Universität der Bundeswehr München
Fakultät für Informatik

Consistent Document Engineering

Jan Scheffczyk

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

1. Berichterstatter: Prof. Dr. Uwe M. Borghoff
2. Berichterstatter: Prof. Dr. Wolfram Kahl

Neubiberg, den 03. Dezember 2004

Contents

1 Introduction **2**
1.1 Criteria for Pragmatic Consistency Management 2
1.2 The Approach in this Thesis 3
1.3 Objectives 5
1.4 Outline 5

2 Introductory Survey **9**
2.1 The Running Example for this Thesis 9
2.2 Formalizing Consistency Rules 10
2.3 Finding Inconsistencies 12
2.4 Repairing Inconsistencies 14

I Consistency Checking: Finding Inconsistencies **19**

3 Consistency-Aware Document Management Systems **21**

4 Formalizing Consistency Rules **24**
4.1 Informal Overview 25
4.2 Abstract Syntax 27
4.3 Writing Proper Consistency Rules 29
4.4 Summary 44

5 Finding Inconsistencies **45**
5.1 Informal Overview 46
5.2 Validity of Rules 48
5.3 Generating Consistency Reports 50
5.4 Examples 53
5.5 Formal Structures for Report Generation 56
5.6 Computing Values 62
5.7 Summary 64

6 Speeding up Consistency Checking **66**
6.1 Informal Overview 67
6.2 Static Analysis 69
6.3 Incremental Consistency Checking 75
6.4 Formalizing Efficient Consistency Rules 87
6.5 Summary 87

II Consistency Maintenance: Repairing Inconsistencies 88

7 Towards Consistency Maintenance 90
7.1 How Can We React to Inconsistencies? . . . 90
7.2 Generating Repairs — A First Account . . . 93
7.3 Feasible Document Repair Generation: A Two-Step Approach . 93
7.4 Consistency Maintaining DMSs . . . 95

8 S-DAGs: Towards Efficient Document Repair Generation 97
8.1 Informal Overview . . . 97
8.2 Hints for Better Repair Actions . . . 101
8.3 Describing Repair Actions by S-DAGs . . . 104
8.4 Generating S-DAGs . . . 108
8.5 Interactive Repair . . . 122
8.6 Summary . . . 127

9 Repair Collections 129
9.1 Informal Overview . . . 130
9.2 Deriving Repair Collections from S-DAGs . . . 133
9.3 Repairing Repositories . . . 145
9.4 Summary . . . 147

III Case Study: Maintaining Consistency in Industrial Software Specifications 149

10 Analysis Modules 151
10.1 Industrial Specifications are Heterogeneous Document Sets . . 151
10.2 Analysis Modules and Consistency Requirements . . . 152
10.3 Summary . . . 165

11 Formalizing Consistency Rules 167
11.1 Developing the Language SDM . . . 167
11.2 Formalizing Consistency Rules . . . 172

12 The Ski School 179
12.1 The Ski School Specification . . . 179
12.2 Developing the Specification . . . 191
12.3 Sample S-DAGs . . . 197
12.4 Sample Repair Collections . . . 208
12.5 Performance Summary . . . 211

13 Costs and Benefits of Consistency Management 215

IV Conclusions **218**

14 Comparison with Related Work **219**
14.1 Consistency Checking 219
14.2 Incremental Evaluation 221
14.3 Consistency Maintenance 223
14.4 Software Engineering Tools 225

15 Conclusions and Outlook **227**
15.1 Summary 227
15.2 Future Research 229
15.3 Applications 230

A Notation **232**

B Implementation **234**
B.1 Consistency Checking in Practice 235
B.2 Accessing Repositories 236
B.3 Haskell Meets Subtypes 237
B.4 Defining Symbol Semantics 240

C Proofs **244**

List of Figures **253**

List of Tables **256**

List of Definitions and Theorems **257**

Bibliography **258**

Abstract

When a group of authors collaboratively edits interrelated documents, consistency problems occur almost immediately. Current document management systems (DMSs) provide useful mechanisms such as document locking and version control, but often lack consistency management facilities. At best, consistency is "defined" via informal guidelines, which do not support automatic consistency checks.

