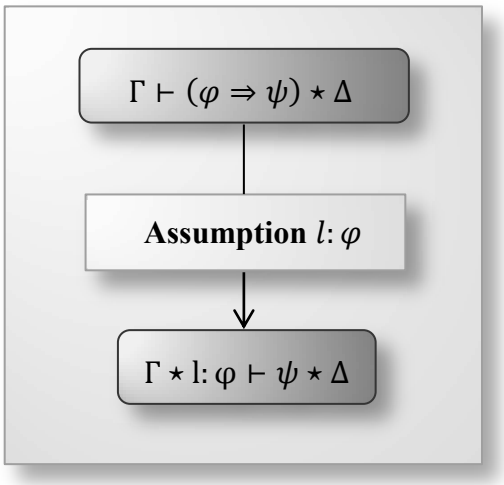


Figure 20. Assumption Check

Decompose. The command *decompose* introduces a new assumption φ on the left hand side of the current task and a new goal ψ on the right hand side. The check for *decompose* is a special case of the check for *assumption*. In fact, ΩMEGA checks whether Δ contains $\varphi \Rightarrow \psi$, which is the first case of the command *assumption*. If this check fails, ΩMEGA tries to derive this situation by applying inferences to the goal of the current task, analogously to the case for *assumption*.

Set. The command *set* is used to bind a meta-variable or to introduce an abbreviation for a term. If x is an unbound meta-variable in the proof state, the command *set* will instantiate this variable with the term t. The substitution $x \rightarrow t$ is added to the proof state. If x is already bound, a failure is generated. The formula $x = t$ will be added as a new premise to the task (see Figure 22).

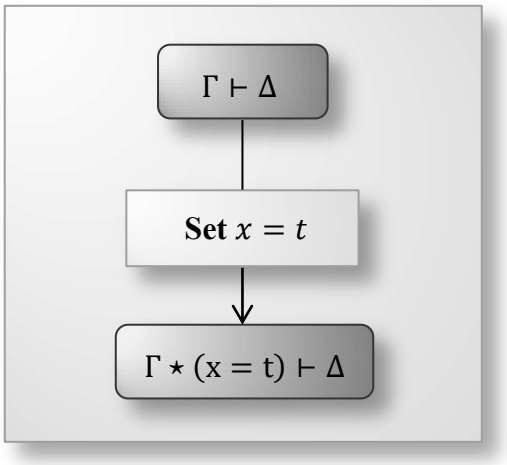


Figure 22. Set Check

Trivial. The command *trivial* is used to indicate that a task is solved (see Figure 21). This is the case if a formula φ occurs on both the left and the right side of the task, if the symbol false occurs at top level on the left hand side of the task, or if the symbol true occurs at top level on the right hand side of a task. A task can also be closed if the inference *name* in *by* is applied and all its premises and conclusions are matched to term positions in the current task.

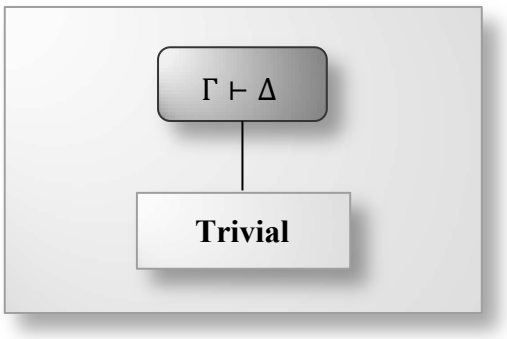


Figure 21. Trivial Check

Subgoals. The command *subgoals* reduces a goal of a given task to $n + m$ subgoals, each of which is represented as a new task, where n corresponds to the subgoals specified by the user and m denotes additional underspecified goals the user has omitted or forgotten (see Figure 23). Each new task stems from a premise P_i of the applied inference, where the goal of the original task is replaced by the proof obligation for the premise. The check succeeds if the inference *name* specified in *by* introduces at least the subgoals specified by the user.

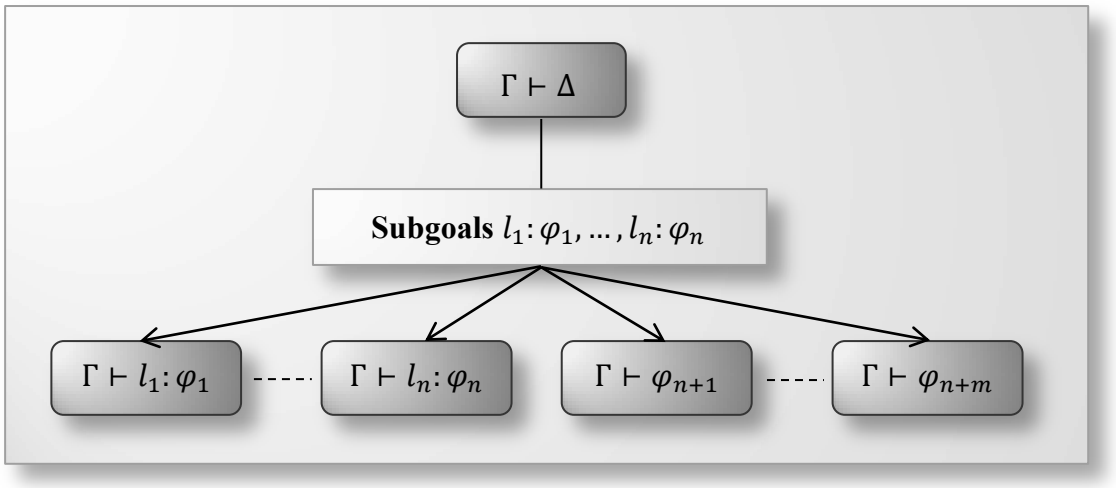


Figure 23. Subgoals Check

If the user does not specify any inference which is able to introduce the subgoals, the ΩMEGA system tries to further reduce the goal in the current task, thus introducing further subgoals, until all specified subgoals are found. As in the case for *assumption*, the antecedent of the sequent is untouched.

Cases. The command *cases* reduces a task containing a disjunction on the left hand side of the task into $n + m$ subtasks where in each case an additional premise is added. As for the *subgoals* command, the user can leave out some of the cases. If the task does not contain a suitable disjunction, ΩMEGA tries to derive a desired disjunction by forward reasoning. The goal remains untouched.

Complex. The command *complex* is an abstract command which subsumes an arbitrary sequence of the previous commands. Note that it is generally not possible to justify such a step with a single inference application, and without further information a full blind search has to be performed to justify this type of proof step.

An important result for the resource-bounded proof search is the empirical evaluation with proof steps of first-year mathematics students in [Benzmüller *et al*, 2007]. They have shown that 95,9% of all proof steps can be verified using a proof search depth limit of just 4 assertion-level steps. These results validate the decision to use Serge Autexier's CoRe calculus [Autexier, 2005] as an alternative to the ND calculus [Gentzen, 1934]. The CoRe calculus supports deductions deeply inside a given formula without requiring preceding structural decomposition as needed in ND or sequent calculus. In ΩMEGA we thus have a smaller distance between the semantic proof representation level and their expansion to the verifiable assertion level.

Proof Lifting. Whenever a part of a proof is changed or extended by ΩMEGA, it must be propagated back to PLATΩ. In principle the proof assistance system can insert arbitrary large parts. Given a selected part of a proof, each proof construction step has to be transformed back into a command of the semantic proof representation language. This so called *proof lifting* process is done by the following static analysis of the changes that a proof construction step applies to the proof agenda.

A proof construction step is executed with respect to an agenda $A_1 = \langle\{\boldsymbol{T_1}, \ldots, T_n\}, \sigma\rangle$ and results in the new agenda $A_2 = \langle\{\boldsymbol{T'_1}, \ldots, T'_k, T_2, \ldots, T_n\}, \sigma'\rangle$. The step has reduced the task T_1 to the subtasks $succ(T_1) = \{T'_1, \ldots, T'_k\}$. We analyze the differences between two tasks $T = \Gamma \vdash \Delta$ and $T' = \Gamma' \vdash \Delta'$ as follows:

$$taskdiff(T, T') = \langle\{\varphi \in \Gamma' | \varphi \notin \Gamma\}, \{\psi \in \Delta' | \psi \notin \Delta\}\rangle$$

If a task is reduced to several subtasks, we obtain a set of differences for each subtask. The differences between the two agendas A_1 and A_2 are then defined by:

$$agendadiff(A_1, A_2) = \langle\{taskdiff(T_1, T_i) | T_i \in succ(T_1)\}, \{\gamma \in \sigma' | \gamma \notin \sigma\}\rangle$$

Moreover, we require that the *name* of the applied proof operator is available, and that the function *lab* returns the set of labels of the formulas which are used in the premises and conclusions of the proof operator. Then, the proof lifting rules are defined with respect to the differences between two agendas as shown in Table 45.

$\boldsymbol{agendadiff(A_1, A_2)}$	**Semantic proof step**
$\langle\emptyset, \emptyset\rangle$	`\trivial[\by[`$name$`],\from[`$lab(name)$`]]`
$\langle\{\langle\{\boldsymbol{x = t}\}, \emptyset\rangle\}, \{(\boldsymbol{x \to t})\}\rangle$	`\set[`$\boldsymbol{x = t}$`]`
$\langle\{\langle\{\boldsymbol{\varphi}\}, \emptyset\rangle\}, \emptyset\rangle$	`\fact[`$\boldsymbol{\varphi}$`,\by[`$name$`],\from[`$lab(name)$`]]`
$\langle\{\langle\{\boldsymbol{\varphi}\}, \{\boldsymbol{\psi}\}\rangle\}, \emptyset\rangle$	`\decompose[\assume[`$\boldsymbol{\varphi}$`],\obtain[`$\boldsymbol{\psi}$`],` `\by[`$name$`],\from[`$lab(name)$`]]`
$\langle\{\langle\emptyset, \{\boldsymbol{\psi}\}\rangle\}, \emptyset\rangle$	`\goal[`$\boldsymbol{\psi}$`,\by[`$name$`],\from[`$lab(name)$`]]`
$\langle\{\langle\emptyset, \{\boldsymbol{\psi_1}\}\rangle, \ldots, \langle\emptyset, \{\boldsymbol{\psi_m}\}\rangle\}, \emptyset\rangle$	`\subgoals[`$\boldsymbol{\psi_1}, \ldots, \boldsymbol{\psi_m}$`,` `\by[`$name$`],\from[`$lab(name)$`]]`
$\langle\{\langle\{\boldsymbol{\varphi_1}\}, \emptyset\rangle, \ldots, \langle\{\boldsymbol{\varphi_m}\}, \emptyset\rangle\}, \emptyset\rangle$	`\cases[`$\boldsymbol{\varphi_1}, \ldots, \boldsymbol{\varphi_m}$`,` `\by[`$name$`],\from[`$lab(name)$`]]`

$\langle\{\langle\boldsymbol{\Gamma_1}, \boldsymbol{\Delta_1}\rangle, \ldots, \langle\boldsymbol{\Gamma_m}, \boldsymbol{\Delta_m}\rangle\}, \emptyset\rangle$	`\complex{}[`
	`\comp[\assume[`$\boldsymbol{\Gamma_1}$`],\obtain[`$\boldsymbol{\Delta_1}$`]], …`
	`\comp[\assume[`$\boldsymbol{\Gamma_m}$`],\obtain[`$\boldsymbol{\Delta_m}$`]],`
	`\by[`$name$`],\from[`$lab(name)$`]]`

Table 45. Proof Step Lifting Rules
(adapted from [Dietrich et al*, 2008])*

After introducing the semantic representation language for mathematical documents which is used by ΩMEGA's interface for document services, we illustrated the different types of proof steps which can be used to sketch a proof and we briefly described how they are processed. Furthermore, we presented how parts of a proof, which have been modified by ΩMEGA, can be lifted to corresponding modifications of the semantic representation. Now we will introduce the document-centric services provided by ΩMEGA via PLATΩ on that basis. The following services have been developed to assist the authoring of mathematical proofs. The integration of these services with the text-editor will be described in more detail in the Chapters 8 and 9.

Verification Service. For a mathematical document in semantic representation PLATΩ offers to verify all contained proofs. To this purpose, PLATΩ stores the theory knowledge of the mathematical theories in MAYA and constructs the proofs in the TASKLAYER. For each proof, the proof data structure ($\mathcal{PDS}$) is initialized and a high-level proof plan is built according to the proof steps in the document. Then, ΩMEGA tries to expand the proof plan to the verifiable assertion level with a resource-bounded search. Thus, a proof step is usually justified by a sequence of lower level proof steps, which turns the proof data structure into a hierarchical data structure. Finally, PLATΩ returns the information about the verification status of proof steps by attributing the corresponding element in the semantic representation as follows.

```
\decompose{ "verified" }[…], \fact{ "unknown" }[…],…
```

Explanation Service. The explanation service is only provided for a particular proof step that has already been verified. The intention of this service is to show more detailed proof steps that justify that particular step. For this purpose, we exploit the hierarchical proof data structure by replacing the selected proof step in the semantic representation with the lifted sequence of proof steps that justifies this step. This expansion can of course be reversed again on request, resulting in an abstraction of a sequence of proof steps.

Proving Service. Finally, ΩMEGA can be asked which assertions (definitions, axioms, lemmas, theorems) are applicable at the actual task of a particular branch in a proof. Furthermore, the application of a selected assertion can be requested as well as an automated completion of parts of the proof. Both requests are processed with bounded resources and as a matter of course may run out of resources before discovering the expected results. The new constructed parts of the proof are lifted to corresponding modifications of the semantic representation and returned by PLATΩ.

7.2 Course Scenario

To demonstrate the benefits of the *Change-Oriented Architecture* to *Mathematical Authoring Assistance* we selected the following course scenario. The lecturer of a mathematics course writes down the lecture notes and produces weekly exercise sheets. Thereby, the exercises are based on the definitions, axioms, theorems and notations that are contained in the lecture notes. The students solve the exercises and write down their solutions. In this scenario, we will address the following two authoring assistance cases.

1) How can we assist the lecturer with the authoring of lecture notes?
2) How can we assist the student with the authoring of exercise solutions?

In order to use the verification services of ΩMEGA, the content of a mathematical document in a text-editor has to be transformed into the semantic representation required by ΩMEGA. Since the exercises depend on the mathematical content in the lecture notes, in particular on the introduced notations, we first have to transform the lecture notes and extend the transformation grammar with rules for the new notations, before we transform the exercises and their solutions. In general, we will focus on the following aspects of the authoring assistance cases.

Authoring Lecture Notes. In this assistance case, our main topic is the notation used by mathematical formulas. How can we analyze the definitions of new notations? How can we extend the invertible grammar for mathematical formulas with synthesized rules for the new notations? How can we support sugaring and souring of notations? How can we adapt the document if a notation is changed? How can we deal with ambiguities? How can we support notational communities of practice?

Authoring Exercise Solutions. In this assistance case, we focus on the verification of the exercise solutions. How can we analyze the proof steps and justifications? How can we incrementally verify the proof steps? How can we propagate changes? How can we integrate the feedback of the verification services in the text-editor? How can we interact with the proof assistance system?

Since the transformation of the text-editor document into the semantic representation is the foundation for authoring assistance, we first have to collect and analyze samples of the content that we have to expect in this scenario.

7.3 Exploratory Study

In order to analyze the linguistic requirements of the expected mathematical content in the course scenario, we conducted an *exploratory study* with a first-year mathematics course at Saarland University to collect a corpus of exercise solutions. We have chosen the exercises of a first-year course because they fit reasonably well to the capabilities of the proof assistance system ΩMEGA.

The idea of the *exploratory study* is to make the students familiar with the text-editor TEXMACS which we use as a user interface (UI) for creating, browsing, solving and submitting exercises and their solutions. The main requirement for the study is the storage of successive versions of exercise sheets, exercises and their student-specific solutions. Therefore we integrated the text-editor TEXMACS with an exercise manager by using a mediation module as shown in Figure 24.

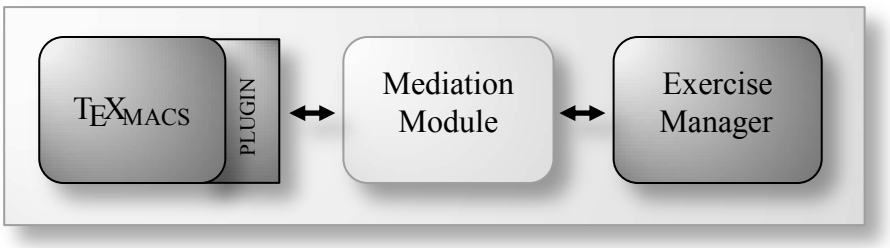


Figure 24. Architecture of the Exploratory Study

The job of TEXMACS is to serve as a UI for navigating through exercises and solutions. We encoded the exercise menu of the study as a regular document (see Figure 25) which communicates with the exercise manager in an AJAX-style fashion [Garrett, 2005] via the mediation module. The exercise manager provides the services to browse exercise sheets and exercises, and to store their solutions. Furthermore, the user accounts of the students are managed by the exercise manager.

In order to analyze the authoring behavior over time, we are also interested in successive snapshots of the exercise solutions written in the text-editor. To this purpose, we use the versioning feature of the mediation module. A timed event in TEXMACS automatically commits new versions of an exercise solution to the mediation module. Hence, successive snapshots of this interface document are tracked by the version control of the mediation module. In addition to that, we developed a replay mechanism in TEXMACS to watch the authoring of exercise solutions over time, simulated by successive checkouts of subsequent solution versions from the mediation module. Thus, we are able to monitor the authoring behavior over time.

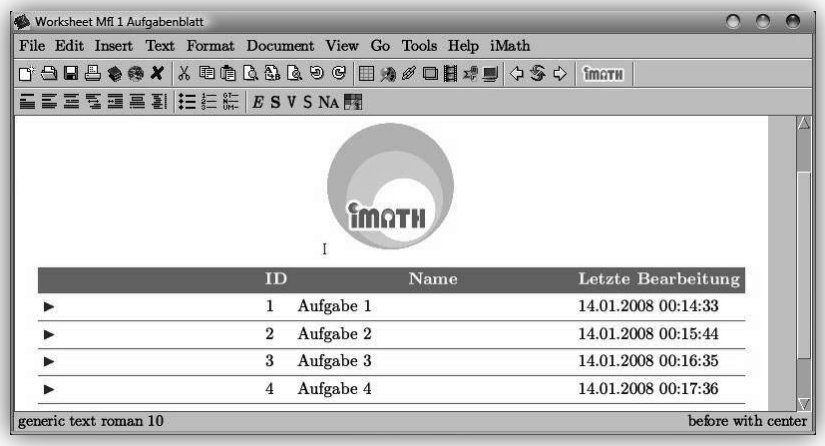


Figure 25. TEXMACS UI of the Exploratory Study

In the first week of the *exploratory study*, we introduced TEXMACS to the students who then got quickly familiar with the intuitive usage provided by the WYSIWYG interface. In the following ten weeks the students had to solve and submit one out of four exercises per week with TEXMACS. After that training period we conducted the real study, a supervised classroom exercise that had to be completely solved using the text-editor TEXMACS. Thereby, 49 students had 90 minutes to solve the following 4 exercises:

1) Let $(a_n)_{n\in\mathbb{N}}$ be a convergent sequence in $\mathbb{R}$ with limit $a \in \mathbb{R}$. Show that it is a Cauchy sequence.
2) Compute the following limits of complex-valued sequences:
   $\lim_{n\to\infty} \frac{2in^3-n^4}{n^4+3in^2-1}$, $\lim_{n\to\infty} \frac{3n^2-2}{n^3+1}$, $\lim_{n\to\infty} \frac{5n-7n^2}{(n+1)^2-8n}$.
3) For $x \in \mathbb{R}_+$ compute $\lim_{n\to\infty} \frac{x^n-n}{x^n+n}$.
4) Compute the following limits of the sequences $(x_n)_{n\in\mathbb{N}}$ with the help of limits proved in the lecture: $x_n = \left(1-\frac{1}{n}\right)^n$, $x_n = \left(1-\frac{1}{n^2}\right)^n$.

Our software took automatic snapshots of the current state of the exercise solutions every 10 seconds. Note that we had no influence on the type of exercises at all, the exercise sheet was completely regular. The exploratory study has been conducted in German. The above exercise descriptions have been translated into English. More details of the exploratory study, briefly called IMATH, are available at [Wagner, 2007] and published as [Wagner & Lesourd, 2008].

7.4 Authoring Behavior

By using the replay mechanism (see Figure 26) specifically developed for this study we were able to analyze the authoring behavior of the students per exercise over time. First we observed that the students spend on average 30% of their time with searching relevant definitions and theorems in the lecture notes, another 30% with producing a draft solution on paper, and the final 40% with authoring the solution.

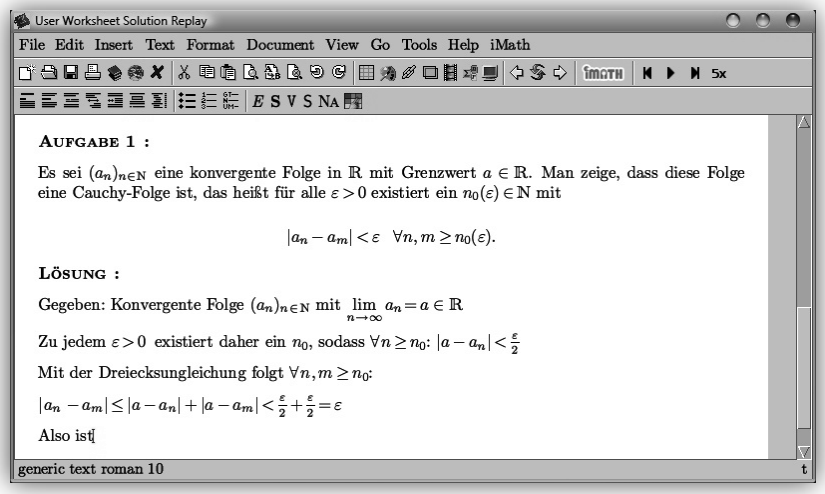


Figure 26. Exercise Replay in TEXMACS

Analyzing the authoring behavior inside TEXMACS, we were able to identify the following types of modifications:

- The standard type of modification is *monotonic increase* of the solution when the student is continuously writing down the solution.
- *Copy & Paste* is used quite often to shorten the complicated input of long mathematical formulas that differ only slightly from previous formulas.
- *Local modifications* can be observed regularly. That means that parts of the last sentence are deleted, rewritten or corrected, like for example adding missing brackets in formulas.
- *Global modifications* can only be observed in very few cases. Refactoring operations like variable renaming affect in general more than one sentence. Sometimes such an operation is only described but not executed, for example ”*By renaming* ε_2 *to* ε *the theorem follows*.”.

7.5 Linguistic Aspects

The mathematical documents collected during the experimental study have been analyzed with respect to different linguistic aspects. In the following we will report in more detail on the analysis of the aspects *formula verbalization*, *style of sentences*, *concluding step* and *justifications*. We selected these aspects because the collected documents show significant differences with respect to these aspects. Note that the exploratory study has been conducted in German. The examples, which we reproduce in the following, have been translated from German into English.

Formula Verbalization. The aspect formula verbalization is divided into the following categories:

1) ***formalized*** (e.g. "$\lim_{n\to\infty} a_n = a$", "$|a_n - a_m| < \varepsilon \quad \forall n, m \geq n_0(\varepsilon)$") where formulas are completely written in symbolic notation,
2) ***weakly verbalized*** (e.g. "a *is the limit of* $(a_n)_{n\in\mathbb{N}}$", "*There exists a* $\varepsilon > 0$ *for which holds that* $\forall n_0(\varepsilon). \exists n, m \geq n_0(\varepsilon)$ ...") where some relations or quantifiers are partly verbalized, and finally
3) ***strongly verbalized*** (e.g. "*For all* ε *holds: there exists a* $n_0(\varepsilon) \in \mathbb{N}$ *with …*", "*For* $x < 1$, x^n *converges on* 0.") where all relations and quantifiers are fully verbalized.