In this thesis, we complement traditional DMSs by consistency management. We propose to use *formal* consistency rules that capture semantic consistency requirements. Rules are formalized in a variant of temporal logic. A static type system supports rule formalization, where types also define (formal) document models. In implementing a *tolerant* view of consistency, we do not expect that the documents satisfy consistency rules. Instead, our novel semantics precisely pinpoints inconsistent document parts and indicates *when*, *where*, and *why* documents are inconsistent. *Speed* is a key issue in consistency management. Therefore, we develop efficient techniques for consistency checking while retaining our tolerant semantics.

Just pinpointing inconsistencies is, however, insufficient for flexible consistency management. We extend our consistency checking approach towards suggesting *repairs*, which resolve inconsistencies. The critical issues are to suggest only some of the best (i.e., least costly) repairs and to generate repairs efficiently. Therefore, we develop a new two-step approach. First, we employ directed acyclic graphs (DAGs) to carry repairs. These graphs are called *suggestion DAGs* (short: S-DAGs). In contrast to the enumeration of all possible repairs, S-DAGs provide a suitable means to generate repairs efficiently and to limit the search space for good repairs. Second, from S-DAGs, we derive one repair collection for all consistency rules. Due to the separation of repair derivation from S-DAG generation, the repository is locked during the computationally cheap S-DAG generation only.

We have implemented a prototype of a consistency management tool. Our case study in the field of software engineering shows that our contributions can significantly improve consistency management in document engineering and scale to a practically relevant problem size.

Chapter 1
Introduction

Larger bodies of writing, such as books, technical documentations, or software specifications, contain many interrelated documents. Typically, a whole team of authors is responsible for producing these documents. In supporting a multi-author environment, document management systems (DMSs) store documents in repositories. On the one hand, DMSs provide fundamental management facilities, e.g., version management, access control, or deployment management. On the other hand, DMSs fail to manage semantic domain-specific consistency requirements. Usually, authors aim to produce an overall consistent work, i.e., certain relations between the documents are maintained. These relations are, however, mostly implicit and vague, e.g., "Links within documents must have a valid target." In order to achieve consistency, authors have to spend plenty of time re-reading and revising their own and related documents. Worse, each check-in to the repository potentially violates consistency. Larger companies define guidelines and policies for writing; but still a *human* reviewer is needed to maintain them. What prevents automatic consistency checks is that guidelines are implicit or at best informal.

Inconsistencies are major obstacles causing time-consuming manual effort, which results in either schedule delays or budget holes or both. Thus, it is high time to integrate consistency management into DMSs. On the other hand, inconsistencies are natural in a multi-author environment. Neither can they be ignored, nor can they be forbidden.

1.1 Criteria for Pragmatic Consistency Management

A *pragmatic* consistency management approach detects inconsistencies automatically and suggests appropriate inconsistency handling strategies [SZ01]. In the literature, detecting inconsistencies is often referred to by the term "consistency checking;" handling inconsistencies is often referred to by the term "consistency maintenance."

Most importantly, consistency management should help authors, i.e., it must not hinder their work nor enforce new document editing practices. In order to accomplish integration into document engineering processes and work flows, consistency management should support to restrict the evolution of documents over time. Of course, this is the main reason for extending DMSs, instead of developing a stand alone consistency management tool. The market for DMSs and revision control systems is large, as is the variety of document models and document formats (although we recognize a strong tendency towards XML). Therefore, consistency management should only make a few assumptions about the underlying DMS, document formats, and document models. Naturally, inconsistencies are major obstacles in larger document engineering projects only.

Thus, consistency management must scale to a practically relevant problem size, where the term "problem size" means the number of documents, their heterogeneity, and the number of consistency requirements. *Speed* is a key to user acceptance. After a check-in to the repository, authors want to know almost immediately whether this check-in is accepted and how it meets the consistency rules.

In many areas, (temporary) toleration of inconsistencies is considered vital for flexible consistency management [Fin00, NER00]. For example, if documents evolve at different rates, enforcing consistency may cause deadlocks. Consistency requirements may be too strict for some purposes — an inconsistency may indicate an exception or a design alternative. Finally, the impact of an inconsistency can be low compared to the costs of resolving it. Thus, the central questions are about *how* inconsistent a repository is and *what* can be done to handle inconsistencies. We do not ask whether inconsistencies occur at all — we know that already. For flexible inconsistency resolution, domain knowledge should be incorporated into consistency management. Also, resolution strategies may change as the documents evolve in time.