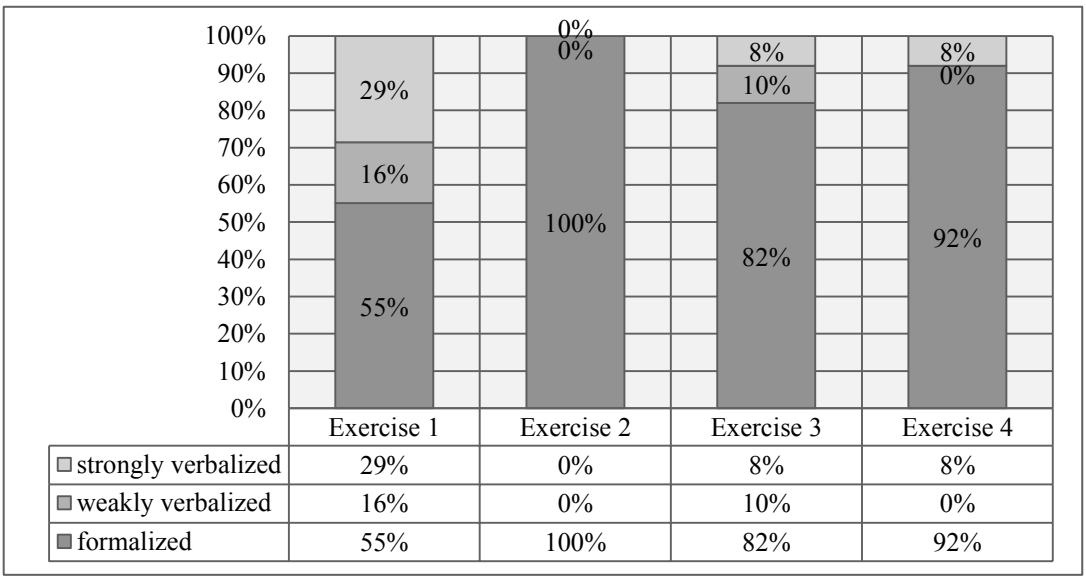


Figure 27. Analysis of Formula Verbalization

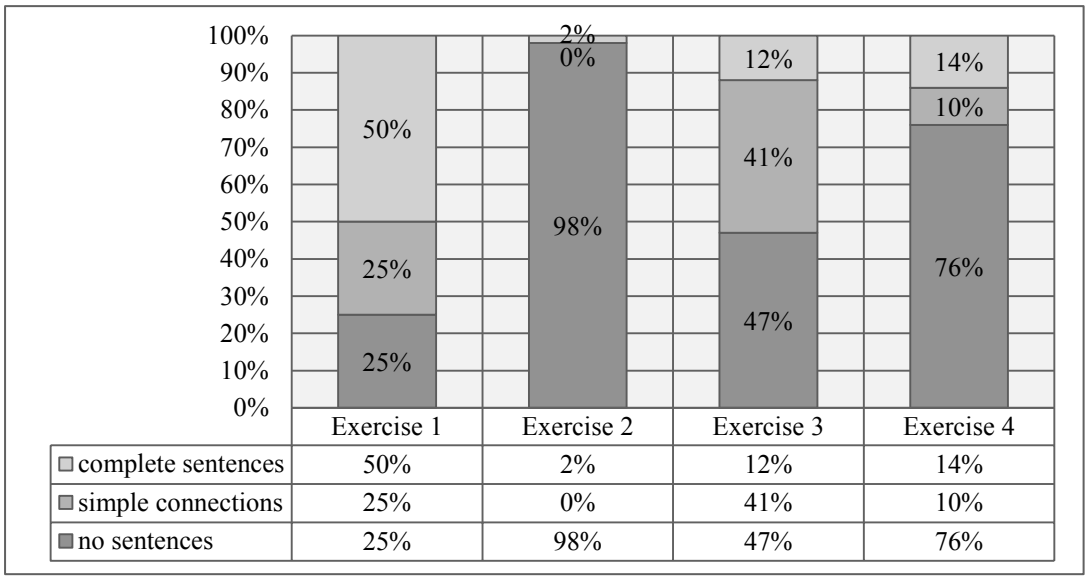
Figure 27 shows the formula verbalization grade for every exercise. An exercise is classified as weakly/strongly verbalized if at least one weakly/strongly verbalized formula occurs. The analysis of the formula verbalization reveals that

- services for symbolic formulas (for example notation check) are worthwhile,
- being able to fully parse/render symbolic formulas and weakly verbalized formulas gives a coverage of at least 70%,
- a linguistic ontology is required to deal with strongly verbalized formulas.

While the first two grades of formula verbalization may be reached by a general shallow grammar, the third grade requires a significant amount of domain-specific information and thus domain knowledge deeply encoded in the grammar.

Style of Sentences. The style of sentences is classified into the following categories:

1) ***no sentences*** if there are no sentences at all,
2) ***simple connections*** if the sentences contain simple key phrases (e.g. "*Therefore it holds that ...*", "*It holds ...*", "*Hence we have ...*", "*Thus it follows ...*", "*Then ...*", "*This means ...*", "*Let ...*", "*Select ...*", "*Assuming ...*") and the building blocks are symbolic or weakly verbalized formulas, and
3) ***complete sentences*** if the sentences are fully verbalized with ontological building blocks like type, identifier, concept or attribute.



	Exercise 1	Exercise 2	Exercise 3	Exercise 4
complete sentences	50%	2%	12%	14%
simple connections	25%	0%	41%	10%
no sentences	25%	98%	47%	76%

Figure 28. Analysis of the Style of Sentences

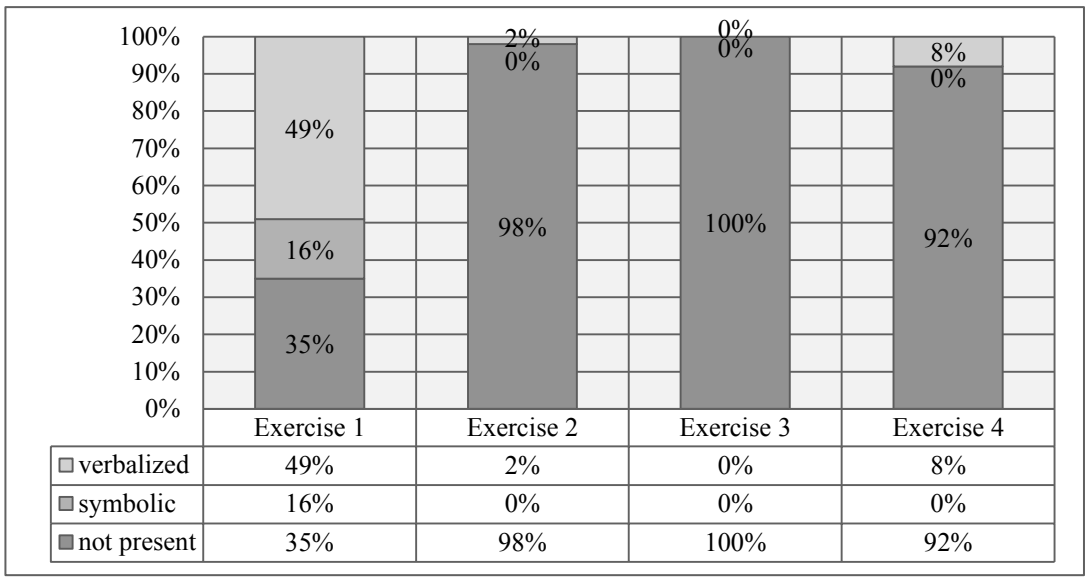
The analysis of the style of sentences in Figure 28 shows that

- a linguistic ontology is absolutely necessary to support text-style proofs like the one in the first exercise because a coverage of less than 50% is not acceptable,
- for computational proofs with case distinctions the handling of simple connectors and weakly verbalized formulas gives an excellent coverage of at least 85%.

Again the first two styles of sentences may be covered by a general shallow grammar, but the third style requires a significant amount of domain-specific information and thus domain knowledge deeply encoded in the grammar. As an intermediate step, one could define a controlled mathematical language which allows for supporting text-style proofs in a controlled fashion.

Concluding Steps. The concluding steps are divided into the following categories:

1) ***not present*** if there is no step concluding the proof,
2) ***symbolic*** if there is a ■, "*qed*" or "*q.e.d.*" at the end of the proof, and
3) ***verbalized*** if the concluding step is a fully verbalized sentence (e.g. "*Therefore* $(a_n)_{n\in\mathbb{N}}$ *is a Cauchy sequence.*", "*... hence the theorem holds.*", "*This contradicts the assumption.*", "*... and thus* $|a_n - a_m| < \varepsilon \quad \forall n, m \geq n_0(\varepsilon)$").



	Exercise 1	Exercise 2	Exercise 3	Exercise 4
verbalized	49%	2%	0%	8%
symbolic	16%	0%	0%	0%
not present	35%	98%	100%	92%

Figure 29. Analysis of Concluding Steps

Note that there are two types of verbalized concluding steps, one *abstract* type that could be used for every proof of the same style, and one *concrete* type that performs the last inference, repeats the theorem or justifies the last conclusion. Figure 29 shows the analysis of concluding steps which draws the following conclusions:

- Verbalized concluding steps do almost never occur in computational proofs, even symbolic conclusions are very rare.
- In correlation to the sentence style aspect we can observe that 50% of the text-style proofs require a linguistic ontology in order to understand the verbalized concluding step.

A verbalized concluding step either requires domain knowledge deeply encoded in the grammar or it could be part of a controlled mathematical language. The analysis shows that the concluding step almost solely occurs in text-style proofs. Thus, the expected style of proofs indicates whether the support of concluding steps by the grammar is reasonable.

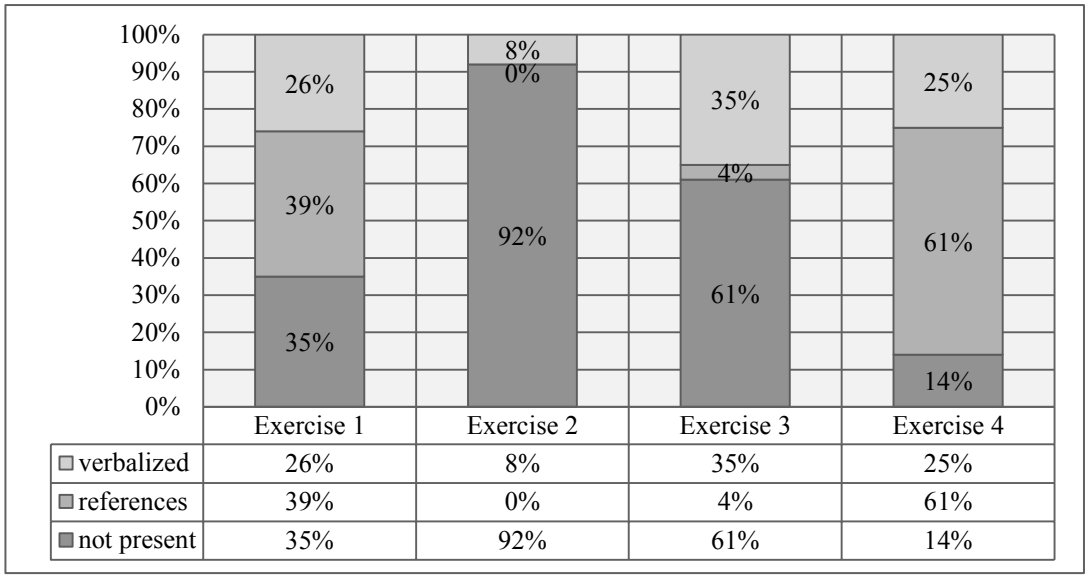
Justifications. The last aspect concerns the justification of proof steps and consists of the following categories:

1) ***not present*** if no proof step is justified at all,
2) ***references*** if the proof steps are justified by referring to used theorems or concepts (e.g. "... *with the triangle inequality*", "*From* 14.23 *it holds with* $p = -1$ …", "$x_1 \overset{theorem\ 14.23}{\Longrightarrow} x_2$"), and
3) ***verbalized*** if the justification is a fully verbalized sentence, possibly with references to proof strategies, complex methods or other sources of knowledge (e.g. "*Since ... it holds ...*", "*... holds because ... and ...*", "*By polynomial division we compute ...*", "*Reason: Main coefficient is* -1", "*In the lecture we proved that ...*", "*From the first part of the exercise it follows that ...*", "*It's a matter of common knowledge that ...*").

An exercise is classified as references/verbalized if at least one reference/verbalized justification sentence occurs in a mathematical proof. The analysis of justifications in Figure 30 reveals that

- not supporting verbalized references results in a medium coverage of at most 65% in the worst case, which might result in a low user acceptance,

- different linguistic ontologies are required to effectively understand verbalized references: a *document* ontology (e.g. for references to other parts of a document), an *argumentation* ontology (e.g. for references to the lecture or domain-specific techniques), and a *concept* ontology (e.g. for references to definitions). At least the concept ontology needs to be extended dynamically for example whenever new definitions or named local hypothesis are stated.



	Exercise 1	Exercise 2	Exercise 3	Exercise 4
verbalized	26%	8%	35%	25%
references	39%	0%	4%	61%
not present	35%	92%	61%	14%

Figure 30. Analysis of Justifications

The cited examples of justifications by references and verbalized justifications show a wide range of notational and argumentative creativity. The collected documents reveal indeed that every author has its unique style of justifying proof steps. Without imposing authoring guidelines, the complexity of the justification analysis would be out of scope for an automated analysis. An alternative solution would be to define justification schemes as part of a controlled mathematical language.

In summary, the analysis of linguistic aspects allows for answering the choice between shallow and deep linguistic grammars as follows. We assume that an author is satisfied with the system if 90% of a common document are covered. Then, we conclude from the analysis results that a system with a shallow linguistic grammar will not be able to satisfy an author. Because of the wide range of justification variants, the development of a system with a deep linguistic grammar and 90% of coverage is unfortunately out of scope for this thesis. Besides, the amount of ambiguities would increase with the expressive power. Therefore, we argue that a system with a controlled mathematical language is a reasonable compromise between both extremes.

7.6 Discussion

In order to develop an assistance system for the course scenario, which integrates the services of a proof assistance system with the authoring of lecture notes and exercise solutions, first of all we had to analyze the linguistic requirements of the expected mathematical content. Therefore, we conducted an exploratory study with a first-year mathematics course to collect and analyze a corpus of documents. We have chosen exercises of a first-year course because their problem complexity fits to the capabilities of a state-of-the-art proof assistance system like the ΩMEGA system. The collected documents have been analyzed with respect to the authoring behavior and some selected linguistic aspects.

Regarding the authoring behavior, we observed a monotonic increase with the frequent use of copy & paste. The last sentence is quite often modified locally, global refactoring is used very rarely and often only described but not executed. Notational consistency checks would be a very helpful authoring assistance service because a lot of modifications were related to the correction of notational errors. Altogether, since the authoring behavior is rather incremental, the incremental translation method of the invertible grammar formalism is well-suited for the continuous transformation of the document in the text-editor into the semantic representation for the proof assistance system.

Regarding the linguistic aspects we analyzed that being able to fully deal with symbolic and weakly verbalized formulas, simple connected sentences and references leads to an average coverage of 70% in our course scenario. This level might be reached with a shallow linguistic grammar. By integrating linguistic ontologies (document, argumentation, and concept) and mechanisms to extend these ontologies dynamically, we might reach a coverage of more than 90%, which usually satisfies the expectations of an author. However, the analysis of justifications revealed a huge amount of linguistic and argumentative variations. The resources needed to develop a system with a deep linguistic grammar and a coverage of 90% are out of scope for this thesis. Nevertheless, a deep linguistic approach is being pursued for example by [Wolska & Kruijff-Korbayová, 2004]. As of May 2010, we could not test the coverage with respect to our collected corpus because there is no system available yet.

As a compromise between shallow and deep linguistic approaches, we use a combination of an extensible shallow grammar for mathematical formulas and a controlled mathematical language for text-style mathematical theories, containing in particular definitions, axioms, theorems and proofs with verbalized justifications. In the following chapters we will illustrate how far we can get with this approach and the concepts and methods of the *Change-Oriented Architecture*.

8 Authoring Lecture Notes

As a first step towards assisting the authoring of mathematical documents like lecture notes, we integrated the proof assistance system ΩMEGA with the text-editor TEXMACS using an annotation based approach. Thereby we rely on annotations for the document structure that must be provided manually by the user to understand the meaning of the natural language parts in a mathematical document. Although it might be acceptable for an author to indicate the macro-structures like theories, definitions and theorems, writing annotated formulas (`\F{"in"}[\V{"x"},\F{"cup"}[\V{"A"},\V{"B"}]]`) instead of presentation formulas (`x \in A \cup B`) is definitely not acceptable. Aiming at a document-centric approach to formalizing mathematics, we present in this chapter a mechanism that allows authors to define their own notation und to use it when writing formulas within the same document. Furthermore, this mechanism for notational self-extensible documents enables the proof assistance system to access the formal content and use the same notation when presenting computed formulas to the author.

In this chapter, we will investigate how notational information in a document can be represented, processed and used to automatically annotate and generate mathematical formulas in the same document. First of all, we will present the transformation pipeline between the document in the text-editor and the semantic representation for the proof assistance system using a *Change-Oriented Architecture*. Inspired by notation definitions in text-books, we then present the means the author should have to define notations. The goal consists of starting from such notations to synthesize invertible grammar rules that allow for reading formulas using that notation. By automatically inverting these rules we are then also able to render the formulas generated or modified by the proof assistance system. After that, we present a basic mechanism for accommodating efficiently modifications of the notations. Finally, we will outline techniques for ambiguity resolution and for the redefinition of notation.

Presentational Convention. The work presented in this chapter has been mainly realized with the text-editor TEXMACS. Although the TEXMACS mark-up-language is analogous to LATEX-macros, one needs to get used to it: For instance a macro application like "`\frac{A}{B}`" in LATEX becomes "`<frac|A|B>`" in TEXMACS-markup. Assuming that most readers are more familiar with LATEX than with TEXMACS, we will use our LATEX-style syntax for labeled trees for the sake of readability.

8.1 Transformation Pipeline

We model the transformation between the document in the text-editor and the semantic representation for the proof assistance system by the Chain of Responsibility Design Pattern of the *Change-Oriented Architecture*, as shown in Figure 31.

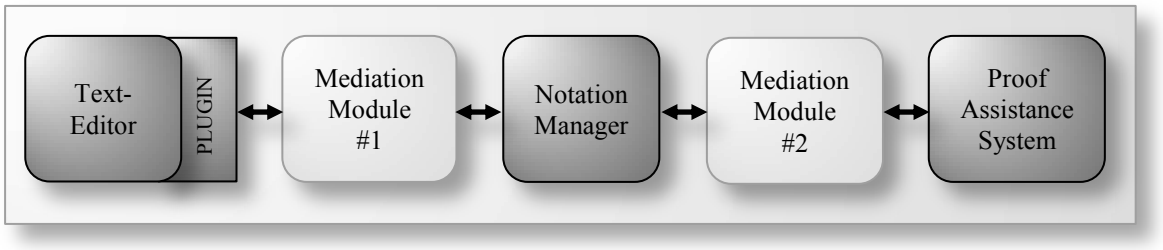


Figure 31. Transformation Pipeline

Let us discuss now the different responsibilities of the components in this pipeline, and how they interact in the usual authoring workflow.

Layout Processing. The main responsibility of the *plugin* in the text-editor is the pre-processing of the document markup, in particular of the layout markup. The document in the text-editor contains plenty of layout information in its plain serialized representation, as well as other kinds of markup which are not relevant for the authoring assistance. Covering this additional markup by the grammar rules of a mediation module is not a reasonable approach because of the tremendous combinatorial possibilities of all kinds of markup. Therefore, we decided to extract the relevant content of the document by applying projections on the original document to lift the content subtrees in the document tree. Table 46 shows an example for lifting markup, where the *document* element represents a *paragraph* and the *concat* element a *run* in the document format of TEXMACS.

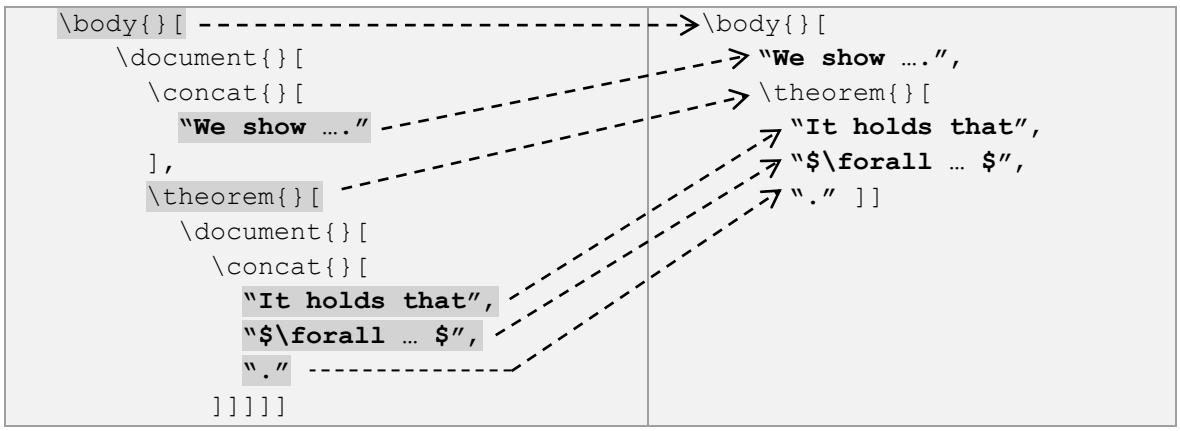


Table 46. Plain Markup (on the left) vs. Lifted Markup (on the right)

For the inverse processing direction, that is, the integration of generated content, we designed a *layout heuristic*. This heuristic scans the generated content for keywords like *definition*, *axiom*, *theorem* and *proof*, which indicate a major group of content, and places this part of the content in the top paragraph of the document. The default behavior of the heuristic is to insert generated content within the same run as the referenced target.

Component Interaction. The task of the *first mediation module* is then to translate between the filtered annotated document of the text-editor and a semantic representation for the proof assistance system. The translation process includes theorems, proofs and proof steps, but it does in particular not include the translation of the content of definitions, notations and mathematical formulas. The *notation manager* now analyzes the definitions and notations in the interface document of the first mediation module. The result of the analysis is then used to synthesize invertible grammar rules for the translation of mathematical formulas. The notation manager extends the invertible grammar of the second mediation module by these new grammar rules. Finally, the *second mediation module* translates the remaining plain parts of the interface document into a complete semantic representation for the proof assistance system.

Synthesizing grammar rules from the defined notation is a process that does not need to be inverted. Therefore it is encapsulated in a service component and not part of a mediation module. Note that the proof assistance system is not able to trigger changes in the notation because the definition of the notation is not anymore part of the interface documents of the second mediation module. This knowledge is only part of the grammar which has been set up for the interface documents by the notation manager. Hence, we prevent the need for inverting the synthesizing of grammar rules by design.

By describing this transformation pipeline, we only presented a high-level overview of the tasks and the interplay of the components of this *Change-Oriented Architecture*. In the following sections, we will provide detailed solutions for the different steps in the transformation pipeline. First of all, we will describe how the semantic annotation language can be translated with an invertible grammar. Then we will present the method for synthesizing grammar rules by analyzing the defined notation. Furthermore, we will discuss how the features of the *Change-Oriented Architecture* can be exploited for the management of notation. Thereby, we will address questions like: How can we support sugaring and souring of notations? How can we adapt the document if a notation is changed? How can we deal with ambiguities? How can we support notational communities of practice?

8.2 Semantic Annotation Language

Mediating between a text-editor and a proof assistance system requires the extraction of the formal content of a document, which is already a challenge in itself if one wants to allow the author to write in natural language without any restrictions. Therefore we currently use the semantic annotation language developed for the PLATΩ system [Wagner *et al*, 2006] as introduced in Chapter 7.1 to semantically annotate different parts of a mathematical document. The annotations can be nested and subdivide the text into dependent theories that contain definitions, axioms, theorems and proofs, which themselves consist of proof steps like for instance subgoal introduction.

The annotations are a set of macros predefined in a TEXMACS style-file and must be provided manually by the author. We were particularly cautious that adding the annotations to a text does not impose any restrictions to the author about how to structure the text. Note that for the proof assistance system the only annotation where the order of arguments clearly matters is the annotation `PROOF` because the order of the contained proof steps is clearly relevant. The semantic annotation language can be modeled by an invertible grammar as shown exemplary in Table 47 for the annotations of a theorem. The first mediation module uses this grammar to translate the annotated document in the text-editor to the semantic representation for the proof assistance system, except of definitions, notation and mathematical formulas.

Head	Production	Creation
TEO	[\theorem{ **x:*NAME*** }[**y:*TEOC***]]	[\theorem{ **x** }[**y**]]
TEOC	[(***PRE***|***t***)* ***CON***? (***PRE***|***t***)*]	
PRE	[\precondition{}[**y:*PREC***]]	[\precondition{}[**y**]]
PREC	[(***FORM***|***t***)* ***FORM*** (***FORM***|***t***)*]	
CON	[\conclusion{}[**y:*CONC***]]	[\conclusion{}[**y**]]
CONC	[***t**** ***FORM*** ***t****]	
…		

Table 47. Invertible Grammar for the Semantic Annotation Language

Thereby, we use the following variable specification to exclude the predefined annotation macros as a matching partner of the pattern variable **t**. The role of this variable is indeed to capture only the annotated natural language content and any custom markup.

```
\variables{}[
    \negative{ name="t" }[ "theorem", "precondition",
                           "conclusion", … ],
    …
    \multirange{}[ "x" , "y" , … ]
]
```

Self-Extensibility. It is not acceptable to require writing formulas in an annotated form in the text-editor. This motivates the need for an *abstraction parser* that converts formulas in LaTeX syntax into their fully annotated form. Furthermore, we also need a *rendering parser* to convert fully annotated formulas obtained from the proof assistance system into LaTeX-formulas using the user-defined notation. The usual software engineering approach would be to write grammars for both directions and integrate the generated parsers into the system. Of course, this method is highly efficient but the major drawback is obvious: the user has to maintain the grammar files together with her documents. In our document-centric philosophy, the only source of knowledge for the mediation process should be the source document in the text-editor.

Therefore, the idea of *dynamic notation* is to start from a basic invertible grammar for types and formulas, where only the notation for the Boolean type `bool`, the complex type constructors $\to$ and $\times$, and the logic operators $\forall,\exists,\lambda,\top,\bot,\wedge,\vee,\neg,\Rightarrow,\Leftrightarrow$ are predefined. Based on that initial grammar the definitions and notations occurring in the document are analyzed in order to extend incrementally the invertible grammar for dealing with new notation knowledge. Note that we modeled the document as a set of mathematical theories with definitions, axioms, theorems and proofs. Thereby, the context of a theory allows for including the knowledge of other existing theories in the document. The scope of a notation should then respect the visibility of its defining symbol or type, that is, the transitive closure of dependent theories.