Clearly, the above criteria call for a tradeoff between formality and pragmatism. Entirely formal approaches provide strong means for consistency management, but severely hinder the work of authors. Informal approaches have proven insufficient for consistency management.

1.2 The Approach in this Thesis

In this thesis, we develop a formal consistency management approach that is built on top of informal document engineering processes. We propose the use of explicit formal *consistency rules* to capture informal consistency requirements. We do not require any adaptions to editing practices of authors. One can consider our approach located "between" formal consistency management and informal consistency management: Our rules are formal, the documents may be informal.

By "consistency checking" we mean to determine how the repository conforms to the rules. This is in contrast to classic logic, where "consistency checking" means to determine whether a set of formulae has a model, i.e., all formulae can be fulfilled at once. We smoothly integrate consistency checking into arbitrary DMSs *without* requiring adaptations to document engineering processes. Consistency rules can express intra- and inter-document requirements regardless of the document model and the document format used. *Strong* rules must be adhered to, whereas *weak* rules may be violated. Rules can restrict how documents evolve in time — hence, we formalize rules in a *temporal* logic. Since rule design is a complex task, a static type system helps to define syntactically well-formed consistency rules. Being aware that check-ins potentially violate consistency rules, we refine traditional boolean semantics by a novel semantics: *Consistency reports* pinpoint inconsistencies within documents precisely. Since the time needed for consistency checking is crucial, we develop methods that

evaluate consistency rules efficiently. In contrast to many other consistency checking approaches, we retain our tolerant semantics.

Checking consistency is, however, only the first step towards flexible consistency management. In this thesis, we also propose strategies to *repair inconsistencies*. Exponential computational complexity of repair enumeration motivates our new two-step approach: In the first step, we describe repair actions by directed acyclic graphs (DAGs). We call these DAGs *suggestion DAGs* (short: S-DAGs). S-DAGs are optimized for efficient generation and also provide a convenient way to visualize inconsistencies and repair actions. A major advantage of S-DAGs is that they can be reduced effectively, such that only the best repair actions remain. Annotations to consistency rules guide S-DAG generation and make domain knowledge available for generating useful repair actions. From S-DAGs, authors can choose actions. S-DAGs provide a computationally tractable approach to generating useful repair actions, but they suffer from an inherent weakness: Authors pick actions separately for each rule, independent of their effect on other rules. If, however, many rules are violated the interaction between actions and potential impacts regarding overall consistency remain unclear. Therefore, in the second step, we derive *a single repair collection for all consistency rules.*[1] The repair collection consists of alternative repair sets, each of which contains repairs that resolve all inconsistencies in the repository. The collection can be sorted w.r.t. user-defined metrics, based on repair ratings. Since the repair collection is derived from S-DAGs only, the repository does not need to be locked during repair derivation.

We implement our consistency checking and consistency maintenance approaches in a prototype system. This thesis provides a case study, in which we apply our consistency maintenance techniques to a document based approach to software specifications developed at sd&m, a well-established German software company.[2] Inconsistencies are major obstacles in industrial software specifications. Our case study gives strong evidence that our tolerant consistency maintenance approach is a feasible way to handle inconsistencies. The case study also shows that our techniques are a useful aid and scale to a complex scenario. Efforts for formalization remain within reasonable limits. The performance of our prototype system is satisfactory.

In this thesis, we use and adapt techniques known from databases and a recent approach towards consistency management for distributed XML documents. Usually, databases implement a strict view of consistency and benefit from a formal database model and a formal update model. In contrast, we implement a tolerant view of consistency and apply our techniques to heterogeneous documents managed by a DMS. The toolkit xlinkit [NEF01, NCEF02, DENT02, NEF03, PNEF03] can be used to manage consistency between *distributed* documents. Xlinkit tolerates inconsistencies, too. Distribution of documents and thus lack of control prohibits temporal consistency rules and hin-

[1] Throughout, we use the term “repair collection” to mean a set of sets of repairs.