In order to avoid the Nixon diamond problem with the inheritance of theories, we require that the inheritance network has a linear shape. To solve the problem of combining invertible grammars in the case that the inheritance network would have a tree shape, we refer to the works of [de Carvalho & Jürgensen, 2008] and [Ganzinger & Giegerich, 1984] who describe methods for joining and composing attribute grammars.

Illustrating Example. Notations defined by authors are typically not specified as grammar rules. Therefore, we first need a user friendly WYSIWYG method to define notations and to automatically generate grammar rules from it. Looking at standard mathematical textbooks, one observes sentences like "*Let x be an element and A be a set, then we write $x \in A$, x is element of A, x is in A or A contains x.*". Supporting this format requires the ability to locally introduce the variables x and A in order to generate grammar rules from a notation pattern like $x \in A$. Without using a deep linguistic analysis, patterns like "*x is in A*" are only supported as pseudo natural language. Besides that, the author should be able to declare a symbol to be right- or left-associative as well as the precedence of symbols. We introduce the following annotation format to define the operator $\in$ and to introduce multiple alternative notations for $\in$ as close as possible to the textbook style.

```
\definition{"Predicate" "$\in$"}[
  "The predicate"
  \concept{"$\in$"}["$elem \times set \rightarrow bool$"}
  "takes an individual and a set and tells whether that
  individual belongs to this set." ]
```

A definition may introduce a new type by \type{*name*} or a new typed symbol by \concept{*name*}[*type*]. We allow to group symbols to simplify the definition of precedence and associativity. By writing \group{*name*} inside the definition of a symbol, this particular symbol is added to the group *name* which is automatically created if it does not exist. Any new concept is first introduced as a prefix symbol. This can be changed by declaring concept specific notations as follows.

```
\notation{"Predicate" "$\in$"}[
  "Let" \declare{"$x$"} "be an individual and"
  \declare{"$A$"} "a set, then we write"
  \denote["$x \in A$"] "," \denote["$x is element of A$"] ","
  \denote["$x is in A$"] "or" \denote["$A contains x$"] "." ]
```

A notation may contain some variables declared by \declare{*name*} as well as the patterns written as \denote[*pattern*]. Furthermore, by writing \left{*name*} or \right{*name*} inside the notation one can specify a symbol or group of symbols to be left or right associative. Finally, the precedence between symbols or groups are defined by \prec[$name_1, \ldots, name_k$], which partially orders the precedence of these symbols and groups of symbols from low to high. Please note that a notation is related to a specific definition by referring its name, in our example "Predicate $\in$".

Figure 32 shows how the above example definition and notation appear in a TEXMACS document. Using a keyboard shortcut the author can easily switch into a so-called "*box-mode*" that visualizes the semantic annotations contained in the document by using an alternative style sheet. Figure 33 shows this annotated view of the document.

So far we went rather informally through our concept of the self-extensibility of notation using illustrating examples. In the following we will describe the realization of this concept using the invertible grammar formalism. Furthermore, we will discuss important aspects as the *management of change for notation* and the *resolution of ambiguities*.

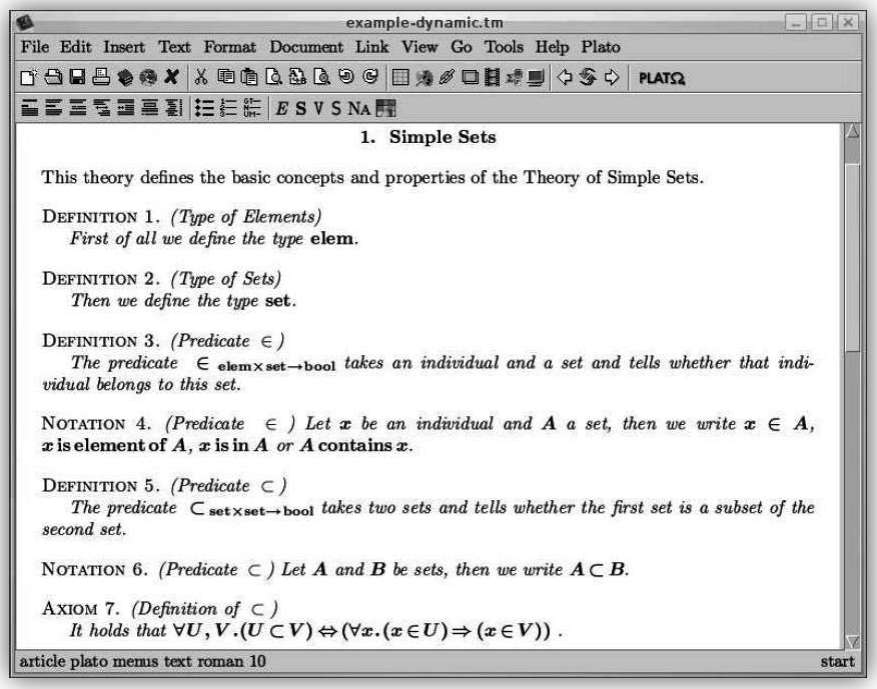


Figure 32. Annotated TEXMACS Document in Text Mode

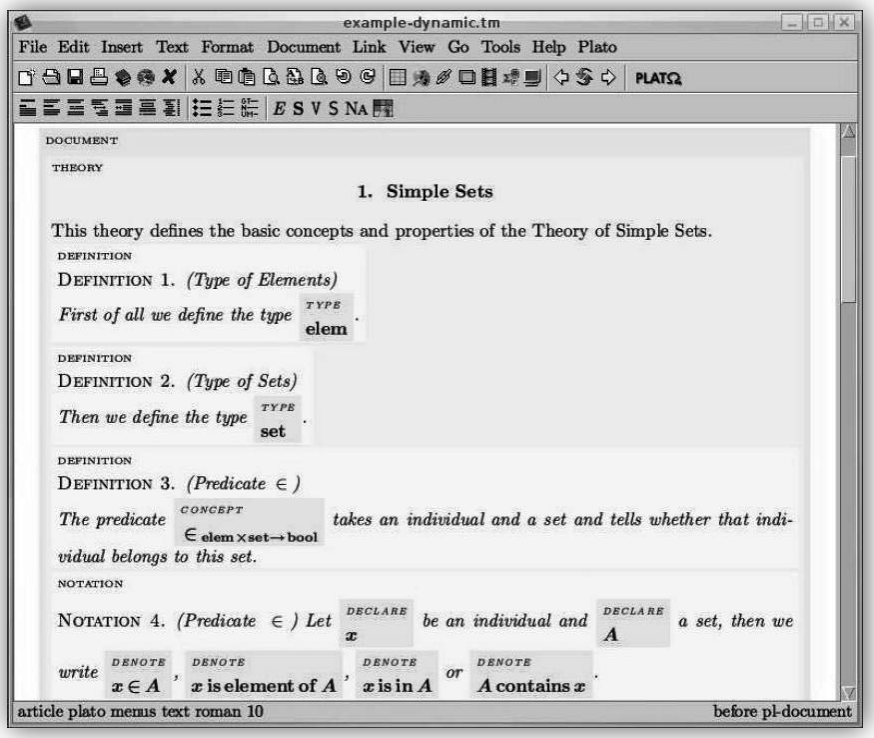


Figure 33. Annotated TEXMACS Document in Box Mode

8.3 Synthesizing Invertible Grammar Rules

We start now with the translation of a semantically annotated document, as for example the document shown in Figure 33. First of all, the natural language parts in the document are removed by the first mediation module in the transformation pipeline. The notation manager then receives a semantic representation of the document where the contents of definitions, notation introductions and mathematical formulas are not yet translated.

The notation manager then computes the linear inheritance chain of the theories contained in the document. For each of these theories, the notation manager sets up an interface document in the second mediation module with the initial invertible grammar. The rules for mathematical formulas can be divided into rules for types, symbols and operator applications. This first step is summarized in Figure 34.

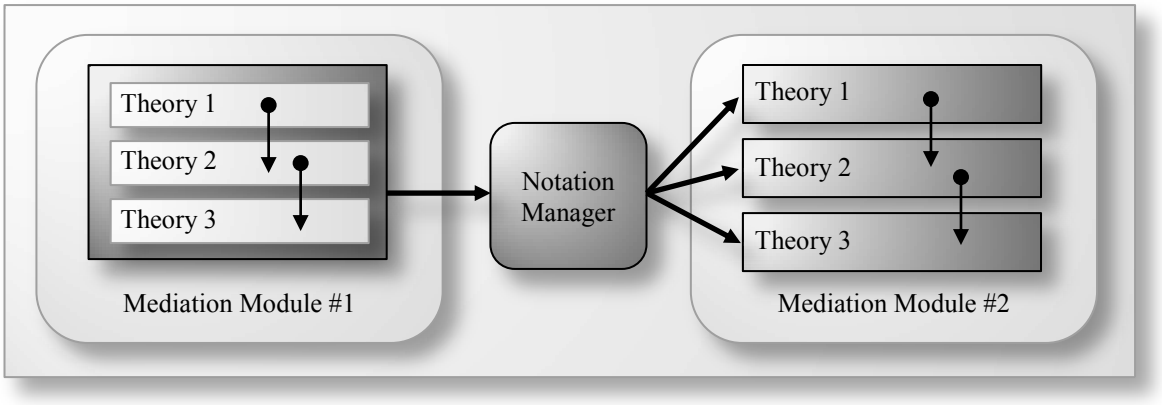
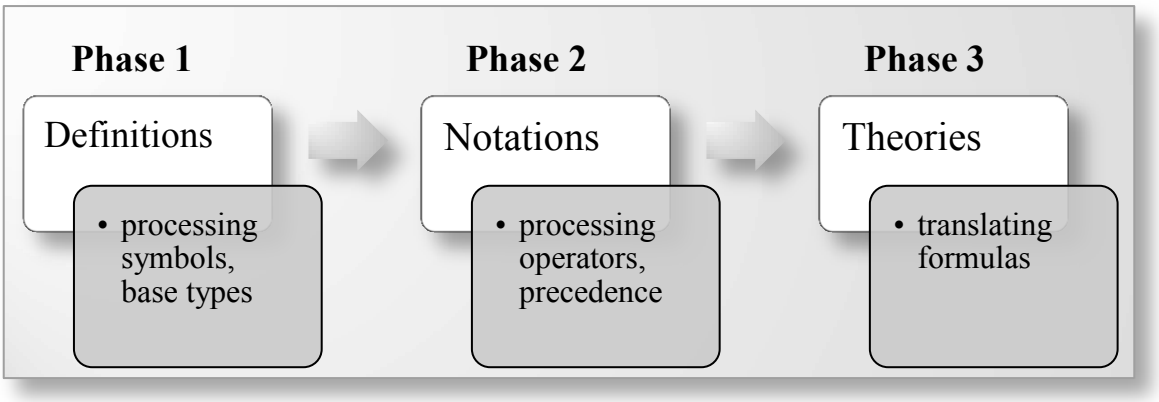


Figure 34. Inverse Multiplexing Mathematical Theories

The following rule is an example for an initial rule of the category `TERM`, which is used for any kind of operator application. This rule recognizes two chunks of input `x` and `y` in the category `TERM` that are separated by `"\wedge"` and it substitutes the obtained transformation results `v` and `w` into `\F{"and"}[v , w]` to create the semantic representation. Furthermore, the constraints guarantee the correctness of our many-sorted simple types.

AND	
L	`TERM`
P_{IN}	`[ x "\wedge" y ]`
M	`[ k:([x],[v],TERM) ,` `  m:([y],[w],TERM) ]`
P_{OUT}	`[ \F{"and"}[ v , w ] ]`
C_{PRE}	`{ k.type = "bool", m.type, "bool", ME.type = "bool" }`
C_{POST}	`{ }`
i	`{ true }`

The goal of processing the theories in the document is now to produce a set of invertible grammar rules for the defined notation. A classical top-down approach, that processes each definition or notation on its own, extending the grammars and recompiling the grammar before processing the next element, is far too inefficient for real time usage due to the expensive compilation process. Therefore, we use a runtime interpreter for the invertible grammar. Additionally, we try to minimize the translation steps as much as possible for a theory in a mathematical document by the following processing workflow.



1) *Phase 1*: All definitions are processed sequentially. For each definition the name of the introduced type or symbol is added to the lexica of that interface document. The scanners of the notation manager are rebuilt. We require that the names of all definitions are unique.
2) *Phase 2*: All notations are processed sequentially. For each notation the introduced patterns are analyzed and invertible grammar rules are synthesized for the invertible grammar. Additionally, the synthesized rules of the transitive closure of all inherited theories are added. The notation manager changes the invertible grammar of that interface document in the second mediation module. We require that there is in each theory at most one notation for a definition occurring in this theory or in a transitively inherited theory.
3) *Phase* 3: The theory is uploaded as the corresponding interface document in the second mediation module. The document is tokenized with the new lexicon and all contained mathematical formulas are automatically processed and translated into a semantic representation using the synthesized invertible grammar.

Note that all notation definitions in the interface document of the first mediation module are processed and removed by the notation manager. Thus the interface document of the second mediation module does not contain notation information. In the following, we will discuss each phase of this translation process with our example from Figure 33.

Processing Definitions. The author may introduce in a definition either a new type by \type{*name*} or a new symbol by \concept{*name*}[*type*]. In both cases, the *name* is added as a token to the *lexicon* (if not yet included). Furthermore, we generate a fresh *internal name* which will be the name of the type (or symbol) communicated to the proof assistance system. The alphanumerical *internal name* is added to the *lexicon* as well. Then we extend the grammar by a rule for types (or symbols) that converts *name* into that *internal name* and by automatic inversion also vice-versa. Overloading of symbol names is only allowed if their types are different.

In the definition of ∈ in our example document in Figure 33 we have the symbol declaration \concept{"\in"}["elem \times set \to bool"]. The *name* is \in and assume the internal name being IN. Both tokens are added to the scanners. Furthermore the following rule is added to the invertible grammar.

SYMBOL-IN	
L	SYMBOL
P_{IN}	["\in"]
M	[ ]
P_{OUT}	["IN"]
C_{PRE}	{ }
C_{POST}	{ }
i	{ true }

The type information of the symbol declaration is additionally processed in the second phase to generate the type constraints in the grammar rule of this operator. Our system supports many-sorted simple types.

Processing Notations. A notation defines one or more alternative notations for some symbol. The author is able to introduce local variables by \declare{x_1}, …, \declare{x_n} and use them in the patterns defining the different notations: \denote[$pattern_1$],…,\denote[$pattern_m$].

Consider the following example for the introduction of alternative notations taken from our running example in Figure 33.

```
"Let" \declare{"$x$"} "be an individual and"
\declare{"$A$"} "a set, then we write"
\denote["$x \in A$"] "," \denote["$x is element of A$"] ","
\denote["$x is in A$"] "or" \denote["$A contains x$"] "."
```

We impose an ordering in which the variables are declared by `\declare` and comply with the domain of the associated operator, that is, `x` is the first argument of `\in` and `A` the second. First of all, the *lexicon* is locally extended by the terminals for the local variables $x_1, \ldots, x_n$ – in this case the variables `x` and `A`. Then each notation pattern is tokenized by the scanner, which returns a list of tokens including new tokens for unrecognized chunks. For instance, in our example above, the scanner knows the tokens for the local variables `x` and `A` when tokenizing `"x is element of A"`. The unknown chunks are `"is"`, `"element"` and `"of"`, which are then added on the fly.

This behavior of the scanner is non-standard, but it is an essential feature to efficiently accommodate new notations. The scanner has been implemented such that it returns the longest match. A hard wired scanner cannot be used because the alphabet of the language is unknown. By using the standard scanner generation algorithm described in [Wilhelm & Maurer, 1997] a DFA is directly generated out of the given lexicon without generating a NFA first. Due to the small size of the automaton, the generation of a scanner is a relatively fast process.

A notation pattern is only accepted if all declared argument variables are recognized by the scanner, namely $x_1, \ldots, x_n$ occur in the pattern without ambiguities. For every notation pattern, the grammar is extended by a rule for function application that converts the notation pattern into the semantic function application with respect to the argument ordering. To this end we take the scanned pattern with local variables as the input pattern and map the local variables to the arguments of the semantic representation of the function application in the output pattern. The arguments are transformed with the category `TERM`. From the type provided by the symbol declaration we generate constraints that guarantee the type correctness. For instance, for the pattern `"A contains x"` we generate the following invertible grammar rule. Note that the notation pattern permutes the arguments of that operator.

OPERATOR-IN-1	
L	`TERM`
P_{IN}	`[ A "contains" x ]`
M	`[ k:([x],[v],TERM),` `  m:([A],[w],TERM) ]`
P_{OUT}	`[ \F{"in"}[ v , w ] ]`
C_{PRE}	`{ k.type = "elem", m.type = "set", ME.type = "bool" }`
C_{POST}	`{ }`
i	`{ true }`

The author is allowed to define multiple notation patterns for an operator. In this case, the grammar rule synthesized from the first pattern has the highest priority and will be used by default in the inverse translation process.

The notation manager uses the type of the defined operator to generate the precondition constraints. Thereby, type equations are added for the rule invocations of the arguments, as well as for the result of this rule. The constraint system of the invertible grammar formalism then guarantees the type correctness by continuously checking the satisfiability of the type constraints during the translation process.

Additionally, the author can define the symbol *name* to be left- or right-associative by `\left{`$name$`}` or `\right{`$name$`}`. This information is then encoded as the matching direction of the input pattern. Note that the matching direction of a pattern is by default to the right. Since there exists often more than one matching mapping because some pattern variables might match more than one partner, we order the matching mappings by the amount of matching partners for variables from high to low, and from left to right (`dir="l"`) or from right to left (`dir="r"`).

The author can define the precedence of operators by using `\prec[`$name_1$`, …,`$name_k$`]` where each $name_i$ is a symbol given in the definitions. This declares the relative precedence of these symbols, where $name_i$ is lower than $name_{i+1}$. Note that the rule precedence is inverse to the corresponding operator precedence because we use a top-down interpreter. From this partial rule order the notation manager computes a total rule order using topological sort with the rules initially ordered as in the theory.

Finally, the author can indicate a chaining operator by `\chaining{`$name$`}`, usually used for commutative predicates like equality. The notation manager then generates a special chaining rule as we will describe in the next chapter.

Processing Theories. The generated invertible grammar rules complement the grammar of the inherited theory if inheritance is used, or they complement the initial grammar if inheritance is not used. The lexicon of the invertible grammar is extended by the new tokens. Then the notation manager changes the configuration of the interface document of this theory in the second mediation module. The resulting invertible grammar is then transmitted together with the type of the interface document used by the notation manager. The mediation module then automatically translates the contained formulas to the semantic representation of the proof assistance system.

Processing Documents. A document may contain multiple theories, each of these possibly having another theory in its context, specified in the `\context` annotation. From the partial order induced by the inheritance, the notation manager computes the total order of all theories using topological sort with the theories initially ordered as in the document. Then, the theories are processed in this order as described.

8.4 Sugaring and Souring

Syntactic sugar is often added to a programming language to make it easier to use by humans without affecting the functionality or expressive power of the language. In the context of mathematical documents, we may consider for example the chaining of equalities in a formula as syntactic sugar. Robert Lamar introduced in [Kamareddine *et al*, 2007b] the inverse process that he calls *souring*. In his approach, the author has to *manually* add souring annotations to the mathematical formulas to guide the de-sugaring process which is realized by rewriting rules. In the following, we will present the souring classes that he introduced and discuss how the souring process can be *automated* using the invertible grammar formalism. By the automated inversion, we obtain automated sugaring for free.

Re-ordering. One regularly faces situations where two expressions with similar semantics are ordered differently. Consider for example the expression "`x is in A`", one can also imagine the author writing "`A contains x`" instead. We presented the solution to this case already as part of the grammar rule synthesis. By declaratively specifying the notation pattern in the document, the notation manager generates the following invertible grammar rule automatically. Thereby, the mapping of positions is encoded in the mapping of variables between the input and output pattern. The following resulting grammar rule can be compared with a nested rewrite rule with additional constraints.

OPERATOR-IN-1	
L	`TERM`
P_{IN}	`[ A "contains" x ]`
M	`[ k:([x],[v],TERM),` `  m:([A],[w],TERM) ]`
P_{OUT}	`[ \F{"in"}[ v , w ] ]`
C_{PRE}	`{ k.type = "elem", m.type = "set", ME.type = "bool" }`
C_{POST}	`{ }`
i	`{ true }`

Sharing/chaining. Mathematicians tend to aggregate equations which follow one another. This is often used for commutative infix predicate operators like equality. The author can indicate the chaining capabilities of an operator *name* in its notation definition by adding `\chaining{`*name*`}`. Then, the notation manager first verifies that the default notation of the operator is infix by analyzing the input pattern. In the case of the equality operator this would generate in addition to the regular grammar rule the following chaining rule. Note that the shared argument `y` is duplicated in the recursive rule invocations. In the inverse direction the grammar interpreter checks automatically the semantic equality of both `y`.

OPERATOR-EQ-CHAINING	
L	TERM
P_{IN}	[**x** "=" **y** "=" **z**]
M	[k:([**x** "=" **y**],[**v**],TERM), m:([**y** "=" **z**],[**w**],TERM)]
P_{OUT}	[\F{"and"}[**v** , **w**]]
C_{PRE}	{ k.type = "bool", m.type = "bool", ME.type = "bool" }
C_{POST}	{ }
i	{ true }

The rule `OPERATOR-EQ-CHAINING` does not satisfy the static check for the *inverse consistency criterion* because the pattern variable `y` occurs in both input patterns of the recursive invocations. Thus, there is no formal guarantee that always a semantically equal term is reconstructed when using the inverse translation on the result. But since we know that the content matched by `y` will be translated equally in both branches, we can admit this grammar rule as an exception of the check for the *inverse consistency criterion*. The following rule will be invoked by the recursive translation process.

EQUAL-TERMS	
L	TERM
P_{IN}	[**x** "=" **y**]
M	[(k,[**x**],[**v**],TERM), (m,[**y**],[**w**],TERM)]
P_{OUT}	[\F{"equal"}[**v** , **w**]]
C_{PRE}	{ k.type = m.type, ME.type = "bool" }
C_{POST}	{ }
i	{ true }

A single variable occurrence is translated by the following rule. The rule calls the identity translation rule `IDR` as a recursive rule invocation which returns the recognized semantic hash index value of the input variable as the value of the attribute `k.index`. Since the meta-variable `(k.index).type` can only be evaluated after the recursive rule invocations, we add a type check constraint in the set of postcondition constraints.

VAR	
L	TERM
P_{IN}	[**c**]
M	[(k,[**c**],[**v**],IDR)]
P_{OUT}	[\V{ **v** }]
C_{PRE}	{ }
C_{POST}	{ ME.type = (k.index).type }
i	{ true }

As an illustrating example we show in Figure 35 the souring parse tree for the input formula "a=b=c".

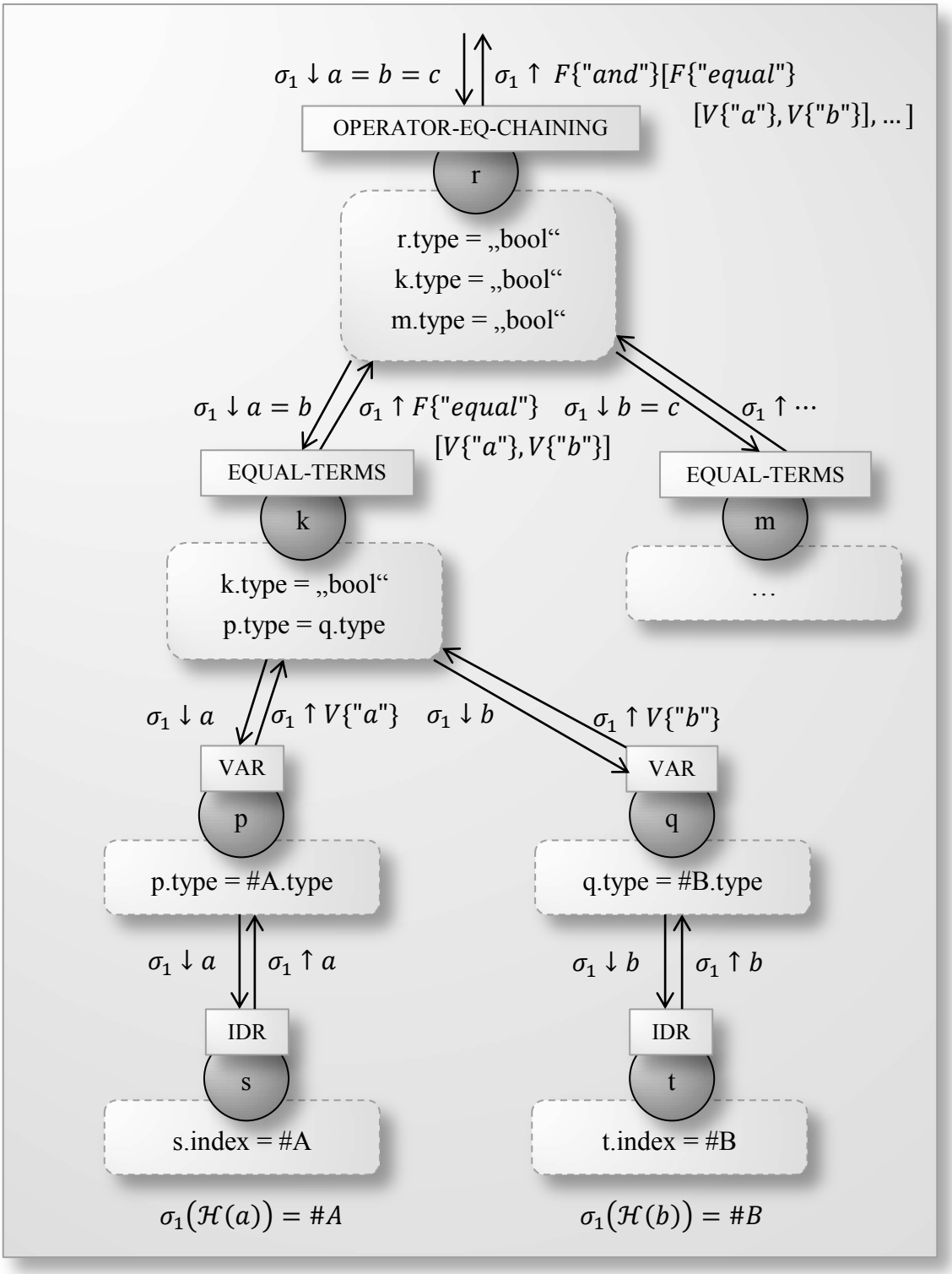


Figure 35. Souring Parse Tree

List Manipulations. The souring annotations for lists indicate how lists of bounded expressions have to be unfolded. A major use case is to handle quantification over multiple variables. Considering for example the sentence "`\forall a,b,c : R.(a+b)+c=a+(b+c)`", we would like to unfold this to a sequence of quantifications of single variables. We support multi-bindings by special grammar rules that are part of the initial grammar. The following rule is for the quantification over multiple variables.

FORALL-MULTI	
L	`TERM`
P_{IN}	`{ dir="r" }[ "\forall" c ":" t "." f ]`
M	`[ (k,[c ":" t],[v],VARINTRO),` `(m,["\forall" n ":" t "." f],[w],TERM) ]`
P_{OUT}	`[ \F{"forall"}[ \B[ v ] , w ] ]`
C_{PRE}	`{ m.type = "bool", ME.type = "bool" }`
C_{POST}	`{ }`
i	`{ false }`

The constraints are again used to guarantee type correctness and the environment is used to transport the information for identifying quantified variables. If we would not check the types, we would have to deal with lots of alternative readings, and the burden of disambiguation would be passed on to the proof assistance system.

The first quantified variable is recursively processed by the following rule which introduces a typed variable. In order to able to recognize this variable during the translation of its scope, the semantic hash value of that variable is computed, returned by the identity rule `IDI` as value of `k.index`, and added to the environment. In particular the environment is propagated along the creation of the parse tree and allows for static scoping. The type is then assigned to this variable by evaluating the meta-variable constraint `(k.index).type = m.type`. Thereby, the value of `m.type` contains the translated type of the new variable.

TYPED-VAR	
L	`VARINTRO`
P_{IN}	`[ c ":" t ]`
M	`[ (k,[c],[v],IDI),` `(m,[t],[w],TYPE) ]`
P_{OUT}	`[ \V{ v }[ w ] ]`
C_{PRE}	`{ }`
C_{POST}	`{ ME.type = m.type, (k.index).type = m.type }`
i	`{ true }`

A complete worked out example can be found in the use case in Chapter 6.4.

8.5 Management of Change

The invertible grammars generated so far by the notation manager are valid for the theories in a specific version of the document in time. When the author continues to edit the document, it may be modified in arbitrary ways, including the change of existing definitions and notations. In order to compute the modified semantic representation of the document for the proof assistance system, we need possibly to adjust the grammars and reprocess modified parts of the document. Always starting from scratch following the described procedure is not efficient and may jeopardize the acceptance by the author if that process takes too long. Therefore there is a need for management of change for the notational parts of a document and those parts that depend on them. The management of change tasks are the following:

1) First, we have to determine any modifications in the context, definitions and notations of every theory in the document.
2) Second, we have to adjust only those theory specific grammars that are affected by the determined modifications, and we have to retranslate the affected theories.
3) Third, the proof assistance system needs an optimal change description of the new semantic representation of the document.

As an illustrating example, we consider the following change to the notation of $\subset$ in our running example from Figure 32 . We define the additional notation pattern $B \supset A$.

```
\notation{"Predicate" "$\subset$"}[
  "Let" \declare{"$A$"} "and" \declare{"$B$"}
  "be sets, then we write" \denote["$A \subset B$"]
  "or" \denote["$B \superset A$"] "." ]
```

Determining notational changes. The notation manager requests an update from the first mediation module for the interface document. The mediation module uses the change graph search to compute the changes with respect to the similarity and edit specification provided by the notation manager. In this setting, the *similarity specification* is defined such that it takes into account that the ordering of all annotations is not relevant except of the content of definitions, notations and proofs. Furthermore, the notation manager does not need change descriptions in full detail, thus the *edit specification* is defined such that the computed changes are limited to the level of theories. In summary, the notation manager is efficiently informed about deleted, added and modified theories.

Thus, the notation manager uses the following specifications when requesting the update.

```
\similarity{}[
  \order{ name="body", layer="content"}[],
  \order{ name="theory", layer="content"}[],
  \keys{ name="theory", layer="content"}[ "name" ] ]
\edit{}[
  \limit{}[
    \global{ name="theory" }[] ] ]
```

The notation manager then receives a change script that replaces the theory which contains the modified notation.

```
[ \replace{ target=/body[1]/theory[1] }
    [ \theory{}[…] ] ]
```

Adjusting invertible grammars. The introduced workflow of the notation manager has to be slightly modified for the incremental case. The processing of the theories starts with the first change in the computed total order of the theories. The former synthesized grammar rules of the preserved heading theories are reused. Then, the usual workflow proceeds. Note that the grammar interpreter of the second mediation module may be able to reuse stored information of the transformation traces although the invertible grammar has been changed.

Thus, the notation manager starts with the processing of the modified theory and then needs to reprocess all succeeding theories in the total order of the theories. The second phase of processing notation leads to the generation of the following grammar rule for the additional notation pattern.

OPERATOR-SUBSET-2	
L	TERM
P_{IN}	[**B** "\superset" **A**]
M	[k:([**A**],[**v**],TERM), m:([**B**],[**w**],TERM)]
P_{OUT}	[\F{"subset"}[**v** , **w**]]
C_{PRE}	{ k.type = "set", m.type = "set", ME.type = "bool" }
C_{POST}	{ }
i	{ true }

The notation manager changes the invertible grammar of the interface document which corresponds to the modified theory. The alternative notation pattern can now be used.

Computing optimal changes. The proof assistance system requests an update from the second mediation module. The mediation module computes the changes with respect to the similarity specification and the edit specification provided by the proof assistance system. In this setting, the *similarity specification* takes into account that the ordering of all parts of the semantic representation except of the content of proofs is not relevant. The *edit specification* contains the weights of all parts of the semantic representation of the document. In particular, the weights of the theorems and axioms reflect the amount of dependent proofs and proof steps. The changes are limited by the edit specification to the level of definitions, theorems, axioms and proof steps. The mediation module computes an optimal change script which is then processed by the internal management of change component of the proof assistance system.

Thus, the notation manager uses the following specifications when requesting the update.

```
\similarity{}[
  \order{ name="body", layer="content"}[],
  \order{ name="theory", layer="content"}[],
  … ,
  \keys{ name="theory", layer="content"}[ "name" ],
  \keys{ name="theorem", layer="content"}[ "name" ],
  … ]
\edit{}[
  \weight{}[
    \delete{ path=/body[1]/theory[1]/axiom[1] weight="10" }[],
    … ],
  \limit{}[
    \global{ name="theorem" }[],
    \global{ name="axiom" }[],
    \global{ name="definition" }[],
    … ] ]
```

The proof assistance system then receives an empty change script because the content of the theory has not been changed. Note that the notation definitions are not part of the interface documents exchanged via the second mediation module.

```
[ ]
```

Another task that is usually considered part of the management of change is *refactoring*. In software engineering, all major integrated development environments support refactoring operations like renaming classes or methods. In the context of mathematical authoring, we want to provide *notational refactoring*. When the author changes the notation of a symbol or operator we ideally want the formulas being automatically adapted.

Notational refactoring. The workflow of the notation manager needs to be again slightly modified to support notational refactoring. In phase 3 of the processing of a theory, the notation manager changes the configuration of the interface document of a theory in the second mediation module by a new invertible grammar and the type of the interface document. If the content of the interface document has not changed, thus only the notation has been modified, the notation manager transmits the type of the interface document of the proof assistance system instead of its own type. Then the second mediation module in turn automatically inverse translates the formulas from the semantic representation with the new grammar rules. Finally, the notation manager propagates these changes via the first mediation module to the text-editor.

With respect to our running example, we consider the case that the notation of $\subset$ has been completely replaced by the new notation pattern.

```
\notation{"Predicate" "$\subset$"}[
  "Let" \declare{"$A$"} "and" \declare{"$B$"}
  "be sets, then we write" \denote["$B \superset A$"] "." ]
```

The notation manager would then synthesize the new grammar rule for this pattern as described before, but the grammar rule of the removed notation pattern is no longer part of the grammar anymore. When the notation manager changes the invertible grammar of the interface document corresponding to the modified theory, the mediation module automatically inverse translates the whole interface document using the provided grammar rules. Hence, the notation used in the mathematical formulas is automatically adapted in the whole document.

While notational refactoring is an operation that is only affecting the rendering of the document in the text-editor, the refactoring of the names of theories, axioms, theorems, basically any ontological unit, affects also the proof assistance system because it uses these names for reference-by-name in its internal dependency graph. In this case the system-wide consistency can only be guaranteed if one either uses an extensive locking mechanism, or if the refactoring operation becomes an atomic change operation. System-wide locking is not a desirable solution. As an intermediate step towards native refactoring, we started investigating the benefits of methods for *ontology-driven refactoring* [Müller & Wagner, 2007]. The idea is to detect (partial) refactoring operations in a change script, report inconsistencies and propose repair operations.

8.6 Ambiguity Resolution

In order to enable a document-centric approach for formalizing mathematics, the added-values offered by the authoring environment must outweigh the additional burden imposed to the author. In the following we present techniques to reduce the amount of ambiguities for the formalization process by exploiting the theory structure contained in a document. In addition to that, our approach offers the added-value of redefining notations for different *communities of practice*.

Reducing Ambiguities. In a mathematical document, the logical context of a formula is determined by the theory it occurs in. Therefore, the different parts of a document are assigned to specific theories. New theories can be defined inside a document and built on top of other theories. The notion of a theory as a collection of mathematical knowledge with an inheritance hierarchy has been developed by OMDOC [Kohlhase, 2006] respectively by development graphs ([Hutter, 2000], [Autexier & Hutter, 2005], [Mossakowski *et al*, 2006]). With the inverse multiplexing process, the notation manager sets up for each theory an interface document with its own invertible grammar. By maintaining different grammars for different theories we avoid some ambiguities that would arise when sticking to have a single grammar. The benefits of this structured approach are that the author can use the same notation pattern in parallel in any non-interfering theories, and that inherited notation can be redefined in a new theory to prevent possibly arising ambiguities.

Consider for example a theory of the integers with multiplication denoted by "$x \times y$" and a completely unrelated theory about sets with the same notation for Cartesian products. This is typically a source of ambiguities that would require the use of type information to resolve the issue. Using different grammars for different theories completely avoids that problem. However, what if the theory about sets inherits the theory of the integers to produce Cartesian products of sets of integers? The ambiguities may be resolved by the type checks encoded in the constraints, as shown for example in the following grammar rule:

OPERATOR-INTMULT-1	
L	`TERM`
P_{IN}	`[ x "\times" y ]`
M	`[ k:([x],[v],TERM),` `m:([y],[w],TERM) ]`
P_{OUT}	`[ \F{"intmult"}[ v , w ] ]`
C_{PRE}	`{ k.type = "int", m.type = "int", ME.type = "int" }`
C_{POST}	`{ }`
i	`{ true }`

Redefining Notations. When inheriting a theory, we want to reuse the formal content, but possibly adapt the notation used to write formulas. This scenario occurs less frequently with a single author but more often if the work is shared among a group of colleagues who want to continue this work but with their own notation. In order to technically support the redefinition of notations, we modify again phase 3 of the theory processing workflow. If the current theory inherits another theory, only those grammar rules are inherited that do not refer to a definition for which there exists a processed notation in the current theory. In combination with notational refactoring, this feature can be used to translate the formulas in a specific theory and all dependent theories from one notation to another one, without affecting the formulas in the inherited theories.

Continuing with our running example, another author starts writing down a new theory that inherits the theory in our example. Assume the author prefers the notation $A \sqsubset B$ for the subset relationship, then the notation can be redefined as follows in the new theory.

```
\notation{"Predicate" "$\subset$"}[
  "Let" \declare{"$A$"} "and" \declare{"$B$"}
  "be sets, then we write" \denote["$A \sqsubset B$"] "." ]
```

Note that the notation is linked to the definition by its name. In this example, the author can use now the preferred notation in the new theory. Furthermore, all formulas with the former notation in the current theory are automatically adapted by notation refactoring.

Communities of Practice. A *notation practice* is the selection of an adequate presentation for symbols. In this sense, an author's notation practice is her individual way of selecting her notation, which she has acquired, and which is influenced by a number of factors. Watt and Smirnova introduce possible reasons for multiple notations of the same mathematical concept, namely *area of application*, *national conventions*, *level of sophistication*, the *mathematical context* and the *historical period* [Smirnova & Watt, 2006]. The eLearning environment ACTIVEMATH [Melis *et al*, 2009] distinguishes similar categories but with additional dimensions. In [Kohlhase & Kohlhase, 2006] Kohlhase proposes the application of the economic theory of communities of practice [Wenger, 2005] to the area of mathematics. According to their discussions, mathematical practice is inscribed into documents, for example by selecting specific notations or referencing other mathematical publications. In [Wagner & Müller, 2007] we outline an extension of our approach which aims at identifying and analyzing the notation practice of individual authors in order to support mathematical communities of practice.

8.7 Discussion

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered in an assisted authoring environment must outweigh the additional burden imposed on the author compared to the amount of work for a non-assisted preparation of a document. One step in that direction is to give the freedom to define and use her own notation inside a document back to the author. In this chapter we presented a mechanism that enables the author to define her own notation in a natural way in the text-editor TEXMACS while being able to get support from the proof assistance system, such as type checking, proof checking, interactive and automatic proving.

The notations are used to parse formulas written by the user in the LaTeX-style she is used to, as well as to render the formulas produced by the proof assistance system. Ambiguities are reduced by assigning a separate invertible grammar to each defined theory in the document and by integrating a weak form of type-checking in the constraints of the grammar rules to resolve remaining ambiguities during the parsing process. The structure of theories also forms the basis to inherit notations defined in other theories. In the following, we will compare our approach [Autexier *et al*, 2007] to related work.

Related Work. Supporting specific mathematical notations is a major concern in all proof assistance systems. With respect to supporting the definition of new notations that are used for type-setting, the systems ISABELLE [Nipkow *et al*, 2002] and MATITA [Asperti *et al*, 2006] are closest. ISABELLE comes with type-setting facilities of formulas and proofs for LaTeX and supports the declaration of the notation for symbols as prefix, infix, postfix and mixfix. Furthermore, it allows the definition of translations which are close to our style of defining notations. The main differences are that the notation is not defined in the LaTeX document but has to be provided in the input files of ISABELLE. Since the input and output document of ISABELLE are different documents, there is no need for mechanisms to efficiently deal with modifications of the notation, which is crucial in our setting.

In the context of MATITA, Padovani and Zacchiroli proposed a mechanism of *abstraction* and *rendering* parsers [Sacerdoti Coen & Zacchiroli, 2004] that are created from notational equations which are comparable to the grammar rules we generate from the notational definitions. Their mechanism is mainly devoted to obtain MATHML [W3C, 2001] representations where a major concern also is to maintain links to the internal objects. Similar to ISABELLE, the notation must be provided in input files of MATITA that are separated from the actual document. Also, they do not consider the effect of changing the notation and to efficiently adjust the parsers. However, they concentrated on the development of a sophisticated disambiguation procedure [Sacerdoti Coen & Zacchiroli, 2008].

In mathematical textbooks, one observes that the authors start using a chaining style of notation after having proved the commutativity of a predicate operator. Brackets are often omitted when the associativity of an operator is proved. This level of support for notation requires reasoning about the mathematical knowledge encoded in the document, basically it requires the expert knowledge of the proof assistance system. This is not compatible with our goal of developing a general mediation framework. We refer to the approach of the system SAD [Verchinine *et al*, 2008], which performs the notational processing of the document totally inside the proof assistance system.

Additionally, we note that our approach inspired the birth of similar methods in OMDoc, namely a toolset for supporting notation-aware interactive documents [Kohlhase *et al*, 2009]. Furthermore, the idea of a self-extensible pattern-based language has been taken to an extreme by Roman Knöll who realized a new self-extensible pattern-based programming language Π [Knöll & Mezini, 2009] where a pattern is simply speaking an EBNF-expression with an associated meaning.

Conclusion. As a first step to reduce the burden of completely annotating the mathematical document in the text-editor, we presented in this chapter a formula language with self-extensible notation. The author is able to define the notation for formulas as patterns inside the theories of a document. We use a structured approach for theories in order to support the reuse of mathematical knowledge and to reduce ambiguities of overloaded operators. We presented a transformation pipeline for generating an invertible grammar from notation patterns defined in the document. This grammar can be used to translate the formulas in a theory into their semantic representation, as well as to render modifications of the semantic representation. The notational patterns allow for prefix, infix, postfix and mixfix operators, as well as pseudo natural language patterns.

We have to admit that the expressive power of the pattern language for formulas is limited in a sense, for example the pattern "*x is in A*" does not yet automatically add a rule for "*x and y are in A*". However, the support for natural language aggregation may be hand-crafted by designing aggregation rules similar to the presented rules for the automated souring of lists of bounded elements.

We limited the theory inheritance to the theories in the same document because a critical problem of all approaches with libraries is the handling of concurrent changes to documents in the library. Clearly, these changes may invalidate the notation used as well as the correctness of the document from a logical point of view. The only "*solution*" to this problem, to the best of our knowledge, has been presented by Klaus Grue with the LOGIWEB system [Grue, 2007]. In his system, documents in the library are immutable by definition, thus he eliminates the need for dealing with changes.

9 Authoring Exercise Solutions

Aiming at a document-centric approach to formalize and verify mathematics we integrated the proof assistance system ΩMEGA with the scientific text-editor TEXMACS. The author writes her mathematical document entirely inside the text-editor in natural language with formulas in LaTeX style, enriched with semantic annotations. Assuming the author has written down the lecture notes and stated some exercises, which are essentially theorems that need to be proven. Then, we discuss in this chapter the authoring of exercise solutions, that is, the authoring of proofs for theorems.

In the last chapter, we presented a self-extensible formula language that reduces the burden to annotate the formulas in the document. We will now take a step further and introduce a *controlled mathematical language*. The author now has the choice to either use natural language with the need to provide the required semantic annotations, or to use the controlled mathematical language without the need for further annotations. Since we propose the controlled language as a conservative extension, both options can of course be used in combination within the same document. Our main focus in this chapter will be on the part of the annotations and of the controlled language that deals with the proofs.

After this discussion of the authoring process, we will describe how proof obligations are generated from an annotated proof or a proof written in the controlled mathematical language respectively, and we will describe how this process can be modeled incrementally using the semantic change computation of the mediation modules. The task of the proof assistance system is the verification of the semantic representation. This includes the correction or completion of proofs, which includes the generation of document fragments written in the controlled mathematical language. To this purpose, we inverse translate the modified semantic representation with an invertible grammar. Note that not all logically verifiable proof steps are acceptable in an exercise solution. For example, directly deriving the conclusion in the first proof step is most likely a too coarse grained proof step. For a detailed discussion of proof step granularity we refer to the work of Marvin Schiller in [Schiller *et al*, 2008] and [Schiller, 2010].

Finally, we integrate the feedback of the proof assistance system with the document in the text-editor. We will propose a *feedback language* as part of the invertible grammar used in the transformation pipeline. Furthermore, we interact with the proof assistance system by writing queries within the document in the text-editor. Thus, we will propose an *interaction language* as part of the invertible grammar. Altogether, the feedback and the interactive queries will be first-class citizens of the document in the text-editor.

9.1 Controlled Mathematical Language

Let us first introduce an example of the kind of documents we want to support. Figure 36 shows a theory about Simple Sets written in the text-editor TEXMACS. This theory defines two base types and several set operators together with their axioms and notation. In general, theories are built on top of other theories and may contain definitions, notation, axioms, lemmas, theorems and proofs. Note that in this example, set equality is written as an axiom because equality is already defined in the base theory.

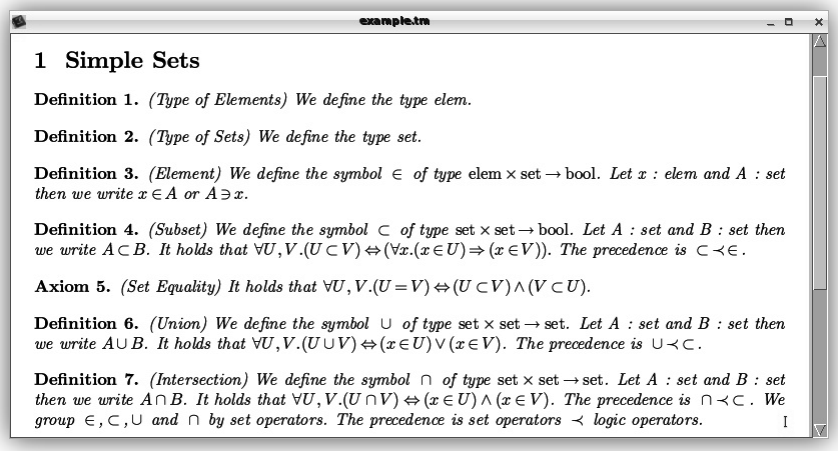


Figure 36. TEXMACS Document with Controlled Mathematical Language

In the approach presented in the previous chapter, the author had full freedom in writing her document but had to manually provide semantic annotations. We now introduce a *controlled mathematical language* to skip the burden of providing these annotations.

Our aim in this thesis is not to carve the controlled mathematical language in stone. Indeed, we believe that this language should be designed together with the target audience, the respective mathematical community. Our goal is the development of a framework for designing and evaluating a controlled mathematical language. Furthermore, the author should be able to use both approaches in combination, semantic annotations and a controlled mathematical language, within the same document.

We propose the controlled mathematical language defined in Table 48 and Table 49 which satisfies the minimal requirements for a rapid authoring process. The invertible grammar is presented in the compact representation to provide a better overview. After the introduction of the controlled mathematical language, we will focus on a concrete example with the invertible grammar rules for proof steps. These rules are the foundations for the authoring of exercise solutions in our scenario.

Head	Production	Creation
DOC	[***THY****]	
THY	[**v:*TNAME*** **y:*CTX*?** **w:*THYC***]	[**v , y , w**]
TNAME	[\section{x:***NAME***}[]]	[\name{ **x** }]
CTX	["We" "use" **x:*NAME*** "."]	[\context[\R[**x**]]]
THYC	(***DEF***|***AXM***|***TEO***|***PRF***)*	
DEF	[\definition{ **x:*NAME*** } [**y:*DEFC***]]	[\definition{ **x** }[**y**]]
DEFC	[(***DEFT***|***DEFS***) ***NOTC***? ***ALTC***? ***SPEC****]	
DEFT	["We" "define" "the" "type" **x:*NAME*** "."]	[\type{ **x** }]
DEFS	["We" "define" "the" "symbol" **x:*NAME*** "of" "type" **y:*TYPE*** "."]	[\concept{ **x** }[**y**]]
NOTC	["Let" **x:*TVARS*** "then" "we" "write" **y:*PATS*** "."]	[**x , y**]
TVARS	[**x:*TVAR***]	[**x**]
TVARS	[**x:*TVAR*** "and" **y:*TVAR***]	[**x , y**]
TVARS	[**x:*TVAR*** "," **y:*TVARS***]	[**x , y**]
TVAR	[**x:*VAR*** ":" **y:*TYPE***]	[\declare{ **x** }[**y**]]
VAR	["$" x:***ID*** "$"]	[\V{ **x** }]
PATS	[**x:*PAT***]	[**x**]
PATS	[**x:*PAT*** "or" **y:*PAT***]	[**x , y**]
PATS	[**x:*PAT*** "," **y:*PATS***]	[**x , y**]
PAT	["$" **x:*ID*** "$"]	[\denote[**x**]]
SPEC	[***GROUP***|***PREC***|***ASSOC***|***CHAIN***]	
GROUP	["We" "group" **x:*NAMES*** "by" **y:*NAME*** "."]	[\group{ **y** }[**x**]]
PREC	["The" "precedence" "is" **x:*NAME*** "≺" **y:*NAME*** "."]	[\prec[**x , y**]]
ASSOC	[***ASSOCR***|***ASSOCL***]	
ASSOCR	["The" "operator" **x:*NAME*** "is" "right-associative" "."]	[\right{ **x** }]
ASSOCL	["The" "operator" **x:*NAME*** "is" "left-associative" "."]	[\left{ **x** }]
CHAIN	["The" "operator" **x:*NAME*** "can" "be" "chained" "."]	[\chaining{ **x** }]
NAMES	[**x:*NAME***]	[**x**]
NAMES	[**x:*NAME*** "and" **y:*NAME***]	[**x , y**]
NAMES	[**x:*NAME*** "," **y:*NAMES***]	[**x , y**]
NAME	[***ID***]	

Table 48. Grammar of the Controlled Mathematical Language (Part 1)

The grammar rules in Table 48 describe the sublanguage which is allowed to be used in the content of definitions. This sublanguage allows for introducing new symbols and types together with notation patterns and additional parsing information. This is an alternative to the semantic annotations for definitions presented in Chapter 8.