[2] sd&m: software design & management AG (a company of Capgemini), see www.sdm.de and www.capgemini.com

ders efficient consistency checking techniques. For readability, we postpone a detailed discussion of related work to Chapter 14.

With kind permission of Universität der Bundeswehr München, preliminary results of this thesis have been published as [SBRS03a, SBRS03b, SBRS04c, SBRS04b, SBRS04a, SSBS04].

1.3 Objectives

The objectives of this thesis are as follows:

- We smoothly integrate tolerant consistency management into arbitrary DMSs without requiring adaptations in document engineering practices already used. Our approach is independent from specific document formats and models. Thus, we facilitate heterogeneous repositories.
- We capture semantic consistency requirements within and between documents by user-defined consistency rules. *Strong* rules must be adhered to; *weak* rules may be violated.
- We point out exactly when, where, and why documents are inconsistent w.r.t. the formalized rules.
- We develop techniques to efficiently check consistency, where our tolerant semantics is retained. We conjecture that our techniques are useful for other research areas as well.
- We introduce effective and efficient methods to derive repairs that can resolve inconsistencies. We establish a new two-step approach: During consistency checking, we generate for each rule an S-DAG. On author demand, we derive a single repair collection from all S-DAGs. Annotations to consistency rules guide S-DAG generation and repair derivation.
- We prove the usefulness of our techniques by a case study in a complex scenario: industrial software specifications.

Throughout this thesis, an important constraint guides our action: Our techniques must be applicable and useful in practice.

1.4 Outline

This thesis is organized as follows: In Chapter 2, we introduce our running example that we use throughout for illustrating our techniques. We also survey the most important results of this thesis from the user perspective. The actual thesis is divided into four parts.

In Part I, we develop methods to *check consistency* in heterogeneous repositories. In Chapter 3, we show how our approach is integrated into arbitrary DMSs. Chapter 4 is concerned with the syntactical issues of rule formalization. We present our abstract syntax and a static type checking algorithm,

which ensures well-typedness of rules. In Chapter 5, we introduce our new tolerant semantics and define a basic consistency checking algorithm. In Chapter 6, we concentrate on efficiently checking consistency rules against a heterogeneous repository. Preliminary results of this part have been published as [SBRS03a, SBRS03b, SBRS04c].

In Part II, we complement our consistency checking approach by *consistency maintenance*. In Chapter 7, we discuss basic design decisions and show how we integrate consistency maintenance into the every-day work with a DMS. Chapter 8 is concerned with S-DAGs as a means to effectively model inconsistencies and repair actions. For efficient S-DAG generation, we use techniques from Chapter 6. In Chapter 9, we derive repair collections from S-DAGs. Preliminary results of this part have been published as [SBRS04b, SBRS04a].

In Part III, we prove the usefulness of our techniques, developed in the previous parts, by our *case study*. We apply consistency maintenance to industrial software specification documents. In Chapter 10, we introduce analysis modules, developed at sd&m, as a means to build specifications for large systems. Also, we identify consistency requirements between these modules. In Chapter 11, we formalize some of these requirements by consistency rules. In Chapter 12, we develop an example specification and show how software engineering can benefit from our consistency maintenance approach. In Chapter 13, we summarize the most important lessons learnt from our case study. A sketch of our case study is published as [SSBS04].

Part IV concludes this thesis. We discuss related work in detail in Chapter 14. In Chapter 15, we summarize this thesis, explore directions for future research, and outline possible applications. Moreover, we give advice to readers about how they can benefit from the contributions of this thesis.

Throughout this thesis, we use formal notations necessary for our algorithms. In App. A, we list these notations briefly, such that the reader can look up notations quickly. The implementation of our prototype is outlined in App. B. For brevity, we give a short presentation only, which also may become out of date. Therefore, we would like to point the reader to our project WWW site

www2-data.informatik.unibw-muenchen.de/cde.html

for up-to-date information. App. C contains proofs of theorems.

Throughout this thesis, *dependency graphs* indicate relationships between chapters. Of course, one can read this thesis chapter by chapter. In addition, we provide three alternative reading orders, depending on whether the reader's focus is on consistency checking, consistency maintenance, or the application of consistency maintenance to software engineering. Fig. 1.1 shows the global dependency graph for this thesis, which will be refined in the individual chapters. Each node represents a chapter; it may be annotated by a reference to a publication about the chapter's topic. Bold edges between nodes represent recommended reading order; dotted edges denote additional content dependencies. For example, when reading Chapter 8 (S-DAGs) readers should have read Chapter 1 (introduction), 2 (survey), and 7 (repair options). They might also want to lookup some content in Chapter 4 (abstract syntax) and 6 (efficient consistency checking).

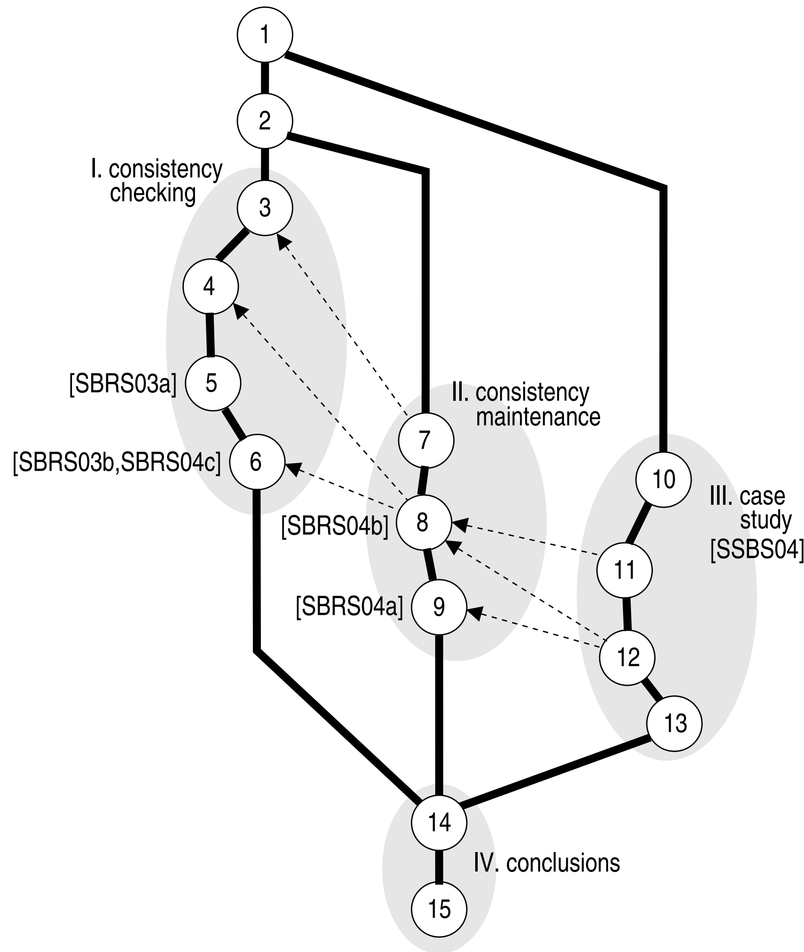


Figure 1.1: Relationships between chapters (bold edges point downwards and denote recommended reading order, dotted edges denote additional content dependencies)

Acknowledgements

Many people have provided most valuable ideas for this thesis. For brevity, I cannot list all of them; some colleagues, however, should receive special mention. First of all, I would like to thank my supervisor Uwe M. Borghoff. He has spent considerable effort in pointing me to "the right directions" and in discussing my half-baked ideas. I am grateful to my advisor Wolfram Kahl for his invaluable comments and advice about how to express practical problems by sound formalisms. The staff at the Institute for Software Technology provided a good research climate. In particular, I would like to thank Lothar Schmitz, Peter Rödig, and Michael Ebert for many fruitful discussions. I am confident that without intellectual support from sd&m this thesis would lack a lot of practically relevant contributions. Christiane Stutz, Johannes Siedersleben, and Andreas Birk contributed precious experience from practice, which "forced" my theoretical thoughts to a practical setting. Also, I would like to

thank the anonymous reviewers from conferences and journals who provided valuable comments on our submissions. Finally, I am grateful for financial support from the Universität der Bundeswehr München for conference and journal contributions, and for the publication of this thesis.