Table 49 shows the grammar rules for axioms, theorems and in particular for proofs and proof steps. Note that the subproofs are linked to the subgoal or case they belong to by referring to the label of that formula.

Head	Production	Creation
AXM	[\axiom{ **x:*NAME*** }[**y:*ALTC***]]	[\axiom{ x }[y]]
TEO	[\theorem{ **x:*NAME*** } [**y:*ALTC***]]	[\theorem{ x }[y]]
ALTC	["It" "holds" "that" **x:*FORM*** "."]	[\conclusion[x]]
FORMS	[**x:*FORM***]	[**x**]
FORMS	[**x:*FORM*** "and" **y:*FORM***]	[**x , y**]
FORMS	[**x:*FORM*** "," **y:*FORMS***]	[**x , y**]
FORM	["(" **x:*LABEL*** ")" "$" **y:*TERM*** "$"]	[\formula{ **x** }[**y**]]
FORM	["$" **x:*TERM*** "$"]	[\formula{}[**x**]]
PRF	[\proof[**x:*STEPS***]]	[\proof{}[**x**]]
STEPS	[((***OSTEP STEPS***)|***CSTEP***)?]	
OSTEP	[(***SET***|***ASS***|***FACT***|***GOAL***|***CGOAL***)]	
CSTEP	[(***GOALS***|***CASES***|***CGOALS***|***TRIV***)]	
TRIV	["Trivial" **x:*BY*** **y:*FROM*** "."]	[\trivial[**x , y**]]
SET	["We" "define" **x:*FORM*** "."]	[\set[**x**]]
ASS	["We" "assume" **x:*FORMS*** **y:*BY*** **s:*FROM*** "."]	[\assumption[**x , y , s**]]
FACT	["It" "follows" "that" **x:*FORMS*** **y:*BY*** **s:*FROM*** "."]	[\fact[**x , y , s**]]
GOAL	["We" "have" "to" "prove" **x:*FORM*** **y:*BY*** **s:*FROM*** "."]	[\goal[**x , y , s**]]
GOALS	["We" "have" "to" "show" **x:*FORMS*** **y:*BY*** **s:*FROM*** "." **t:*SPRFS***]	[\subgoals[**x , y , s , t**]]
CASES	["We" "have" "the" "cases" **x:*FORMS*** **y:*BY*** **s:*FROM*** "." **t:*SPRFS***]	[\cases[**x , y , s , t**]]
CGOAL	["We" "have" "to" "prove" **x:C*FORM*** **y:*BY*** **s:*FROM*** "."]	[\decompose[**x , y , s**]]
CGOALS	["We" "have" "to" "show" **x:*CFORMS*** **y:*BY*** **s:*FROM*** "." **t:*SPRFS***]	[\complex[**x , y , s , t**]]
CFORMS	[**x:*CFORM***]	[\comp[**x**]]
CFORMS	[**x:*CFORM*** "and" **y:*CFORM***]	[\comp[**x**] , \comp[**y**]]
CFORMS	[**x:*CFORM*** "," **y:*CFORMS***]	[\comp[**x**] , **y**]
CFORM	[**x:*FORM*** "assuming" **y:*FORMS***]	[\assume[**y**], \obtain[**x**]]
SPRFS	[***SPRF****]	
SPRF	["We" "prove" **x:*LABEL*** "." **y:*STEPS***]	[\proof{ **x** }[**y**]]
BY	["by" **x:*NAME*** "in" **y:*NAME***]	[\by[\R{ **x** }[**y**]]]
BY	["by" **x:*NAME***]	[\by[\L{ **x** }]]
BY	[ ]	[ ]
FROM	["from" **x:*LABELS***]	[\from[**x**]]
FROM	[ ]	[ ]
LABELS	[**x:*LABEL***]	[\L{ **x** }]
LABELS	[**x:*LABEL*** "and" **y:*LABEL***]	[\L{ **x** } , \L{ **y** }]
LABELS	[**x:*LABEL*** "," **y:*LABELS***]	[\L{ **x** } , **y**]
LABEL	[*ID*]	

Table 49. Grammar of the Controlled Mathematical Language (Part 2)

We will now take a closer look at the interplay between the controlled mathematical language and the semantic annotation language. Thereby, we focus on the invertible grammar rules for the proof step `FACT`. The following grammar rule deals with all sentences corresponding to this proof step in the controlled mathematical language, for example the sentence "*It follows that* $x = y = z$".

FACT-CML	
L	`FACT`
P_{IN}	`[ "It" "follows" "that" x "." ]`
M	`[ k:([x],[v],FORMS) ]`
P_{OUT}	`[ \fact[ v ] ]`
C_{PRE}	`{ }`
C_{POST}	`{ }`
i	`{ true }`

In combination with the chaining rule for =, we obtain the following semantic representation for the input sentence "*It follows that* $x = y = z$".

```
\fact[ \F{"and"}[\F{"equal"}[\V{"x"},\V{"y"}],
                 \F{"equal"}[\V{"y"},\V{"z"}]]]
```

Alternatively, the author could have written "`\fact[`*Thus, we have* $x = y = z$.`]`" by using the semantic annotation language. Then, the following grammar rule for the semantic annotation language would have produced the same semantic representation.

FACT-ANNOTATION	
L	`FACT`
P_{IN}	`[ \fact[ z1 , x , z2 ] ]`
M	`[ k:([x],[v],FORMS) ]`
P_{OUT}	`[ \fact[ v ] ]`
C_{PRE}	`{ }`
C_{POST}	`{ }`
i	`{ true }`

Note that the variable specification has been specified such that the pattern variables `z1` and `z2` are not allowed to match the token "`$`". This effectively reduces the combinatorial matching possibilities of the input pattern, because the pattern variables `z1` and `z2` cannot match formulas. Since these variables are not used in the rule invocations, the complete input mapping `z1->["Thus" "," "we" "have"]` and `z2->["."]` is only stored in the transformation trace. This allows for reconstructing the original annotated sentence in the inverse translation process.

Let us now consider the inverse translation of the following semantic representation produced by the proof assistance system.

```
\fact[ \F{"and"}[\F{"equal"}[\V{"z"},\V{"x"}],
                \F{"equal"}[\V{"x"},\V{"y"}]]]
```

First, there is the question which grammar rule to use, either the following inverted controlled language rule or the inverted semantic annotation rule.

FACT-CML	
L	FACT
P_{IN}	[\fact[**v**]]
M	[k:([**v**],[**x**],FORMS)]
P_{OUT}	["It" "follows" "that" **x** "."]
C_{PRE}	{ }
C_{POST}	{ }
i	{ true }

FACT-ANNOTATION	
L	FACT
P_{IN}	[\fact[**v**]]
M	[k:([**v**],[**x**],FORMS)]
P_{OUT}	[\fact[**z1 , x , z2**]]
C_{PRE}	{ }
C_{POST}	{ }
i	{ true }

Since the annotation rule can only be successfully processed if a transformation trace can be reused, the controlled language rule is the better default choice in this case. The order of the rules in the invertible grammar reflects this preference.

The second choice point is about the use of the chaining rule. Do we prefer the generated sentence "*It follows that* $z = x = y$." or "*It follows that* $z = x$ *and* $x = y$.". Intuitively, one would answer that it depends on the context. Indeed, our exploratory study indicates that this choice depends on personal preferences.

We propose to address this problem in the notation manager by comparing separately for every theory in a document the amount of formulas written in chaining and non-chaining style. Then, we encode this information into the rule precedence between the chaining and non-chaining rules. By default, the chaining rules have a higher priority. Hence, when the author starts a proof in a theory with a formula written in chaining style, the next proof step produced by the system will be rendered in chaining style, too.

9.2 Incremental Proof Verification

Let us continue our initial example from Figure 36 with the theory in Figure 37 that inherits the previous one and that states a theorem about the distributivity of ∩, which the author already started to prove.

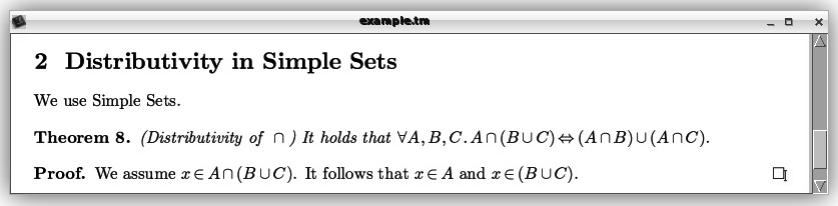


Figure 37. Theorem with partial proof

Note that a proof is implicitly related to the last stated theorem previous to this proof in the document. The partial proof in this example is written in our controlled mathematical language as follows.

```
\proof[
  "We assume" "$x \in A \cap (B \cup C)$" "."
  "It follows that" "$x \in A$" "and" "$x \in (B \cup C)$" "." ]
```

The invertible grammar interpreter translates this proof to the following semantic representation for the proof assistance system ΩMEGA.

```
\proof{}[
  \assumption[ \F{"in"}[\V{"x"},\F{"cap"}[\V{"A"},
                                  \F{"cup"}[\V{"B"},\V{"C"}]]] ],
  \fact[ \F{"and"}[\F{"in"}[\V{"x"},V{"A"}],
                   \F{"in"}[\V{"x"},\F{"cup"}[\V{"B"},\V{"C"}]]] ]]
```

The verification service of ΩMEGA tries to reconstruct a high-level proof plan in its internal proof data structure ($\mathcal{PDS}$) according to the proof steps in the document. This proof plan is then expanded to the verifiable assertion level with a resource-bounded search. Thus, a proof step is usually justified by a sequence of lower level proof steps, which turns the proof data structure into a hierarchical data structure.

Figure 38 shows the reconstructed proof plan for our example. The tasks of the proof verification are shown as oval boxes connected by justifications, where the squared boxes indicate which type of proof step has been applied.

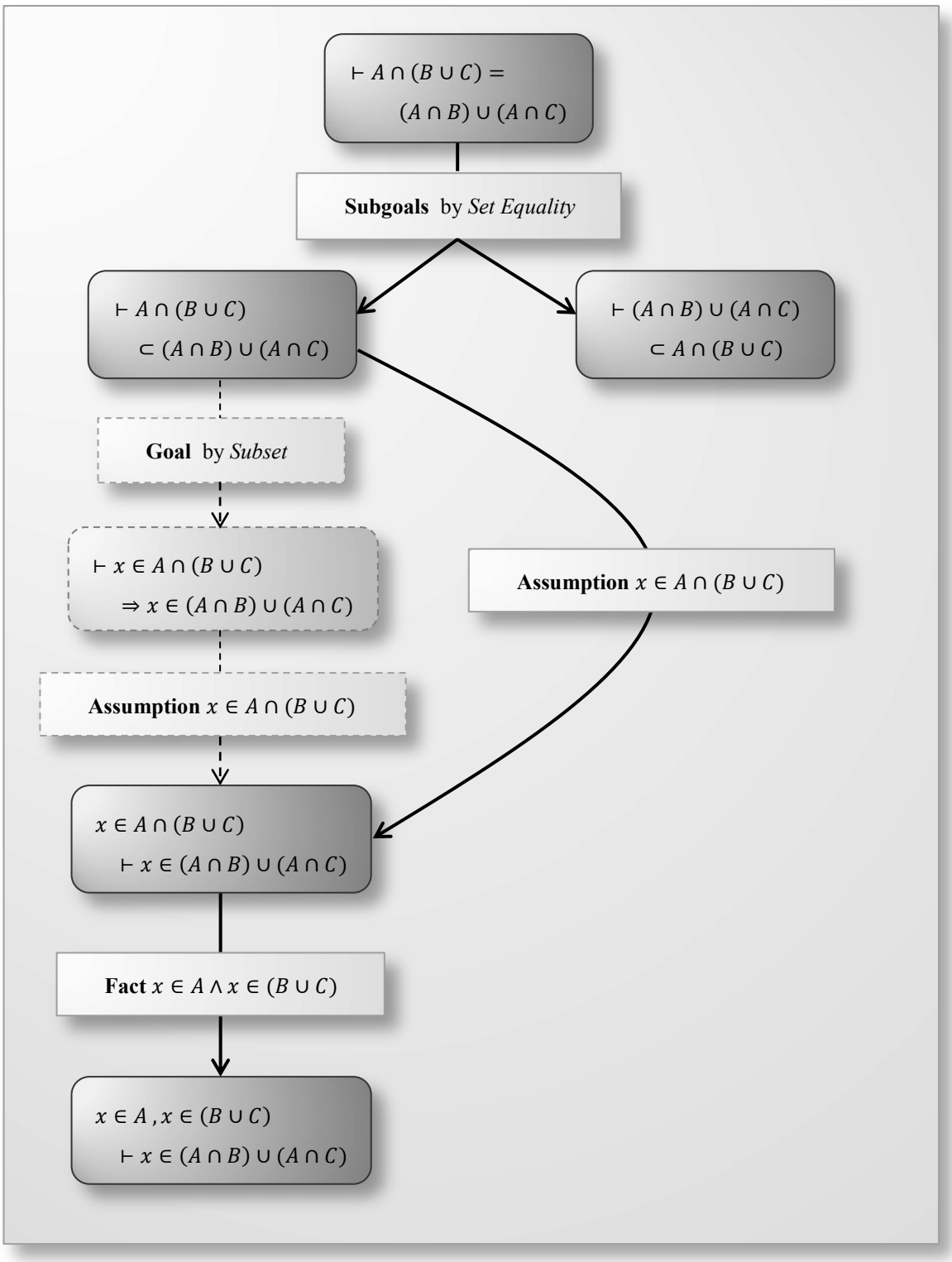


Figure 38. Repaired Partial Proof in ΩMEGA

The verification of a single proof step can become time consuming if some information is underspecified. In the worst case a complete proof search has to be performed. To obtain adequate response times, a given proof step is checked by ΩMEGA in two phases. First, a quick check is performed, where ΩMEGA checks with a simple matching algorithm whether the given proof step can be justified by a single inference application. If the test succeeds, the step is sound, as the inference application method used by ΩMEGA is proved to be correct.

If it is not possible to justify the proof step with a single inference application, a more complex proof repair mechanism is started. This mechanism tries to derive the missing information needed to justify the given proof step by performing a heuristically guided resource bounded search. Otherwise, ΩMEGA reports a failure.

Looking at our running example, the user wanted to prove the theorem and started already with a partial proof. As none of the proof checking rules for *assumption* is applicable, the proof plan repair mode of ΩMEGA is started. The repair method tries to further refine the goal $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ to construct a situation in which the *assumption* step is applicable. Indeed, after two refinements the proof step becomes applicable (see Figure 38). The proof repair revealed a second subgoal which is now lifted to the semantic representation of the interface document used by the mediation module.

Besides that, the hierarchy of higher level and lower level proof steps in a proof plan is shown in this example by the expansion of the *assumption* proof step which is justified by the dashed sequence of a *goal* and an *assumption* proof step.

Considering our running example, the verification service of ΩMEGA returns the following modified semantic representation.

```
\proof{}[
 \subgoals[
  "(1)":\F{"subset"}[\F{"cap"}[\V{"A"},\F{"cup"}[\V{"B"},\V{"C"}]],
                      \F{"cup"}[\F{"cap"}[\V{"A"},\V{"B"}],
                                \F{"cap"}[\V{"A"},\V{"C"}]]],
  "(2)":\F{"subset"}[\F{"cup"}[\F{"cap"}[\V{"A"},\V{"B"}],
                                \F{"cap"}[\V{"A"},\V{"C"}]],
                     \F{"cap"}[\V{"A"},\F{"cup"}[\V{"B"},\V{"C"}]]],
  \by[ \L{"Set" "Equality"} ],
  \proof{"(1)"}[
    \assumption[ \F{"in"}[\V{"x"},\F{"cap"}[\V{"A"},
                                \F{"cup"}[\V{"B"},\V{"C"}]]] ],
    \fact[ \F{"and"}[\F{"in"}[\V{"x"},V{"A"}],
                  \F{"in"}[\V{"x"},\F{"cup"}[\V{"B"},\V{"C"}]]]]]]]
```

Finally, this new semantic representation is inverse translated by the mediation module. Since all former children of the proof have been removed, the only preserved subtree is in this case the proof subtree. Thus, the inverse translation cannot exploit any information stored in the transformation trace and has to inverse translate the semantic representation from scratch. This process computes the following interface document for the text-editor.

```
\proof[
  "We have to show" "(1)" "$A \cap (B \cup C) \subset (A \cap B)
  \cup (A \cap C)$" "and" "(2)" "$(A \cap B) \cup (A \cap C)
  \subset A \cap (B \cup C)$" "by Set Equality" "."
  "We prove" "(1)" "."
  "We assume $x \in A \cap (B \cup C)$" "."
  "It follows that" "$x \in A$" "and" "$x \in (B \cup C)$" "." ]
```

The plugin in the text-editor requests an optimal change script from the mediation module. Since there are no hidden costs or limitations when changing the document in the text-editor, we use the default *edit specification* with uniform costs and no limitations. Furthermore, we have to assume that the document in the text-editor does not contain any semantic annotations. Since the order of the concrete syntax is in general relevant, we use the default *similarity specification* which defines every tree layer to be ordered and to have no keys. The repaired partial proof is then patched in the document in the text-editor TEXMACS and rendered as shown in Figure 39.

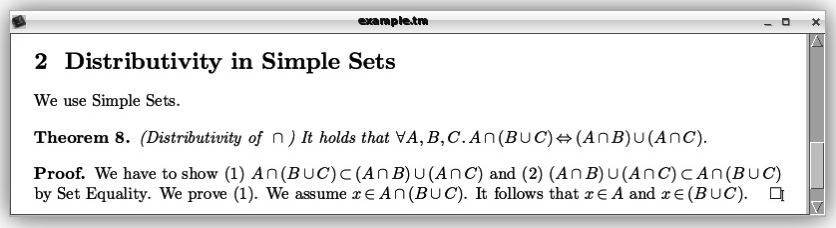


Figure 39. Theorem with repaired partial proof

The incremental proof verification now proceeds as follows: When the author modifies the document, the content is automatically translated by the mediation modules to a new semantic representation for the proof assistance system. ΩMEGA requests an optimal change script from the mediation module and tries to verify or repair the modifications to the proof plan. This may lead to modifications of the semantic representation which are propagated to the document in the text-editor as just described. And the cycle continues.

9.3 Feedback Integration

Our worked out example always used screenshots taken from our client integration with the scientific text-editor TEXMACS to show the effects on the rendered document. Besides the integration with TEXMACS, we have a prototypical integration with MS WORD, developed by Thorsten Hey [Hey, 2009]. Figure 40 shows the same mathematical theory authored with system support in both text-editors.

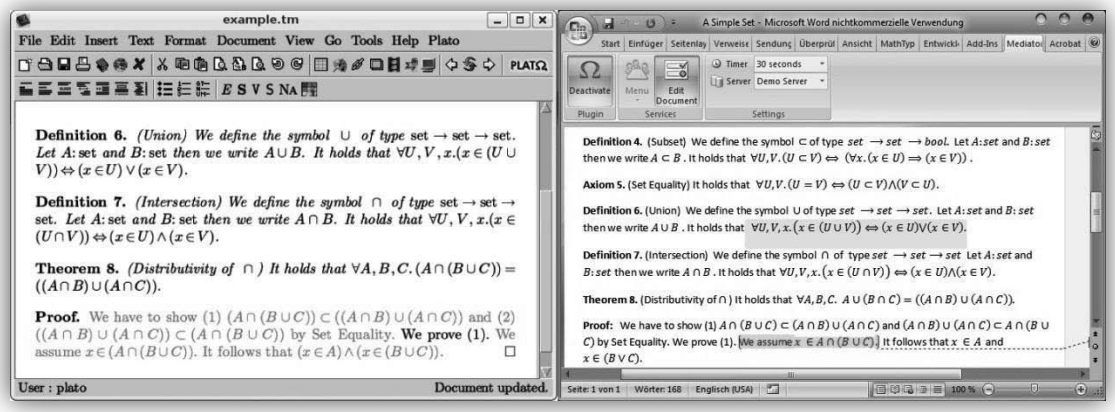


Figure 40. Feedback Integration with TEXMACS and MS WORD

We proposed the authoring of exercise solutions with a combination of semantic annotations and a controlled language. Since the authoring with any kind of controlled language is clearly supported by both text-editors, we will briefly discuss the technical realization of semantic annotations.

Realization of Semantic Annotations. In TEXMACS we use macros to represent semantic annotations. We have written a style file with an identity macro for every element in the semantic annotation language. The author has to place an annotation macro around the content she wants to annotate. The advantage of this approach is that annotations can be visualized in arbitrary ways by switching the style file. The drawback is that annotations are invisible boxes in the document that handicap the cursor movement. Due to the fundamental rectangular style of these boxes, the layout is sometimes distorted.

In MS WORD we overload the comment feature for annotating the document. The author has to comment the content she wants to annotate with the name of the annotation. Unfortunately, there is a technical limitation for the visualization of nested annotations.

The lesson learned is that a robust solution for the realization of semantic annotations requires the means to represent annotations persistently as part of the document format of the text-editor.

Services and Feedback. The verification service of the proof assistance system ΩMEGA returns the information about the verification status of proof steps by attributing the corresponding element in the semantic representation as follows.

```
\fact{ "verified" }
   [ \F{"and"}[\F{"in"}[\V{"x"},V{"A"}],
               \F{"in"}[\V{"x"},\F{"cup"}[\V{"B"},\V{"C"}]]]] ]
```

In order to visualize this feedback of the proof assistance system, we propose to integrate *feedback annotations* to the proof steps in the semantic annotation language. For example, the inverse grammar rule of the fact proof step can be modified as follows to generate this proof step in the controlled language but augmented with a semantic feedback annotation.

FACT-CML-FEEDBACK	
L	FACT
P_{IN}	[\fact{ **w** }[**v**]]
M	[k:([**v**],[**x**],FORMS), m:([**w**],[**y**],ID)]
P_{OUT}	[\fact{ **y** }["It" "follows" "that" **x** "."]]
C_{PRE}	{ }
C_{POST}	{ }
i	{ true }

The incremental interpreter then generates the following fragment for the text-editor.

```
\fact{ "verified" }
   [ "It follows that" "$x \in A$" "and"
     "$x \in (B \cup C)$" "." ]
```

The macro evaluation of this proof step can be defined in arbitrary ways. The contained sentence can be for example colored according to the verification status, or decorated with (✓) for a valid step, (⌛) for a step being processed, or (✗) for an invalid step.

In order to integrate the feedback annotation with the grammar rules of the semantic annotation language, we have to consider possible side-effects on the reuse of the transformation trace for generating the custom content in the inverse translation process. This is not important for the controlled language since there is no custom content in this case. In order to be able to reuse the transformation trace, the annotated content has to be processed by a separated grammar rule. The reason is that if the verification status changes, the semantic representation of the whole proof step does not match anymore with the stored transformation trace, but the semantic representation of the content still matches.

9.4 Interactive Authoring

During the authoring of exercise solutions, the student might get stuck with the proof. Besides the passive integration of the verification services of the proof assistance system with visual feedback, we propose an active integration by an *authoring integrated query language (AIQL)*. The idea of this integration is to extend the controlled mathematical language in order to enable the author to formulate an in-place request in the document like "*Prover: Explain.*", "*Prover: Summarize.*" "*Prover: Complete.*", "*Prover: Help.*" or for example "*Prover: Apply Set Equality.*". The proof assistance system retrieves this request as part of a change script and replaces the request by the computed results. This approach to interactive authoring with the proof assistance system does neither require additional methods in the interface protocol nor a bookkeeping of the relation between elements in the different representations. We extend the controlled mathematical language as shown in Table 50.