Chapter 2
Introductory Survey

In this chapter, we survey major results of this thesis by a small and simple example, which we shall use throughout. Far more complex examples that raise closely related consistency problems can be found in [Min83, SWJF96]. Although our example may appear too simple at first sight, we use it in order to *illustrate* our formal approach. We shall see that even this simple example causes interesting consistency issues. For a more realistic scenario, see our case study in Part III. First, we introduce our running example. In Sect. 2.2, we present formalized versions of consistency requirements in this example. Sect. 2.3 illustrates how inconsistencies are presented to users. Sect. 2.4 outlines our approach towards consistency maintenance. Fig. 2.1 illustrates the context of this chapter.

2.1 The Running Example for this Thesis

Assume that we want to archive manuals over a long period of time. Documents (and manuals) reference manuals through keys (see Fig. 2.2). Since names and kinds of manuals may change over time, we need key resolvers mapping keys to their semantics, i.e., manual kind and name. Multiple key resolvers may exist. Their actual names are hidden from authors. In order to ensure consistency, we require that (1) (two-step) links to manuals are valid and (2) names and kinds of manuals are invariant over time. A reference to a key k is valid, if k is defined in a key resolver and this definition points to an existing manual of the correct kind.

Let a toy repository develop as shown in Fig. 2.2. Initially, the repository consists of a plain text document doc1.txt and two XML documents: a key resolver keys.xml and a technical manual man1.xml. The kind of man1.xml is changed towards field manual by the second check-in. By the third check-in, a new text document doc2.txt is added. Then some new manuals are checked in. Finally, a new key resolver keys2.xml is added and the key definition for the key kaA3 in keys.xml is changed. Such a sequence of check-ins may appear, e.g., if text documents, manuals, and key resolvers are maintained by different

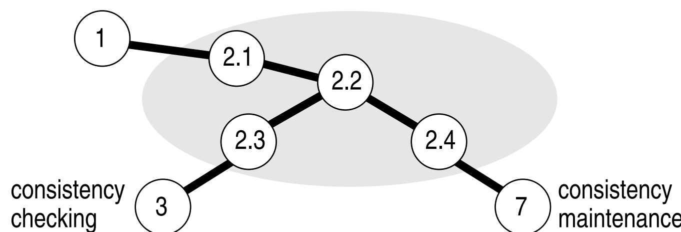
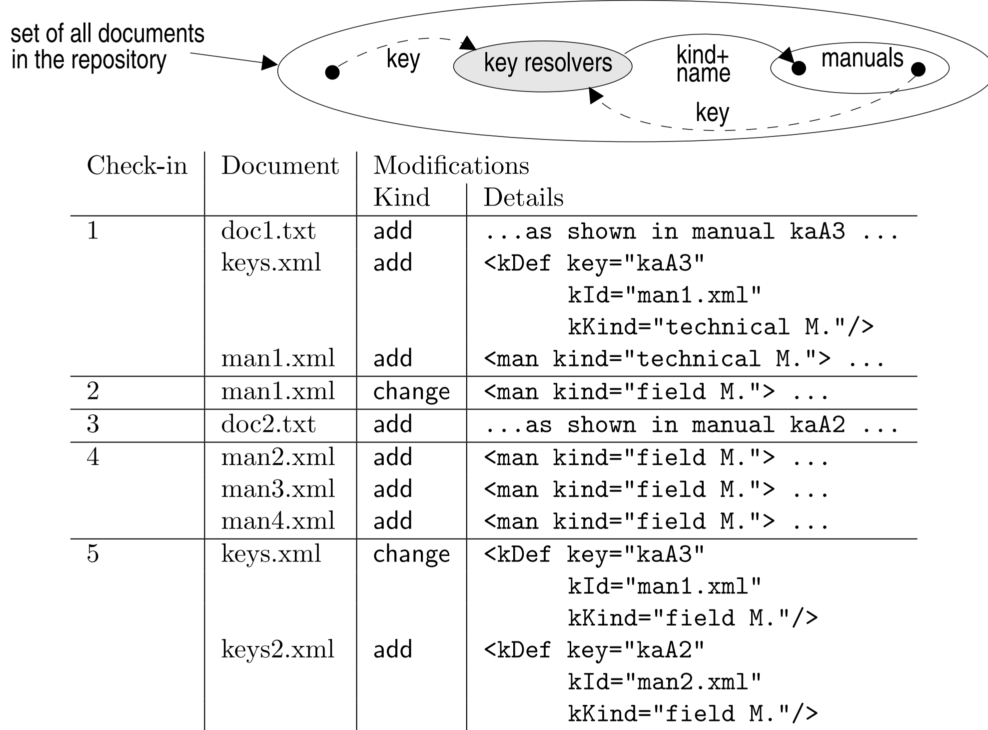