Head	Production	Creation
OSTEP	[(***SET***|***ASS***|***FACT***|***GOAL***|***CGOAL***) ***AIQ***?]	
CSTEP	[(***GOALS***|***CASES***|***CGOALS***|***TRIV***) ***AIQ***?]	
AIQ	["Prover:" **x:*CMDS***]	[\aiq[**x**]]
CMDS	[***CMD CMDS***?]	
CMD	[(***EXP***|***SUM***|***CPL***|***HLP***|***APP***)]	
EXP	["Explain" "."]	[\explain{}]
SUM	["Summarize" "."]	[\summarize{}]
CPL	["Complete" "."]	[\complete{}]
HLP	["Help" "."]	[\help{}]
APP	["Apply" **x:*NAME*** "."]	[\apply{ **x** }]
NAME	[*ID*]	

Table 50. Grammar of the Authoring Integrated Query Language

Considering our running example, the author asks the explanation service of ΩMEGA ("*Prover: Explain.*") to show more detailed proof steps that justify the assumption step.

```
\proof[ …
   "We prove" "(1)" "." "We assume" "$x \in A \cap (B \cup C)$" "."
   "Prover: Explain." "It follows that" … ]
```

This query is translated by the mediation module to the semantic representation.

```
\proof{}[ …
   \assumption[ … ], \aiq[ \explain{} ], \fact[ … ], … ]
```

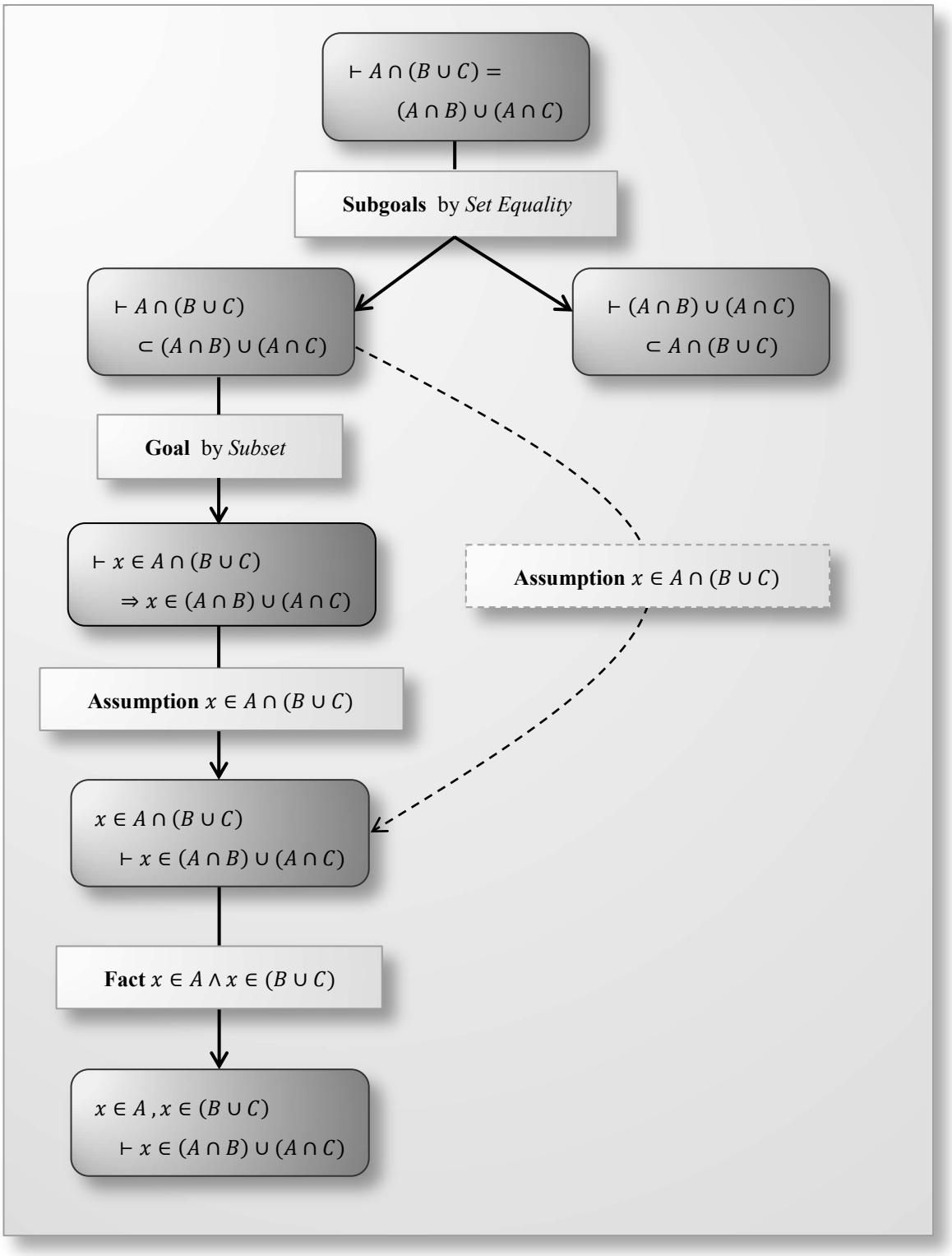


Figure 41. Proof Step Explanation by Lifting Justifications

We exploit the hierarchical proof data structure from our example in Figure 38 by replacing the selected proof step in the semantic representation with the lifted sequence of proof steps that justifies this step, as shown in Figure 41.

As a result ΩMEGA removes the query from the semantic representation and adds the lifted goal proof step as follows.

```
\proof{}[
 \subgoals[ … ,
  \proof{"(1)"}[
    \goal[ \F{"impl"}[\F{"in"}[\V{"x"},\F{"cap"}[\V{"A"},
                                        \F{"cup"}[\V{"B"},\V{"C"}]]],
             \F{"in"}[\V{"x"},\F{"cup"}[\F{"cap"}[\V{"A"},\V{"B"}],
                                         \F{"cap"}[\V{"A"},\V{"C"}]]]],
           \by[ \L{"Subset"} ] ],
    \assumption[ \F{"in"}[\V{"x"},\F{"cap"}[\V{"A"},
                                        \F{"cup"}[\V{"B"},\V{"C"}]]] ],
    \fact[ \F{"and"}[\F{"in"}[\V{"x"},V{"A"}],
                  \F{"in"}[\V{"x"},\F{"cup"}[\V{"B"},\V{"C"}]]]]]]]
```

The mediation module then inverse translates this modified semantic representation to the following modified interface document for the text-editor.

```
\proof[ …
  "We prove" "(1)" "." "We have to prove" "$(x \in A \cap
  (B \cup C)) \Rightarrow (x \in (A \cap B) \cup (A \cap C))$"
  "by Subset" "." "We assume" "$x \in A \cap (B \cup C)$" "." … ]
```

This expansion can of course be reversed again on request ("*Prover: Summarize.*"), resulting in an abstraction of the sequence of proof steps.

Similarly, the author may ask the proof assistance system to automatically complete a proof attempt ("*Prover: Complete.*"), to show the next proof step as a hint ("*Prover: Help.*"), or to apply an available assertion ("*Prover: Apply Set Equality.*"), that is, a definition, an axiom or a theorem. All requests are processed by generating one or more additional proof steps in the proof data structure. Finally, these proof steps are added to the semantic representation and propagated to the text-editor via the inverse translation pipeline as described in the previous example.

The presented authoring integrated query language has been designed for the interaction scenario between students and the proof assistance system. To satisfy the advanced requirements of expert authors, we could similarly integrate a specialized language like CRYSTAL [Dietrich & Schulz, 2010] which supports structured queries in a tactic language for the proof assistance system ΩMEGA.

9.5 Discussion

In this chapter, we introduced a *Controlled Mathematical Language* as an alternative to the semantic annotation of the mathematical document. Furthermore, we illustrated how proof obligations for ΩMEGA can be incrementally created from the proof steps written in exercise solutions. Finally, we demonstrated how the transformation pipeline can be used to propagate feedback for proof steps and to interact with the verification services of ΩMEGA within the text-editor.

Styles of Authoring. One of the most prominent systems for the publication of machine checked mathematics is MIZAR [Rudnicki & Trybulec, 1999] with one of the largest libraries of formalized mathematics. The language of the library is a well-designed compromise between a human-readable and a machine-processable language. Since the MIZAR system is not interactive, the usual workflow is to prepare an article, compile it and loop both steps until there is no error reported. In contrast to that, our *Change-Oriented Architecture* allows for both, a human-oriented and a machine-oriented representation, by providing techniques to automate the incremental translation in both directions.

ISABELLE/ISAR [Wenzel, 2007] is a generic framework for human-readable formal proof documents, both like and unlike MIZAR. The ISAR proof language provides general principles that may be instantiated to particular object-logics and applications. ISABELLE tries to check an ISAR proof, shows the proof status but does not patch the proof script for corrections. By integrating the services of the proof assistance system ΩMEGA, we are able to repair detected errors or under-specifications in proof steps.

A representative of distributed systems for the publication of machine checked mathematics is LOGIWEB [Grue, 2007]. It allows the authoring of articles in a customizable language but strictly separates the input from the output document, resulting in the usual LaTeX workflow. By using the WYSIWYG text-editors TeXmacs and MS WORD we combine input and output representation in a document-centric approach. In contrast to the standard batch-style workflow, we support an interactive style of authoring.

With respect to formalizing mathematical documents, there exist two extreme approaches. First, there is the MATHLANG [Kamareddine *et al*, 2004] approach of annotating every single piece of information in the document with their semantic annotation language in order to translate the result to various proof assistance systems [Lamar *et al*, 2009]. Second, there is the SAD system [Lyaletski *et al*, 2006] that tries to couple a controlled natural language as deep as possible with a proof assistance system [Verchinine *et al*, 2008]. In contrast to that, we presented a framework that supports both approaches in parallel and even in combination within the same document.

Management of Change. The way proof changes are handled in our approach differs to the best of our knowledge significantly from the way they are handled in other proof assistants, for example ISABELLE, COQ, or HOL. Two prominent user interfaces for such proof assistants are CTCOQ [Bertot, 1999] and PROOFGENERAL. In these systems the user develops a proof of a particular theorem of a given theory as a proof script by typing commands in an ASCII text-editor. Information about the open goals and other messages are sent to separate buffers of the text-editor. Proof scripts are composed of commands and they can be stepwise executed by moving the execution point of the proof script. The already executed parts of a proof script are thereby locked to avoid accidental editing. An undo step moves the execution point one step backwards. In our approach a proof step or an undo operation corresponds to the more general notion of a document change.

Locality of Proof Script Changes. In CTCOQ and PROOFGENERAL a change can only be performed at the current execution point, while in our approach the user can directly edit arbitrary parts of the proof script, in which case a corresponding change script is sent to the proof assistance system via the mediation module.

Parallel Editing of Proof Scripts. Our approach allows the user to edit the proofs of several theorems simultaneously, which is not possible if changes are restricted to only one location. The assertions that are available at a specific location are always those which are available in the theory or in an inherited theory. In our approach we therefore rely on the elaborate truth maintenance capabilities of the proof assistance system.

A sophisticated truth maintenance mechanism that comes close to ΩMEGA's approach is implemented in MATITA, which is an interactive theorem prover which organizes the mathematical knowledge in a searchable knowledge base. To ensure consistency of the library, MATITA employs two mechanisms called *invalidation* and *regeneration*. If a mathematical concept is changed, the concept itself and all concepts depending on it are invalidated and need to be regenerated to verify whether they are still valid. To regenerate an invalidated part of the library, MATITA re-executes the scripts that produced the invalidated concepts.

In our approach we do not invalidate the complete proof, but only those proof steps that depend on a changed part. For these parts we would also have to re-execute the scripts. However, we refrain to do so, because re-execution of scripts is time consuming. Therefore, our objective has been to improve the mechanism for propagation of changes and to develop mechanisms to repair proofs locally depending on the kinds of changes.

Integration. The invertible grammar formalism is a solid basis for further linguistic improvements of the *Controlled Mathematical Language*. For example, Oliver Bender designed grammar rules for the aggregation of proof steps [Bender, 2010] in our framework. Note however that the expressive power is always limited by the representational power of the proof assistance system. For example, how does the system deal with sequences like $a_1 + 1, \ldots, a_n + n$? (see [Kerber & Pollet, 2007] for more examples)

Besides the integration with the text-editor TEXMACS, Thorsten Hey developed a similar integration with MS WORD [Hey, 2009]. We have chosen WYSIWYG editors because the macro expansion of LATEX causes severe problems for the processing of semantic annotations. Indeed, in order to analyze all semantically important parts we would have to expand all macros, because a macro may hide an annotation or operators in a formula. Since the language elements of LATEX are macros themselves, we need to stop the macro expansion at the appropriate level, which is technically not supported by LATEX.

Furthermore, there is no possibility to implement the *observer pattern* in TEXMACS or MS WORD by a plugin to track document changes. Therefore, we use the versioning-based approach of mediation modules with *documents as interfaces* where the client communicates the *edit specification* of a document to obtain an update with an optimal change script at a reasonable granularity. Furthermore, we proposed to integrate a *feedback annotation language* and an *interaction language* with the controlled mathematical language. This approach turns the feedback and the interaction queries into first-class citizens of the document.

In our versioning-based approach, the task of conflict resolution is transferred to the proof assistance system. The text-editor simply rejects conflicting changes and commits its own state of the document. There are alternative approaches to this problem. Holger Gast proposed the IAPP protocol [Gast, 2008] that transfers the rights to change regions of a document between the text-editor and the proof assistance system. The PROOFGENERAL system [Aspinall, 2000] uses a locking approach where the regions of the document are locked which are currently being processed by the proof assistance system. The PGIP protocol of PROOFGENERAL is an interface protocol to connect theorem provers with user interfaces based on ASCII-text input and a single focus point-of-control. Many interactive theorem provers still use an interaction model that stems from command-line interpreters. In [Aspinall *et al*, 2008] we outline how the PGIP protocol can be extended towards supporting a *Change-Oriented Architecture*.

IV
Conclusion

10 Contributions

In this thesis, we presented a novel *Change-Oriented Architecture* and we outlined its application to *Mathematical Authoring Assistance*. The novelty of this architectural concept is the usage of *documents as interface* between components of the architecture. Thereby, the communication of document changes replaces standard method calls. A benefit of this architecture is the more independent component development thanks to mostly local change management.

We developed in this thesis an efficient algorithm for computing weighted semantic differences between documents. By using this algorithm we produce optimal change scripts for interface documents. The mediation between the interfaces of two components requires potentially the bidirectional transformation of the interface documents. For this purpose, we have developed a formalism for transformation grammars that is automatically invertible. Furthermore, we have developed a prototype system for mathematical authoring assistance by integrating the ΩMEGA system with the text-editors TEXMACS and MS WORD using a *Change-Oriented Architecture*. We will summarize the results of these three parts in the following.

Semantic Change Computation. We introduced a similarity specification that allows for defining when two documents are to be considered semantically similar or equal. Based on this notion we developed an algorithm for computing the optimal change script between two documents. Since the elements of an interface document represent objects in the interfaced component, we take into account the edit weights for changing or deleting these objects to compute the globally optimal change script. This is not necessarily the change script with the smallest change operations. Our contribution is the reduction of this constrained weighted tree alignment problem to a search problem. We developed a solution based on Dijkstra's shortest path algorithm and we have proved the termination and correctness of our algorithm. Its average time complexity is better in comparison to the state-of-the-art algorithm.

Invertible Transformation Grammar. We introduced a new formalism for transformation grammars with a focus on its automatic inversion because interface documents between two components have to be translated in both directions. A similarity specification can be defined for the source and target document. The notion of similarity is then taken into account in the pattern matching of the grammar interpreter. Unification constraints can be attached to the rules and have to be satisfied by the constructed parse tree.

Furthermore, in order to generate non-translated parts of the document in the inverse translation, we store the trace of the transformation. The presented formalism is a combination of the formalisms used by the natural language generation tool TG/2 and attribute grammars. Hence both parser and generator paradigms are supported. Additionally, the grammar writer declaratively defines the processing order for every rule. Thus, the knowledge about information flow is embedded into the grammar rules. We proved the correctness of the grammar inversion.

Mathematical Authoring Assistance. We illustrated the benefits of a *Change-Oriented Architecture* in a course scenario. Thereby, we conducted an exploratory study with a first-year mathematics course to analyze the edit behavior and selected linguistic aspects for mathematical authoring assistance. In order to support the self-extensible nature of mathematical documents we developed a new method which dynamically synthesizes grammar rules for the notation introduced for example in lecture notes. Furthermore, we presented incremental methods to verify the proof sketches in student exercise solutions by the ΩMEGA system. We modeled the transformation between the text-editor document and its formalization in multiple phases, each defined by an invertible transformation grammar.

Altogether we presented in this thesis a novel architecture for continuously evolving distributed systems whose components communicate their declarative status descriptions. In order to adapt this architecture to a particular system, one has to model the interface documents, define their similarity specifications and create transformation grammars for each pair of connected components. This defines the instances of *mediation modules* between the components of a system.

In this thesis, we described how the *Change-Oriented Architecture* solves the *mediation problem in-the-small* between a text-editor and the semantic services of the proof assistance system ΩMEGA. There exists also the *mediation problem in-the-large* between the different components of ΩMEGA. For this purpose, we outlined how the methods and concepts of the *Change-Oriented Architecture* support the evolutionary development of a knowledge-based system like ΩMEGA as presented in [Autexier *et al*, 2008]. The *Change-Oriented Architecture* can be used as a uniform component interface that reduces the amount of change management needed when the components evolve in parallel.

In combination with the prototypical integration with the text-editors TEX_{MACS} and MS WORD, we have developed a system that can be instantiated as a user interface for content-oriented applications. It allows the user to work in his concrete syntax, while the abstract syntax of the application is automatically being synchronized.

We conclude our presentation of a new *Change-Oriented Architecture for Mathematical Authoring Assistance* by answering some reflective questions.

Why is the propagation of mathematical knowledge by changes a good choice?

Let us consider the use case of interactive mathematical authoring with a scientific text-editor like TEXMACS. In this scenario, a mathematician develops a mathematical document with definitions, axioms, theorems and proofs in the text-editor. Efficient propagation of mathematical knowledge is essential in this case, since the document needs to be continuously synchronized with its formal counterpart in the proof assistance system. If we always replace the complete knowledge, we would rewrite the whole document in the text-editor. Consequently, we would lose large parts of the natural language text written by the author. On the other hand, we would lose the verification previously performed by the proof assistance system. For example, any already verified proof or any computation result from external systems would be lost. Clearly, propagating the changes between both sides preserves most of the work done on either side.

Can we do better than just computing and propagating syntactic changes?

By using semantic annotations or a *controlled mathematical language* for document authoring, the mediation module is able to extract the semantic tree-structured representation using the invertible grammar. Structured means in this context, that for example the proof steps are classified into assumptions, facts, subgoal introductions and other steps. Additionally, the syntactic formulas are translated into structural formulas making variables and operators explicit. If any syntactic change is reflected in a change of the semantic representation, this would trigger changes inside the proof assistance system, even if the change was only a semantically irrelevant document reorganization. We can do better by using our semantic change computation with an appropriate *similarity specification*.

How can we use the semantics to optimize the change computation?

A proof has in general to be re-verified when proof steps are permuted but not if the order of subgoals or the order of their subproofs changes. The latter is in fact only syntax sugaring and irrelevant to the formal verification. Although the document syntactically changes in that case, such a modification is filtered out by specifying the similarity for subgoal introductions in a way that the order of their content, that is the subgoals and subproofs, is not relevant to the change computation. Furthermore, for the formal verification all definitions, axioms and theorems are visible inside the whole theory. Hence, a rearrangement has no effect and does not need to be propagated from the text-editor to the proof assistance system.

Is the smallest possible change always the optimal change?

The aim is to compute changes that are optimal, where optimal means that processing them requires the least possible resources. Each element in the interface document of a component corresponds to an internal knowledge item. This item may contain lots of hidden knowledge items, or it may be the case that lots of other items depend on this item in its current state. For example, the weight of a theorem in a document can be defined by the amount of proof steps which apply this theorem. An algorithm for change computation then has to take these hidden weights into account in order to compute the optimal change script. Therefore, each service component sends the *edit specification* with these *edit weights* to the *mediation module* which computes an optimal change script for that service component.

Are there any other reasons to avoid arbitrary small changes?

The management of change is not equally sophisticated in all service components for several reasons. On the one hand, a component might only be interested in high-level changes because it is only responsible for the management in-the-large. On the other hand, a component, that wants to deal with arbitrary small changes, has to implement change methods for each type of element that can be changed. Hence, supporting arbitrary small changes implies implementing lots of change methods, which is not always a reasonable task. Therefore, each component sends the *edit specification* with its *edit limitation* to the *mediation module* which in turn computes a change script with an adequate granularity of changes for that service component.

What are the benefits of using documents as interfaces?

The concept of documents as interfaces allows for a rapid prototyping of knowledge-based distributed systems. The design process of a system using the *Change-Oriented Architecture* is usually as follows: First, one has to classify the different kinds of knowledge in the system and to separate their management into different components. Then, the dependencies between the kinds of knowledge imply the connections between the service components. The two-way translation between the interface documents of connected components is modeled by invertible grammars. Finally, each component can be developed independently and at different speed, since other connected components need only to know the interface document. They do not depend on any particular method or any particular level of change support of this component. Nevertheless, we have to note that the developed level of management of change of the core components, that are the components in the heart of the information flow, governs the performance and effectiveness of the entire system.

11 Future Work

"To undertake the formalization of just 100,000 pages of core mathematics would be one of the most ambitious collaborative projects ever undertaken in pure mathematics, the sequencing of a mathematical genome" – *Thomas Hales*

The mathematician Thomas Hales is well-known for his computer-aided proof of the Kepler conjecture, a centuries-old problem in discrete geometry which states that the most space-efficient way to pack spheres is in a pyramid shape. The proof relies extensively on methods from the theory of global optimization, linear programming, and interval arithmetic. Hales' proof proved difficult to verify. In 2003, it was reported that the *Annals of Mathematics* publication had an unusual editorial note stating that parts of the paper have not been possible to check, despite the fact that a team of 12 reviewers worked on verifying the proof for more than four years and that the reviewers were 99% certain that it is correct. However, the actual publication contains no such note. In response to the difficulties in verifying his proof, Hales launched the FLYSPECK project, an attempt to use computers to automatically verify every step of the proof. Unfortunately, Hales expects the project is likely to take 20 man-years of work.

This example taken from the cutting-edge of mathematics indicates that we are living in an era where proofs are becoming increasingly complex and computers are becoming necessary to perform verification. Additionally, it shows two problems that are becoming more and more important to solve. First, even after formalizing the proof with many years of work, the formalization itself needs to be verified by mathematical reviewers. Consider for example the inequality in Figure 42, which is one of the smaller inequalities by the way. We believe the reviewers will have a hard time to get familiar with the complex notation. This is only one example where our work could contribute in increasing the human readability of formalized mathematics.

```
(* SPIII-1998 Lemma 4.2 *)

let J_322621318 = (`!v0 v1 v2 v3 v4.
((is_quad_cluster_v v0 v1 v2 v3 v4) ==>
        ((sigma_quad_approx1 v0 v1 v2 v3 v4 <
        -- (&9494)/(&1000)
                + ((&30508)/(&100000))*
                ((dih_or_v v0 v1 v2 v4)+
                (dih_or_v v0 v2 v3 v1)))))`);;
```

Figure 42. Example inequality from the Flyspeck project (reproduced from [Hales, 2010])

Second, long-term projects often change team members over time or get contributions from third party collaborators. The new contributors need to get familiar with the existing results of the project. The pure verbalization of the formalized proof might be a good starting point, but with a bidirectional transformational approach they are immediately able to contribute.

Apart from providing a comfortable environment for editing formalized mathematics, it is important to integrate the theorem prover with this environment to obtain a productive system. Like in software engineering where the code is written in a specific programming language, mathematical theorems and proofs are formalized with respect to the logical foundations of a specific theorem prover. Hence, in order to turn our prototype system into an *integrated development environment* (IDE) for mathematics, which is the vision of our work, we still have a long way to go. Let me list the major problems that we encountered during our work, together with suggestions on how they could possibly be solved.

Multiple Prover Integration. Many of the sophisticated mathematical assistance services provided by our prototype have only been possible to be realized due to the special services provided by the ΩMEGA system. Definitely, Thomas Hales would not be happy to adapt the whole formalization of his proof to meet the requirements of a different theorem prover.

Fortunately, there is an open mathematical document format OMDOC available which proposes a standard to represent, provision and manage all kinds of mathematical knowledge. The problem with using OMDOC as an interchange format for theorem provers is however the discrepancy of their logical foundations. Additionally, since most theorem provers hide their creative knowledge in tactics and strategies, the devil lies clearly in the detail. There have been first steps ([Horozal & Rabe, 2009], [Dumbrava *et al*, 2009]) towards an approach for solving this blocking problem, but these case studies are still far away from an effective solution. Hence, we have to consider a different approach in the meanwhile.

There has been a second approach focusing on a generic user interface for theorem provers, the PROOFGENERAL project. Instead of trying to standardize the document format, this approach analyzed the interfaces and processing states of interactive theorem provers. They came up with a generic framework that has been adopted by all major theorem provers. We acknowledge this success in the community and therefore started to work on a plan to integrate our prototype system with PROOFGENERAL in order to integrate with all major theorem provers. Technical details of this collaboration have already been published in [Aspinall *et al*, 2008].

Libraries of Formalized Mathematics. In software engineering we often judge the productivity of a programming language by its battery of libraries. Nothing is more frustrating than losing time by reinventing the wheel. The same holds in fact for theorem provers and their libraries of formalized mathematics. Freek Wiedijk is continuously monitoring the progress made on an arbitrarily selected list of 100 well-known fundamental theorems [Wiedijk, 2010]. The statistics as of January 2010 are shown in Table 51. We list only the top five theorem provers.

Theorem Prover	Top 100 Theorems
HOL Light	74
Mizar	50
Isabelle	45
Coq	44
ProofPower	42

Table 51. Statistics of proved Top 100 theorems (reproduced from [Wiedijk, 2010])

These results might be the reason why Thomas Hales has chosen HOL Light to formalize his proof of the Kepler conjecture. A major part of the estimated 20 person years of work is definitely devoted to formalize missing libraries. Libraries are in fact more important than a particular system.

Theorem provers will become really productive systems as soon as their libraries reflect the mathematics in the undergraduate curriculum of a mathematics study. Freek Wiedijk estimates the time to formalize such a standard library for mathematics to take approximately 140 man-years of work [Wiedijk, 2000]. One important factor in this estimation is the relation between the size of an informal text of mathematics and its formalized counterpart. For the top five theorem provers this so called *de Bruijn factor* [Wiedijk, 1998] turned out to be close to 4. We argue that this factor can be significantly reduced by using a controlled mathematical language and an invertible transformation to the formal language of the theorem prover.

It is quite dazzling to see how much effort the mathematical knowledge management community has already invested to digitalize libraries of mathematics (for example [Bouche, 2008], [Ruddy, 2009], [Libbrecht *et al*, 2009]) in the last few years. New interface paradigms have been explored with electronic pen based input interfaces (for example [Suzuki *et al*, 2009], [Smirnova & Watt, 2008]). Unfortunately, all the expertise in representing, provisioning and managing mathematical knowledge did not solve the fundamental lack of formalized libraries. In fact, there is a trend towards setting up a framework and waiting for the Wikipedia effect.