Figure 2.1: Chapter 2 in context



Check-in	Document	Modifications: Kind	Details
1	doc1.txt	add	...as shown in manual kaA3 ...
	keys.xml	add	<kDef key="kaA3" kId="man1.xml" kKind="technical M."/>
	man1.xml	add	<man kind="technical M."> ...
2	man1.xml	change	<man kind="field M."> ...
3	doc2.txt	add	...as shown in manual kaA2 ...
4	man2.xml	add	<man kind="field M."> ...
	man3.xml	add	<man kind="field M."> ...
	man4.xml	add	<man kind="field M."> ...
5	keys.xml	change	<kDef key="kaA3" kId="man1.xml" kKind="field M."/>
	keys2.xml	add	<kDef key="kaA2" kId="man2.xml" kKind="field M."/>

Figure 2.2: Example repository for this thesis

authors. As it stands, our repository violates the consistency requirements. For example, the second check-in introduces two inconsistencies: (1) the key reference to kaA3 is inconsistent, because the kind of man1.xml is different from the kind of the key definition for kaA3; (2) the kind of the manual man1.xml has changed.

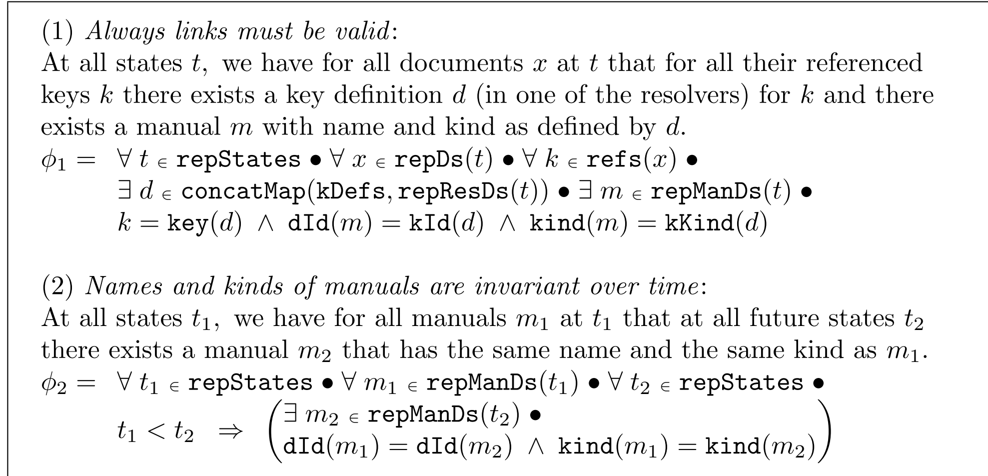
One might argue that, instead of our tolerant approach, stricter control over the kinds of manuals should be employed. For a repository in development, however, this is infeasible, because changes in the manual kinds are sometimes necessary resulting in the need to change the key resolver entry as well. We shall see later that, in general, we cannot employ automatic repairs (as done in active databases) because of the complex informal semantics of documents.

2.2 Formalizing Consistency Rules

In order to detect inconsistencies automatically, we must capture our consistency requirements formally. As shown in Fig. 2.3, we formalize our requirements via *formal consistency rules* using a certain kind of predicate logic.

Rule ϕ_1 first quantifies over all repository states,[1] provided by `repStates`. We obtain the current documents for a state t by `repDs`(t). For each document x, the variable k comprises the referenced keys, computed via `refs`(x). Key

[1] A repository *state* corresponds to a snapshot of the repository at a given time. A check-in to the repository causes a state transition.