Controlled Mathematical Language. In our opinion, the development of a controlled natural language for mathematics has to be a joint initiative with the target user group, the mathematicians. Therefore, we are collaborating with the NAPROCHE project of the Mathematical Institute in Bonn. They will use agile methods to continuously improve the *controlled mathematical language* (CML). They start with a combination of the base language, which we obtained from our exploratory study at Saarland University, and the semantic annotation language of the PLATΩ system, which can be used to annotate fragments that are beyond the coverage of the CML.

The members of the NAPROCHE project will then use the CML and the annotations to formalize their lecture notes of an introduction course to formal logic which targets Gödel's completeness theorems [Koepke, 2007]. Additionally, the students of this course will be asked to formalize their exercise solutions with the system. The continuous feedback will help us to naturally extend or adapt the CML in order to reduce the amount of annotations needed. Our hope is of course that this process will converge.

For the aspect of natural language analysis our goal is the analysis of sentences and parts of them that occur in definitions, theorems and proofs. Not all linguistic elements are going to be covered by the CML because on the one hand the grammar of the language would explode, and on the other hand these elements might not have a formal counterpart, for example adverbs like "accordingly" or "finally".

For the aspect of natural language generation we want to investigate the generation of context-aware CML fragments. For example, the formula $a \in M \to \cdots$ can be shown as "*if a is in M, then ...*" in a strongly verbalized context, and as "*if* $a \in M$*, then ...*" in a weakly verbalized context. Furthermore, we will integrate a discourse memory [Mossel, 2005] in order to allow for referring expressions in CML.

In order to make the CML more robust, we plan to integrate the findings of the MATHLANG project about a *narrative structure* [Kamareddine *et al*, 2007a] of mathematical texts. Thereby, Krzysztof Retel analyzed the rhetorical structure of articles in mathematical textbooks with respect to argumentative consistency and a gradual formalization into the MIZAR system. Consistency checks on intermediate transformation results may improve the general robustness of the CML.

Finally, we will have a close look at the results of the MOLTO project whose objective is the development of a multilingual on-line translation service with a stronger focus on translation correctness than on coverage. We will investigate the integration of the *grammatical framework* GF and its linguistic resources to transfer our resulting CML into multiple languages.

Besides the strong application relevance for the mathematical domain we believe that the presented *Change-Oriented Architecture* will offer significant performance and development improvements to other domains. We discuss two interesting examples in the following.

Visual Computing. In the field of visual computing, different technologies like multimedia or physics simulation merge with 3D computer graphics to create more realistic virtual worlds. For describing such virtual worlds in great detail, the *scene graph* has proved to be a flexible and powerful data structure to represent hierarchical data and its semantic interrelationships. A monolithic rendering approach would prevent exchange and recombination of different renderers. Therefore, a *service-oriented rendering architecture* SORA is proposed in [Repplinger *et al*, 2010]. The most interesting aspect of this approach is that it uses the interpretation-independent standard document format X3D [Brutzman & Daly, 2007] to represent the scene graph semantically. We propose to investigate the benefits of computing semantic changes between scene graphs to increase the rendering performance.

Formal Methods. The design and development of *safety-critical systems* require the use of formal methods in specification and verification [Frese *et al*, 2008]. The long-term research goal in this area is a verification system that maintains the dependencies between *requirements*, *code* and *documentation*, in order to actively support the change management process by notifying about affected parts and possibly proposing corrections. Unfortunately, the types of documents involved range from completely informal documents, consisting of paragraphs with natural language, to formal specifications which are highly structured and which have a precisely defined semantics. We propose to investigate the benefits of a mixed initiative approach with annotations and controlled languages to formalize the content of informal documents such that formal methods for management of change, for example development graphs [Autexier & Hutter, 2005], can be applied.

In general. It seems promising to investigate the benefits of semantic change computation for the performance optimization of distributed systems that use a structured semantic representation at the interfaces of their system components. Furthermore, the application areas which can profit from the formalism for invertible transformations are areas where informal content needs to be formalized, formal content needs to be verbalized, or different levels of formality need to be synchronized.

References

[Abiteboul *et al*, 2002] Abiteboul, Serge, Cluet, Sophie and Milo, Tova. (2002). Correspondence and translation for heterogeneous data. *Theoretical Computer Science*, vol 275, no. 1-2, pp. 179-213.

[Alexoudi *et al*, 2004] Alexoudi, Marianthi, Zinn, Claus and Bundy, Alan. (2004). English summaries of mathematical proofs. In Benzmüller, Christoph and Windsteiger, Wolfgang (eds.), *Proceedings of the Workshop on Computer-Supported Mathematical Theory Development*, Cork, Ireland, RISC Report Series, no. 04-14, RISC Institute, University of Linz, pp. 49-60.

[Amerkad *et al*, 2001] Amerkad, Ahmed, Bertot, Yves, Pottier, Loic and Rideau, Laurence. (2001). Mathematics and proof presentation in PCOQ. In Goré, Rajeev, Leitsch, Alexander and Nipkow, Tobias, (eds.), *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR), Workshop on Proof Transformation, Proof Presentations and Complexity of Proofs*, Siena, Italy, Springer.

[Asperti *et al*, 2006] Asperti, Andrea, Sacerdoti Coen, Claudio, Tassi, Enrico and Zacchiroli, Stefano. (2006). User Interaction with the MATITA Proof Assistant. *Journal of Automated Reasoning*, vol 39, no. 2, pp. 109-139.

[Aspinall *et al*, 2005] Aspinall, David, Lüth, Christoph and Winterstein, Daniel. (2005). Parsing, editing, proving: The PGIP display protocol. In Aspinall, David and Lüth, Christoph, (eds.), *Proceedings of 6th Workshop on User Interfaces for Theorem Provers (UITP)*, Edinburgh, Scotland.

[Aspinall *et al*, 2008] Aspinall, David, Autexier, Serge, Lüth, Christoph and Wagner, Marc. (2008). Towards Merging PLATΩ and PGIP. In Autexier, Serge and Benzmüller, Christoph, (eds.), *Proceedings of the 8th Workshop on User Interfaces for Theorem Provers (UITP)*, Montréal, Québec, Canada, pp. 3-21.

[Aspinall, 2000] Aspinall, David. (2000). PROOF GENERAL: A generic tool for proof development. In Graf, Susanne and Schwartzbach, Michael, (eds.), *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Berlin, Germany, Springer, pp. 38-42.

[Autexier & Dietrich, 2006] Autexier, Serge and Dietrich, Dominik. (2006). Synthesizing proof planning methods and ΩANTS agents from mathematical knowledge. In Borwein, Jonathan M. and Farmer, William M., (eds.), *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM)*, Wokingham, UK, Springer, pp. 94-109.

[Autexier & Hutter, 2005] Autexier, Serge and Hutter, Dieter. (2005). Formal software development in MAYA. In Hutter, Dieter and Stephan, Werner (eds.), *Festschrift in Honor of Jörg H. Siekmann*, Springer, pp. 407-432.

[Autexier *et al*, 2004] Autexier, Serge, Benzmüller, Christoph, Fiedler, Armin, Horacek, Helmut and Vo, Bao Q. (2004). Assertion-level proof representation with under-specification. *Electronic Notes in Theoretical Computer Science*, vol 93, pp. 5-23.

[Autexier *et al*, 2005] Autexier, Serge, Benzmüller, Christoph, Dietrich, Dominik, Meier, Andreas and Wirth, Claus-Peter. (2005). A generic modular data structure for proof attempts alternating on ideas and granularity. In Kohlhase, Michael, (ed.), *Proceedings of the 4th International Conference on Mathematical Knowledge Management (MKM)*, Bremen, Germany, Springer, pp. 126-142.

[Autexier *et al*, 2007] Autexier, Serge, Fiedler, Armin, Neumann, Thomas and Wagner, Marc. (2007). Supporting User-Defined Notations when Integrating Scientific Text-Editors with Proof Assistance Systems. In Kauers, Manuel, Kerber, Manfred, Miner, Robert and Windsteiger, Wolfgang, (eds.), *Proceedings of the 6th International Conference on Mathematical Knowledge Management (MKM)*, Hagenberg, Austria, Springer, pp. 176-190.

[Autexier *et al*, 2008] Autexier, Serge, Benzmüller, Christoph, Dietrich, Dominik and Wagner, Marc. (2008). Organisation, Transformation, and Propagation of Mathematical Knowledge in ΩMEGA. *Journal Mathematics in Computer Science*, vol 2, no. 2, pp. 253-277, Birkhäuser Basel.

[Autexier *et al*, 2009] Autexier, Serge, Benzmüller, Christoph, Dietrich, Dominik and Siekmann, Jörg. (2009). ΩMEGA: Resource Adaptive Processes in Automated Reasoning Systems. In Crocker, Matthew and Siekmann, Jörg H., (eds.), *Resource Adaptive Cognitive Processes, Part III – Resource-Adaptive Rationality in Machines*, Springer, pp. 389-423.

[Autexier, 2005] Autexier, Serge. (2005). The CoRe calculus. In Nieuwenhuis, Rober, (ed.), *Proceedings of the 20th International Conference on Automated Deduction (CADE)*, Tallin, Estonia, Springer, pp. 84-98.

[Bender, 2010] Bender, Oliver. (2010). Natürlich sprachliche Repräsentation von Mathematischen Diskursen, Diploma Thesis, Computerlinguistik, Universität des Saarlandes, Saarbrücken.

[Benzmüller & Sorge, 1998] Benzmüller, Christoph and Sorge, Volker. (1998). A blackboard architecture for guiding interactive proofs. In Giunchiglia, Fausto, (ed.), *Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, Sozopol, Bulgaria, Springer, pp. 102-114.

[Benzmüller & Sorge, 2000] Benzmüller, Christoph and Sorge, Volker. (2000). ΩANTS - An open approach at combining Interactive and Automated Theorem Proving. In Kerber, Manfred and Kohlhase, Michael, (eds.), *Proceedings of the 8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (CALCULEMUS)*, A K Peters, pp. 81-97.

[Benzmüller *et al*, 2006] Benzmüller, Christoph, Fiedler, Armin, Meier, Andreas, Pollet, Martin and Siekmann, Jörg. (2006). ΩMEGA. In Wiedijk, Freek, (ed.), *The seventeen provers of the world*, Springer, pp. 127-141.

[Benzmüller *et al*, 2007] Benzmüller, Christoph, Dietrich, Dominik, Schiller, Marvin and Autexier, Serge. (2007). Deep Inference for Automated Proof Tutoring. In Hertzberg, Joachim, Beetz, Michael and Englert, Roman, (eds.), *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI)*, Osnabrück, Germany, Springer, pp. 435-439.

[Bernth, 1997] Bernth, Arendse. (1997). EasyEnglish: A Tool for Improving Document Quality. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP)*, Washington, USA, pp. 159-165.

[Bertot, 1999] Bertot, Yves. (1999). The CTCOQ system: Design and architecture. *Formal Aspects of Computing*, vol 11, no. 3, pp. 225-243.

[Bille, 2005] Bille, Philip. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, vol 337, no. 1-3, pp. 217-239.

[Bouche, 2008] Bouche, Thierry. (2008). CEDRICS: When CEDRAM meets Tralics. In Sojka, Petr, (ed.), *Proceedings of the 1st Workshop Towards a Digital Mathematics Library (DML)*, Birmingham, UK, Masaryk University, pp. 153-165.

[Brutzman & Daly, 2007] Brutzman, Don and Daly, Leonard. (2007). *X3D: Extensible 3D Graphics for Web Authors*, Morgan Kaufmann.

[Buchberger *et al*, 1997] Buchberger, Bruno, Jebelean, Tudor, Kriftner, Franz, Marin, Mircea, Tomuta, Elena and Vasaru, Daniela. (1997). A Survey of the Theorema Project. In Küchlein, Wolfgang, (ed.), *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*, Maui, Hawaii, USA, ACM Press, pp. 384-391.

[Busemann & Horacek, 1998] Busemann, Stephan and Horacek, Helmut. (1998). A flexible shallow approach to text generation. In Hovy, Eduard, (ed.), *Proceedings of the 9th International Natural Language Generation Workshop (INLG)*, Niagara-on-the-Lake, Ontario, Canada, pp. 238-247.

[Busemann, 1996] Busemann, Stephan. (1996). Best-first surface realization. In Scott, Donia, (ed.), *Proceedings of the 8th International Natural Language Generation Workshop (INLG)*, Herstmonceux, Sussex, UK, pp. 101-110.

[Busemann, 2001] Busemann, Stephan. (2001). Language generation for cross-lingual document summarization. In Sheng, Huanye, (ed.), *Proceedings of the International Workshop on Innovative Language Technology and Chinese Information Processing (ILT&CIP)*, Beijing, China, Science Press.

[Busemann, 2005] Busemann, Stephan. (2005). Ten years after: an update on TG/2 (and friends). In Wilcock, Graham, Jokinen, Kristiina, Mellish, Chris and Reiter, Ehud, (eds.), *Proceedings of the 10th European Natural Language Generation Workshop (ENLG)*, Aberdeen, Scotland, pp. 32-39.

[Chawathe *et al*, 1996] Chawathe, Sudardshan, Rajaraman, Anand, Garcia-Molina, Hector and Widom, Jennifer. (1996). Change detection in hierarchically structured information. In Jagadish, H. V. and Mumick, Inderpal Singh, (eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montréal, Québec, Canada, ACM Press, pp. 493-504.

[Cheikhrouhou & Sorge, 2000] Cheikhrouhou, Lassaad and Sorge, Volker. (2000). PDS - A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the 1st International Conference on Artificial and Computational Intelligence for Decision Control and Automation in Engineering and Industrial Applications (ACIDCA)*, Monastir, Tunisia.

[Chen, 1998] Chen, Weimin. (1998). More efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, vol 26, no. 2, pp. 370-385.

[Chen, 2001] Chen, Weimin. (2001). New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, vol 40, no. 2, pp. 135-158.

[Chomsky, 1956] Chomsky, Noam. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, vol 2, no. 3, pp. 113-124.

[Chomsky, 1957] Chomsky, Noam. (1957). *Syntactic Structures*. Mouton, The Hague.

[Choppella & Haynes, 2005] Choppella, Venkatesh and Haynes, Christopher. (2005). Source-tracking unification. *Information and Computation*, vol 201, no. 2, pp. 121-159.

[Clark *et al*, 2005] Clark, Peter, Harrison, Philip, Jenkins, Thomas, Thompson, John and Wojcik, Richard H.. (2005). Acquiring and Using World Knowledge Using a Restricted Subset of English. In Russell, Ingrid and Markov, Zdravko, (eds.), *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, Clearwater Beach, Florida, USA, AAAI Press, pp. 506-511.

[Claypool & Rundensteiner, 2004] Claypool, Kajal and Rundensteiner, Elke. (2004). AUP: Adaptive Change Propagation Across Data Model Boundaries. In Williams, Howard and MacKinnon, Lachlan, (eds.), *Proceedings of the 21st British National Conference on Databases (BNCOD)*, Edinburgh, Scotland, UK, Springer, pp. 72-83.

[Cobéna *et al*, 2002] Cobéna, Grégory, Abdessalem, Talel and Hinnach, Yassine. (2002). A comparative study for XML change detection. In Pucheral, Philippe, (ed.), *Proceedings of the 18th Journées Bases de Données Avancées (BDA)*, Evry, Actes.

[Coquand & Huet, 1988] Coquand, Thierry and Huet, Gerard. (1988). The calculus of constructions. *Information and Computation*, vol 76, no. 2-3, pp. 95-120.

[Davenport, 2000] Davenport, James. (2000). A Small OPENMATH Type System, *ACM SIGSAM Bulletin*, vol 34, no. 2, pp. 16-21.

[de Bruijn, 1970] de Bruijn, Nicolaas G. (1970). The mathematical language AUTOMATH, its usage and some of its extensions. In Laudet, Michel, Lacombe, Daniel, Nolin, Louis and Schützenberger, Marcel, (eds.), *Proceedings of the Symposium on Automatic Demonstration*, Versailles, France, Springer, pp. 29-61.

[de Bruijn, 1994] de Bruijn, Nicolaas G. (1994). Mathematical Vernacular: a Language for Mathematics with Typed Sets. In Nederpelt, Rop P., Geuvers, J. Herman and de Vrijer, Roel C., (eds.), *Selected Papers on Automath*, North-Holland Publishing Company, pp. 865-936.

[de Carvalho & Jürgensen, 2008] de Carvalho, Jackson Marques and Jürgensen, Helmut. (2008). A Dynamical Document Structure to Capture the Semantics of Mathematical Concepts. In Dini, Petre and Dascalu, Sergiu, (eds.), *Proceedings of the 1st International Conference on Advances in Computer-Human Interaction (ACHI)*, Sainte Luce, Martinique, IEEE Computer Society, pp. 257-264.

[Dietrich & Buckley, 2007] Dietrich, Dominik and Buckley, Mark. (2007). Verification of Proof Steps for Tutoring Mathematical Proofs. In Luckin, Rosemary, Koedinger, Kenneth R. and Greer, Jim E., (eds.), *Proceedings of the 13th International Conference on Artificial Intelligence in Education (AIED)*, Los Angeles, California, USA, IOS Press, pp. 560-562.

[Dietrich & Schulz, 2010] Dietrich, Dominik and Schulz, Ewaryst. (2010). Crystal: Integrating Structured Queries into a Tactic Language. *Journal of Automated Reasoning*, vol 44, no. 1-2, pp. 79-110.

[Dietrich *et al*, 2008] Dietrich, Dominik, Schulz, Ewaryst and Wagner, Marc. (2008). Authoring Verified Documents by Interactive Proof Construction and Verification in Text-Editors. In Autexier, Serge, Campbell, John, Rubio, Julio, Sorge, Volker, Suzuki, Masakazu and Wiedijk, Freek, (eds.), *Proceedings of the 7th International Conference on Mathematical Knowledge Management (MKM)*, Birmingham, UK, Springer, pp. 398-414.

[Dietrich, 2006] Dietrich, Dominik. (2006). The TASKLAYER of the ΩMEGA system, Diploma thesis, Informatik, Universität des Saarlandes, Saarbrücken.

[Dijkstra, 1959] Dijkstra, Edsger W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, vol 1, pp. 269-271.

[Dumbrava *et al*, 2009] Dumbrava, Stefania, Horozal, Fulya and Sojakova, Kristina. (2009). A Case Study on Formalizing Algebra in a Module System. In Rabe, Florian and Schürmann, Carsten, (eds.), *Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants (MLPA)*, Montreal, Canada, ACM Press, pp. 11-18.

[Fiedler, 2001] Fiedler, Armin. (2001). User-adaptive Proof Explanation, PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, Saarbrücken, Germany.

[Franke & Kohlhase, 1999] Franke, Andreas and Kohlhase, Michael. (1999). System Description: MATHWEB a System for Distributed Automated Theorem Proving. In Ganzinger, Harald, (ed.), *Proceedings of the 16th International Conference on Automated Deduction (CADE)*, Trento, Italy, Springer, pp. 217-221.

[Frese *et al*, 2008] Frese, Udo, Hausmann, Daniel, Lüth, Christoph, Täubig, Holger and Walter, Dennis. (2008). Zertifizierung einer Sicherungskomponente mittels durchgängig formaler Modellierung. In Maalej, Walid and Brügge, Bernd, (eds.), *Proceedings of Software Engineering (Workshops)*, Munich, Germany, Lecture Notes in Informatics, GI, pp. 335-338.

[Fuchs *et al*, 2008] Fuchs, Norbert, Kaljurand, Kaarel and Kuhn, Tobias. (2008). Attempto Controlled English for Knowledge Representation. In Baroglio, Cristina, Bonatti, Piero A., Maluszynski, Jan, Marchiori, Massimo, Polleres, Axel and Schaffert, Sebastian, (eds.), *Proceedings of the 4th International Summer School on Reasoning Web*, Venice, Italy, Springer, pp. 104-124.

[Gabbay *et al*, 2010] Gabbay, Dov, Siekmann, Jörg and Wirth, Claus-Peter. (2010). *Hilbert Bernays Project*, viewed 22 January 2010, <http://www.ags.uni-sb.de/~cp/p/hilbertbernays>.

[Gamma *et al*, 1994] Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[Ganzinger & Giegerich, 1984] Ganzinger, Harald and Giegerich, Robert. (1984). Attribute coupled grammars. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, Montréal, Québec, Canada, ACM Press, pp. 157-170.

[Garrett, 2005] Garrett, Jesse J. (2005). *AJAX: A new approach to web applications*, viewed 22 January 2010, <http://www.adaptivepath.com/ideas/essays/archives/000385.php>.

[Gast, 2008] Gast, Holger. (2008). Managing proof documents for asynchronous processing. In Autexier, Serge and Benzmüller, Christoph, (eds.), *Proceedings of the 8th Workshop on User Interfaces for Theorem Provers (UITP)*, Montréal, Québec, Canada, pp. 37-51.

[Gazdar *et al*, 1985] Gazdar, Gerald, Klein, Ewan H., Pullum, Geoffrey K. and Sag, Ivan A. (1985). *Generalized Phrase Structure Grammar*, Harvard University Press, Oxford.

[Gentzen, 1934] Gentzen, Gerhard. (1934). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, vol 39, pp. 176-210, 405-431.

[Gentzen, 1969] Gentzen, Gerhard. (1969). *The Collected Papers of Gerhard Gentzen (1934-1938)*, English translation by M. E. Szabo, North-Holland Publishing Company.

[Geuvers *et al*, 2000] Geuvers, Herman, Wiedijk, Freek, Zwanenburg, Jan, Pollack, Randy and Barendregt, Henk. (2000). *Theory development leading to the Fundamental Theorem of Algebra*, viewed 22 January 2010, <http://www.cs.ru.nl/~freek/fta/>.

[Greenwald *et al*, 2003] Greenwald, Michael B., Moore, Jonathan T., Pierce, Benjamin C. and Schmitt, Alan. (2003). A Language for Bi-Directional Tree Transformations, Technical Report MS-CIS-03-08, University of Pennsylvania.

[Grue, 2007] Grue, Klaus. (2007). The Layers of Logiweb. In Kauers, Manuel, Kerber, Manfred, Miner, Robert and Windsteiger, Wolfgang, (eds.), *Proceedings of the 6th International Conference on Mathematical Knowledge Management (MKM)*, Hagenberg, Austria, Springer, pp. 250-264.

[Hales, 2010] Hales, Thomas. (2010). *The Flyspeck Project*, viewed 22 January 2010, <http://code.google.com/p/flyspeck/>.

[Hedin, 1994] Hedin, Görel. (1994). An Overview of Door Attribute Grammars. In Fritzson, Peter, (ed.), *Proceedings of the 5th International Conference on Compiler Construction (CC)*, Edinburgh, UK, Springer, pp. 31-51.

[Hey, 2009] Hey, Thorsten. (2009). iMath - Ein Word Plugin für Mathematische Assistenz-systeme, Bachelorarbeit, Universität des Saarlandes, Saarbrücken.

[Hilbert & Bernays, 1934] Hilbert, David and Bernays, Paul. (1934). *Grundlagen der Mathematik, Vol.1*, Berlin, Springer.

[Hilbert & Bernays, 1939] Hilbert, David and Bernays, Paul. (1939). *Grundlagen der Mathematik, Vol.2*, Berlin, Springer.

[Holland-Minkley *et al*, 1999] Holland-Minkley, Amanda M., Barzilay, Regina and Constable, Robert L. (1999). Verbalization of high-level formal proofs. In *Proceedings of the 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence (AAAI)*, Orlando, Florida, USA, MIT Press, pp. 277-284.

[Horozal & Rabe, 2009] Horozal, Fulya and Rabe, Florian. (2009). Representing Model Theory in a Type-Theoretical Logic Framework. In Ayala-Rincón, Mauricio and Kamareddine, Fairouz, (eds.), *Proceedings of the 4th Workshop on Logical and Semantic Frameworks, with Applications (LSFA)*, Brasilia, Brazil, pp. 49-65.

[Huang, 1994] Huang, Xiarong. (1994). Human Oriented Proof Presentation: A Reconstructive Approach, PhD thesis, Saarland University.

[Hutter, 2000] Hutter, Dieter. (2000). Management of change in verification systems. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, IEEE Computer Society, pp. 23-34.

[ISO/IEC, 2006] ISO/IEC. (2006). Information Technology - Open Document Format for Office Applications (OpenDocument) v1.0, International Standards, ISO/IEC 26300:2006(E).

[ISO/IEC, 2008] ISO/IEC. (2008). Information Technology - Office Open XML Formats, International Standards, ISO/IEC 29500:2008.

[Jansson & Lingas, 2001] Jansson, Jesper and Lingas, Andrzej. (2001). A fast algorithm for optimal alignment between similar ordered trees. In Amir, Amihood and Landau, Gad M., (eds.), *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Jerusalem, Isreal, Springer, pp. 232-240.

[Jaskowski, 1934] Jaskowski, Stanislaw. (1934). On the rules of suppositions in formal logic, *Studia Logica*, vol 1, pp. 5-32 (reprinted in: McCall, Storrs, (ed.), *Polish logic 1920-1939*, pp. 232-258, Oxford University Press, 1967).

[Jiang *et al*, 1994] Jiang, Tao, Wang, Lusheng and Zhang, Kaizhong. (1994). Alignment of trees - an alternative to tree edit. In Crochemore, Maxime and Gusfield, Dan, (eds.), *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Asilomar, California, USA, Springer, pp. 75-86.

[Kamareddine & Nederpelt, 2004] Kamareddine, Fairouz and Nederpelt, Rob. (2004). A Refinement of de Bruijn's Formal Language of Mathematics. *Journal of Logic, Language and Information*, vol 13, no. 3, pp. 287-340.

[Kamareddine *et al*, 2004] Kamareddine, Fairouz, Maarek, Manuel and Wells, Joe B. (2004). MATHLANG: Experience-driven Development of a New Mathematical Language. In Asperti, Andrea, Bancerek, Grzegorz and Trybulec, Andrzej, (eds.), *Proceedings of the 3rd International Conference on Mathematical Knowledge Management (MKM)*, Bialowieza, Poland, Springer, pp. 160-174.

[Kamareddine *et al*, 2007a] Kamareddine, Fairouz, Maarek, Manuel, Retel, Krzysztof and Wells, Joe B. (2007). Narrative Structure of Mathematical Texts. In Kauers, Manuel, Kerber, Manfred, Miner, Robert and Windsteiger, Wolfgang, (eds.), *Proceedings of the 6th International Conference on Mathematical Knowledge Management (MKM)*, Hagenberg, Austria, Springer, pp. 296-312.

[Kamareddine *et al*, 2007b] Kamareddine, Fairouz, Lamar, Robert, Maarek, Manuel and Wells, Joe B. (2007). Restoring Natural Language as a Computerised Mathematics Input Method. In Kauers, Manuel, Kerber, Manfred, Miner, Robert and Windsteiger, Wolfgang, (eds.), *Proceedings of the 6th International Conference on Mathematical Knowledge Management (MKM)*, Hagenberg, Austria, Springer, pp. 280-295.

[Kamp & Reyle, 1993] Kamp, Hans and Reyle, Uwe. (1993). *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*, Kluwer Academic Publisher.

[Kamprath *et al*, 1998] Kamprath, Christine, Adolphson, Eric, Mitamura, Teruko and Nyberg, Eric. (1998). Controlled Language for Multilingual Document Production: Experience with Caterpillar Technical English. In *Proceedings of the 2nd International Workshop on Controlled Language Applications (CLAW)*, Pittsburgh, USA.

[Kaplan & Bresnan, 1981] Kaplan, Ronald and Bresnan, Joan. (1981). Lexical-functional Grammar: A Formal System for Grammatical Representation. In Bresnan, Joan, (ed.), *The Mental Representation of Grammatical Relations*, Cambridge, MIT Press.

[Kastens, 1980] Kastens, Uwe. (1980). Ordered attribute grammars, *Acta Informatica*, vol 13, no. 3, pp. 229-256.

[Kerber & Pollet, 2007] Kerber, Manfred and Pollet, Martin. (2007). Informal and Formal Representations in Mathematics. In Matuszewski, Roman and Zalewska, Anna, (eds.), *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, University of Bialystok, pp. 75-94.

[Kilpelainen & Mannila, 1995] Kilpelainen, Pekka and Mannila, Heikki. (1995). Ordered and unordered tree inclusion, *SIAM Journal of Computing*, vol 24, no. 2, pp. 340-356.

[Klein, 1998] Klein, Philip N. (1998). Computing the edit-distance between unrooted ordered trees. In Bilardi, Gianfranco, Italiano, Giuseppe F., Pietracaprina, Andrea and Pucci, Geppino, (eds.), *Proceedings of the 6th Annual European Symposium on Algorithms (ESA)*, Venice, Italy, Springer, pp. 91-102.

[Knöll & Mezini, 2009] Knöll, Roman and Mezini, Mira. (2009). Π: A Pattern Language. In Arora, Shail and Leavens, Gary T., (eds.), *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, USA, ACM Press, pp. 503-522.

[Knuth, 1968] Knuth, Donald E. (1968). Semantics of context-free languages, *Theory of Computing Systems*, vol 2, no. 2, pp. 127-145.

[Knuth, 1969] Knuth, Donald E. (1969). *The Art of Computer Programming*, Addison Wesley.

[Koepke & Schröder, 2003] Koepke, Peter and Schröder, Bernhard. (2003). Natürlich formal. In Willee, Gerd, Schröder, Bernhard and Schmitz, Hans-Christian, (eds.), *Computerlinguistik – Was geht, was kommt? Computational Linguistics - Achievements and Perspectives*, St. Augustin, Gardez!-Verlag.

[Koepke, 2007] Koepke, Peter. (2007). Gödel's completeness theorem with natural language formulas. In Müller, Thomas and Newen, Albert, (eds.), *Logik, Begriffe, Prinzipien des Handelns - Logic, Concepts, Principles of Action*, Paderborn, mentis-Verlag, pp. 49-63.

[Kohlhase & Kohlhase, 2006] Kohlhase, Andrea and Kohlhase, Michael. (2006). Communities of Practice in MKM: An Extensional Model. In Borwein, Jonathan M. and Farmer, William M., (eds.), *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM)*, Wokingham, UK, Springer, pp. 179-193.

[Kohlhase *et al*, 2009] Kohlhase, Michael, Lange, Christoph, Müller, Christine, Müller, Normen and Rabe, Florian. (2009). Notations for Active Mathematical Documents. Technical Report, KWARC, Jacobs University, Bremen, Germany.

[Kohlhase, 2000] Kohlhase, Michael. (2000). OMDOC: Towards an Internet Standard for the Distribution and Teaching of Mathematical Knowledge. In Campbell, John A. and Roanes-Lozano, Eugenio, (eds.), *Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation (AISC)*, Madrid, Spain, Springer, pp. 32-52.

[Kohlhase, 2006] Kohlhase, Michael. (2006). *OMDOC: Open Mathematical Documents*, version 1.2, Springer.

[Lamar *et al*, 2009] Lamar, Robert, Kamareddine, Fairouz and Wells, Joe B. (2009). MATHLANG Translation to Isabelle Syntax. In Carette, Jacques, Dixon, Lucas, Sacerdoti Coen, Claudio and Watt, Stephen M., (eds.), *Proceedings of the 8th International Conference on Mathematical Knowledge Management (MKM)*, Grand Bend, Canada, Springer, pp. 373-388.

[Lavie & Tomita, 1993] Lavie, Alon and Tomita, Masaru. (1993). GLR* - An efficient Noise-skipping Parsing Algorithm for Context Free Grammars. In *Proceedings of the 3rd International ACM SIGPARSE Workshop on Parsing Technologies (IWPT)*, Tilburg, The Netherlands, ACM Press, pp. 123-134.

[Libbrecht *et al*, 2009] Libbrecht, Paul, Kortenkamp, Ulrich and Mercat, Christian. (2009). Web-Library of Interactive Geometry. In Sojka, Petr, (ed.), *Proceedings of the 2nd Workshop Towards a Digital Mathematics Library (DML)*, Grand Bend, Canada, Masaryk University, pp. 95-106.

[Lin *et al*, 2001] Lin, Guo-Hui, Ma, Bin and Zhang, Kaizhong. (2001). Edit distance between two RNA structures. In Lengauer, Thomas, Sankoff, David, Istrail, Sorin and Peuvzner, Pavel, (eds.), *Proceedings of the 5th International Conference on Computational Biology (RECOMB)*, Montréal, Québec, Canada, ACM Press, pp. 211-220.

[Liu & Teitelbaum, 1995a] Liu, Yanhong A. and Teitelbaum, Tim. (1995). Caching intermediate results for program improvement. In Scherlis, William, (ed.), *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, La Jolla, California, USA, ACM Press, pp. 190-201.

[Liu & Teitelbaum, 1995b] Liu, Yanhong A. and Teitelbaum, Tim. (1995). Systematic derivation of incremental programs, *Science of Computer Programming*, vol 24, no. 1, pp. 1-39.

[Liu *et al*, 1996] Liu, Yanhong A., Stoller, Scott D. and Teitelbaum, Tim. (1996). Discovering auxiliary information for incremental computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, Florida, USA, ACM Press, pp. 157-170.

[Lyaletski *et al*, 2006] Lyaletski, Alexander, Paskevich, Andrey and Verchinine, Konstantin. (2006). SAD as a mathematical assistant - how should we go from here to there?, *Journal of Applied Logic*, vol 4, no. 4, pp. 560-591.

[Mamane & Geuvers, 2006] Mamane, Lionel E. and Geuvers, Herman. (2006). A document-oriented Coq plugin for TEXMACS. In *Proceedings of the 2nd Workshop on Mathematical User Interfaces (MATHUI)*, Wokingham, Great Britain.

[Martelli & Montanari, 1982] Martelli, Alberto and Montanari, Ugo. (1982). An Efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol 4, no. 2, pp. 258-282.

[Melis & Meier, 2000] Melis, Erica and Meier, Andreas. (2000). Proof Planning with Multiple Strategies. In Lloyd, John, Dahl, Veronica, Furbach, Ulrich, Kerber, Manfred, Lau, Kung-Kiu, Palamidessi, Catuscia, Pereira, Luís Moniz, Sagiv, Yehoshua and Stuckey, Peter J., (eds.), *Proceedings of the 1st International Conference on Computational Logic (CL)*, London, UK, Springer, pp. 644-659.

[Melis & Siekmann, 1999] Melis, Erica and Siekmann, Jörg. (1999). Knowledge-Based Proof Planning, *Journal of Artificial Intelligence*, vol 115, no. 1, pp. 65-105.

[Melis *et al*, 2008] Melis, Erica, Meier, Andreas and Siekmann, Jörg. (2008). Proof planning with multiple strategies, *Journal of Artificial Intelligence*, vol 172, no. 6-7, pp. 656-684.

[Melis et al, 2009] Melis, Erica, Goguadze, George, Libbrecht, Paul and Ullrich, Carsten. (2009). Culturally Aware Mathematics Education Technology. In Blanchard, Emmanuel and Allard, Danièle, (eds.), *The Handbook of Research in Culturally-Aware Information Technology: Perspectives and Models*, IGI-Global.

[Mellish, 1989] Mellish, Chris. (1989). Some chart-based techniques for parsing ill-formed input. In Hirschberg, Julia, (ed.), *Proceedings of the 27th Annual Meeting on the Association for Computational Linguistics (ACL)*, Vancouver, British Columbia, Canada, ACL, pp. 102-109.

[Mossakowski *et al*, 2006] Mossakowski, Till, Autexier, Serge and Hutter, Dieter. (2006). Development graphs - proof management for structured specifications, *Journal of Logic and Algebraic Programming*, vol 67, no. 1-2, pp. 114-145.

[Mossel, 2005] Mossel, Eelco. (2005). A reference memory with flexible rule-based functionality for anaphora resolution, Diploma thesis, University of Twente, Enschede, The Netherlands.

[Müller & Wagner, 2007] Müller, Normen and Wagner, Marc. (2007). Towards Improving Interactive Mathematical Authoring by Ontology-driven Management of Change. In Hinneburg, Alexander, (ed.), *Proceedings of Lernen – Wissen – Adaption (LWA), Workshop Wissens- und Erfahrungsmanagement (WM)*, Halle, Germany, Springer, pp. 289-295.

[Nakagawa & Buchberger, 2001] Nakagawa, Koji and Buchberger, Bruno. (2001). Presenting Proofs Using Logicographic Symbols. In *Proceedings of the 1st Workshop on Proof Transformation and Presentation and Proof Complexities (PTP)*, Siena, Italy.

[Nipkow *et al*, 2002] Nipkow, Tobias, Paulson, Lawrence C. and Wenzel, Markus. (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, Springer.

[op den Akker *et al*, 1990] op den Akker, Rieks, Melichar, Borivoj and Tarhio, Jorma. (1990). The hierarchy of LR-attributed grammars. In Deransart, Pierre and Jourdan, Martin, (eds.), *Proceedings of the International Conference on Attribute Grammars and their Application (WAGA)*, Paris, France, Springer, pp. 13-28.

[Padovani & Zacchiroli, 2006] Padovani, Luca and Zacchiroli, Stefano. (2006). From Notation to Semantics: There and Back Again. In Borwein, Jonathan M. and Farmer, William M., (eds.), *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM)*, Wokingham, UK, Springer, pp. 194-207.

[Pereira & Warren, 1980] Pereira, Fernando and Warren, David. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Journal of Artificial Intelligence*, vol 13, no. 3, pp. 231-278.

[Pereira & Warren, 1983] Pereira, Fernando and Warren, David. (1983). Parsing as deduction. In Calzolari, Nicoletta, (ed.), *Proceedings of the 21st Annual Meeting on the Association for Computational Linguistics (ACL)*, Morristown, USA, ACL, pp. 137-144.

[Pfenning, 1999] Pfenning, Frank. (1999). Logical Frameworks. In Robinson, John Alan and Voronkov, Andrei, (eds.), *Handbook of Automated Reasoning*, vol 2, Elsevier Science Publishers, pp. 1063-1147.

[Pollard & Sag, 1994] Pollard, Carl and Sag, Ivan A. (1994). *Head-Driven Phrase Structure Grammar*, Chicago, University of Chicago Press.

[Radzevich, 2006] Radzevich, Svetlana. (2006). Semantic-based diff, patch and merge for XML-documents, Master thesis, Saarland University, Saarbrücken.

[Ranta, 2004] Ranta, Aarne. (2004). Grammatical Framework: A Type-Theoretical Grammar Formalism, *Journal of Functional Programming*, vol 14, no. 2, pp. 145-189.

[Repplinger *et al*, 2010] Repplinger, Michael, Löffler, Alexander, Schug, Benjamin and Slusallek, Philipp. (2010). SORA: a Service-Oriented Rendering Architecture. In Latoschik, Marc Erich, Reiners, Dirk, Blach, Roland, Figueroa, Pablo and Dachselt, Raimund, (eds.), *Proceedings of the 3rd Workshop on Software Engineering and Architecture for Realtime Interactive Systems (SEARIS)*, IEEE Virtual Reality, Waltham, MA, USA, Shaker Verlag, pp. 25-32.

[Richter, 1996] Richter, Thorsten. (1996). A new measure of the distance between ordered trees and its applications, Technical Report, Department of Computer Science, University of Bonn.

[Richter, 1997] Richter, Thorsten. (1997). A new algorithm for the ordered tree inclusion problem. In Apostolico, Alberto and Hein, Jotun, (eds.), *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Aarhus, Denmark, Springer, pp. 150-166.

[Robinson, 1965] Robinson, John A. (1965). A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the ACM*, vol 12, no. 1, pp. 23-41.

[Rönnau *et al*, 2009] Rönnau, Sebastian, Philipp, Geraint and Borghoff, Uwe. (2009). Efficient Change Control of XML Documents. In Borghoff, Uwe M. and Chidlovskii, Boris, (eds.), *Proceedings of the 9th ACM Symposium on Document Engineering (DOCENG)*, Munich, Germany, ACM Press, pp. 3-12.

[Ruddy, 2009] Ruddy, David. (2009). The Evolving Digital Mathematics Network. In Sojka, Petr, (ed.), *Proceedings of the 2nd Workshop Towards a Digital Mathematics Library (DML)*, Grand Bend, Canada, Masaryk University, pp. 3-16.

[Rudnicki & Trybulec, 1999] Rudnicki, Piotr and Trybulec, Andrzej. (1999). On Equivalents of Well-foundedness, *Journal of Automated Reasoning*, vol 23, no. 3, pp. 197-234.

[Sacerdoti Coen & Zacchiroli, 2004] Sacerdoti Coen, Claudio and Zacchiroli, Stefano. (2004). Efficient ambiguous parsing of mathematical formulae. In Asperti, Andrea, Bancerek, Grzegorz and Trybulec, Andrzej, (eds.), *Proceedings of the 3rd International Conference on Mathematical Knowledge Management (MKM)*, Bialowieza, Poland, Springer, pp. 347-362.

[Sacerdoti Coen & Zacchiroli, 2008] Sacerdoti Coen, Claudio and Zacchiroli, Stefano. (2008). Spurious Disambiguation Errors and How To Get Rid of Them, *Journal of Mathematics in Computer Science*, vol 2, no. 2, pp. 355-378.

[Schiller *et al*, 2008] Schiller, Marvin, Dietrich, Dominik and Benzmüller, Christoph. (2008). Proof step analysis for proof tutoring – a learning approach to granularity. *Teaching Mathematics and Computer Science*, vol 6, no. 2, pp. 325-343.

[Schiller, 2010] Schiller, Marvin. (2010). Granularity Analysis for Tutoring Mathematical Proofs, PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, Saarbrücken, Germany.

[Selkow, 1977] Selkow, Stanley M. (1977). The tree-to-tree editing problem, *Information Processing Letters*, vol 6, no. 6, pp. 184-186.

[Shieber, 1988] Shieber, Stuart M. (1988). A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, Hungary, pp. 614-619.

[Shieber, 1993] Shieber, Stuart M. (1993). The Problem of Logical-Form Equivalence, *Journal of Computational Linguistics*, vol 19, pp. 179-190.

[Siekmann & Autexier, 2007] Siekmann, Jörg and Autexier, Serge. (2007). Computer supported formal work: Towards a digital mathematical assistant. In Matuszewski, Roman and Zalewska, Anna, (eds.), *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, University of Bialystok, pp. 231-248.

[Siekmann *et al*, 2002] Siekmann, Jörg, Benzmüller, Christoph, Fiedler, Armin, Meier, Andreas and Pollet, Martin. (2002). Proof Development with ΩMEGA: $\sqrt{2}$ is irrational. In Baaz, Matthias and Voronkov, Andrei, (eds.), *Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, Tbilisi, Georgia, Springer, pp. 367-387.

[Siekmann *et al*, 2006] Siekmann, Jörg, Benzmüller, Christoph and Autexier, Serge. (2006). Computer supported Mathematics with ΩMEGA, *Journal of Applied Logic, Special Issue on Mathematics Assistance Systems*, vol 4, no. 4, pp. 533-559.

[Siekmann, 1989] Siekmann, Jörg. (1989). Unification Theory, *Journal of Symbolic Computation*, vol 7, no. 3-4, pp. 207-274.

[Smirnova & Watt, 2006] Smirnova, Elena and Watt, Stephen M. (2006). Notation Selection in Mathematical Computing Environments. In Dumas, Jean-Guillaume, (ed.), *Proceedings of the 1st Conference on Transgressive Computing (TC)*, Granada, Spain, pp. 339-355.

[Smirnova & Watt, 2008] Smirnova, Elena and Watt, Stephen M. (2008). Communicating Mathematics via Pen-Based Computer Interfaces. In Negru, Viorel, Jebelean, Tudor, Petcu, Dana and Zaharie, Daniela, (eds.), *Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Timisoara, Romania, IEEE Computer Society Press, pp. 9-18.

[Suzuki *et al*, 2009] Suzuki, Yu, Misue, Kazuo and Tanaka, Jiro. (2009). Interaction Technique for a Pen-Based Interface Using Finger Motions. In Jacko, Julie A., (ed.), *Proceedings of the 13th International Conference on Human Computer Interaction, Part II: Novel Interaction Methods and Techniques (HCI)*, San Diego, CA, USA, Springer, pp. 503-512.

[Tai, 1979] Tai, Kuo-Chung. (1979). The tree-to-tree correction problem, *Journal of the ACM*, vol 26, no. 3, pp. 422-433.

[van der Hoeven, 2001] van der Hoeven, Joris. (2001). GNU TEXMACS: A free, structured, WYSIWYG and technical text editor, In *Cahiers GUTenberg*, no. 39-40, pp. 39-50.

[Verchinine *et al*, 2008] Verchinine, Konstantin, Lyaletski, Alexander, Paskevich, Andrei and Anisimov, Anatoly. (2008). On Correctness of Mathematical Texts from a Logical and Practical Point of View. In Autexier, Serge, Campbell, John, Rubio, Julio, Sorge, Volker, Suzuki, Masakazu and Wiedijk, Freek, (eds.), *Proceedings of the 7th International Conference on Mathematical Knowledge Management (MKM)*, Birmingham, UK, Springer, pp. 583-598.

[W3C, 1999] W3C. (1999). *XML Path Language (XPath) Version 1.0*, viewed 22 January 2010, <http://www.w3.org/TR/xpath>.

[W3C, 2001] W3C. (2001). *Mathematical Markup Language (MathML) Version 2.0*, viewed 22 January 2010, <http://www.w3.org/TR/MathML2>.

[W3C, 2008] W3C. (2008). *Extensible Markup Language (XML) Version 1.0 (5th Edition)*, viewed 22 January 2010, <http://www.w3.org/TR/xml>.

[Wagner & Lesourd, 2008] Wagner, Marc and Lesourd, Henri. (2008). Using TEXMACS in Math Education: An exploratory Study. In *Proceedings of the 4th Workshop on Mathematical User Interfaces (MATHUI)*, Birmingham, UK.

[Wagner & Müller, 2007] Wagner, Marc and Müller, Christine. (2007). Towards Community of Practice Support for Interactive Mathematical Authoring. In *Proceedings of the 1st Workshop on Scientific Communities of Practice (SCOOP)*, Bremen, Germany.

[Wagner *et al*, 2006] Wagner, Marc, Autexier, Serge and Benzmüller, Christoph. (2006). PLATΩ: A Mediator between Text-Editors and Proof Assistance Systems. In Autexier, Serge and Benzmüller, Christoph, (eds.), *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP)*, Seattle, USA, pp. 65-84.

[Wagner, 2007] Wagner, Marc. (2007). *iMath - Exploratory Study in Math Education*, viewed 22 January 2010, <http://www.marcwagner.info/imath>.

[Wenger, 2005] Wenger, Etienne. (2005). *Communities of Practice: Learning, Meaning, and Identity*, Cambridge University Press.

[Wenzel *et al*, 2008] Wenzel, Makarius, Paulson, Lawrence C. and Nipkow, Tobias. (2008). The Isabelle Framework. In Mohamed, Otmane Ait, Muñoz, César and Tahar, Sofiène, (eds.), *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Montréal, Québec, Canada, Springer, pp. 33-38.

[Wenzel, 2007] Wenzel, Markus. (2007). Isabelle/Isar - A generic framework for human-readable proof documents. In Matuszewski, Roman and Zalewska, Anna, (eds.), *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, University of Bialystok, pp. 277-298.

[Whitehead & Russell, 1910-1913] Whitehead, Alfred N. and Russell, Bertrand. (1910-1913). *Principia Mathematica*, Cambridge University Press.

[Wiedijk, 1998] Wiedijk, Freek. (1998). *The De Bruijn Factor*, viewed 22 January 2010, <http://www.cs.ru.nl/~freek/factor>.

[Wiedijk, 2000] Wiedijk, Freek. (2000). *Estimating the costs of a standard library for a mathematical proof checker*, viewed 22 January 2010, <http://www.cs.ru.nl/~freek/notes/mathstdlib2.pdf>.

[Wiedijk, 2003] Wiedijk, Freek. (2003). Formal Proof Sketches. In Berardi, Stefano, Coppo, Mario and Damiani, Ferruccio, (eds.), *Proceedings of the International Workshop on Types for Proofs and Programs (TYPES)*, Torino, Italy, Springer, pp. 378-393.

[Wiedijk, 2010] Wiedijk, Freek. (2010). *Formalizing 100 Theorems*, viewed 22 January 2010, <http://www.cs.ru.nl/~freek/100>.

[Wilhelm & Maurer, 1997] Wilhelm, Reinhard and Maurer, Dieter. (1997). *Übersetzerbau - Theorie, Konstruktion, Generierung*, 2nd edn, Springer.

[Wilhelm, 1981] Wilhelm, Reinhard. (1981). A modified tree-to-tree correction problem, *Information Processing Letters*, vol 12, no. 3, pp. 127-132.

[Wilhelm, 1984] Wilhelm, Reinhard. (1984). Inverse currying transformation on attribute grammars. In *Proceedings of the 11th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Salt Lake City, Utah, USA, ACM Press, pp. 140-147.

[Wolska & Kruijff-Korbayová, 2004] Wolska, Magdalena and Kruijff-Korbayová, Ivana. (2004). Analysis of Mixed Natural and Symbolic Input in Mathematical Dialogs. In Scott, Donia, (ed.), *Proceedings of the 42nd Annual Meeting on the Association for Computational Linguistics (ACL)*, Barcelona, Spain, ACL, pp. 25-32.

[XML:DB, 2000] XML:DB. (2000). *XUpdate XML Update Language*, viewed 22 January 2010, <http://xmldb-org.sourceforge.net/xupdate>.

[Yellin & Mueckstein, 1985] Yellin, Daniel M. and Mueckstein, Eva-Maria M. (1985). Two-way translators based on attribute grammar inversion. In *Proceedings of the 8th International Conference on Software Engineering (ICSE)*, London, UK, IEEE Computer Society Press, pp. 36-42.

[Yellin & Strom, 1988] Yellin, Daniel and Strom, Rob. (1988). INC: a language for incremental computations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, ACM Press, pp. 115-124.

[Zhang & Shasha, 1989] Zhang, Kaizhong and Shasha, Dennis. (1989). Simple fast algorithms for the editing distance between trees and related problems, *SIAM Journal of Computing*, vol 18, no. 6, pp. 1245-1262.

[Zhang *et al*, 1992] Zhang, Kaizhong, Statman, Rick and Shasha, Dennis. (1992). On the editing distance between unordered labeled trees, *Information Processing Letters*, vol 42, no. 3, pp. 133-139.

[Zhang, 1995] Zhang, Kaizhong. (1995). Algorithms for the constrained editing problem between ordered labeled trees and related problems, *Pattern Recognition*, vol 28, no. 3, pp. 463-474.

[Zimmer & Autexier, 2006] Zimmer, Jürgen and Autexier, Serge. (2006). The MATHSERVE Framework for Semantic Reasoning Web Services. In Furbach, Ulrich and Shankar, Natarajan, (eds.), *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, Seattle, USA, Springer, pp. 140-144.

[Zinn, 2004] Zinn, Claus. (2004). Understanding Informal Mathematical Discourse, PhD thesis, University of Erlangen-Nürnberg.

Index