(1) *Always links must be valid*:
At all states t, we have for all documents x at t that for all their referenced keys k there exists a key definition d (in one of the resolvers) for k and there exists a manual m with name and kind as defined by d.

$$\begin{aligned}\phi_1 = \;& \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\ & \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \exists\, m \in \texttt{repManDs}(t) \bullet \\ & k = \texttt{key}(d) \;\wedge\; \texttt{dId}(m) = \texttt{kId}(d) \;\wedge\; \texttt{kind}(m) = \texttt{kKind}(d)\end{aligned}$$

(2) *Names and kinds of manuals are invariant over time*:
At all states t_1, we have for all manuals m_1 at t_1 that at all future states t_2 there exists a manual m_2 that has the same name and the same kind as m_1.

$$\begin{aligned}\phi_2 = \;& \forall\, t_1 \in \texttt{repStates} \bullet \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \forall\, t_2 \in \texttt{repStates} \bullet \\ & t_1 < t_2 \;\Rightarrow\; \begin{pmatrix}\exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{dId}(m_1) = \texttt{dId}(m_2) \;\wedge\; \texttt{kind}(m_1) = \texttt{kind}(m_2)\end{pmatrix}\end{aligned}$$

Figure 2.3: Formal example rules

definitions in the resolvers are computed with the help of `concatMap`, which applies `kDefs` to each key resolver, returned by $\texttt{repResDs}(t)$. When applied to a key resolver, `kDefs` returns a list of key definitions. Finally, the resulting lists are concatenated. Thus, d ranges over the key definitions from all key resolvers. The variable m ranges over all manuals at state t, obtained by $\texttt{repManDs}(t)$. We require that the key k equals the key defined by the key definition d, and the name of the manual m equals the name d points to,[2] and the kind of m equals the kind d points to. In some cases it might seem appropriate to neglect the case of letters when comparing manual kinds; then we would simply use another predicate symbol, say $\equiv$, which neglects case. Clearly, this is beyond the possibilities offered by link checkers.

In rule ϕ_2, we twice quantify over time, because we have to relate different versions of manuals: the old version at state t_1, the new version at state t_2.

The rules in Fig. 2.3 appear more complex than the vague requirements in the previous section. Formal rules are much more *precise* than informal guidelines and give no room for misinterpretations: When formalizing consistency rules, we have to decide what is actually required in our application. In our experience, formalization has contributed to a common understanding of what consistency means. This is vital for any collaborative work.

Formalizing consistency rules may become complex. Therefore, we divide formalization into different tasks and supply tools to every stake-holder. Chapter 3 shows how consistency rules are formalized in a multi user environment and how we integrate consistency checking into the every-day work with a DMS. In particular, if complex higher-order functions and predicates are used, formalization of rules is prone to errors. Therefore, we provide users with a static type system, which indicates syntactic formalization errors at the time

[2] By $\texttt{dId}(m)$ we denote the name of the document m; $\texttt{dState}(m)$ denotes the check-in state of m.

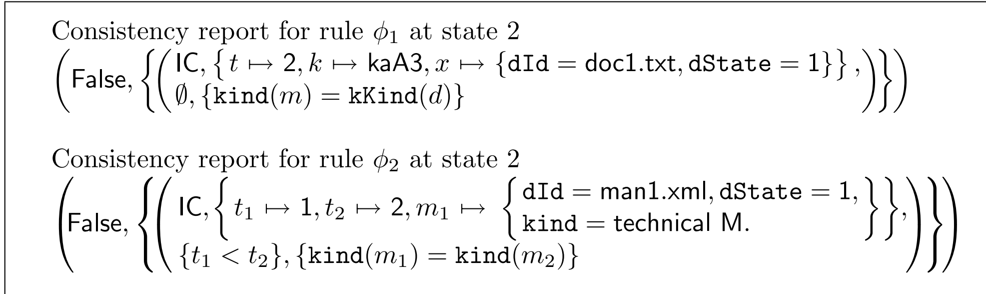


Figure 2.4: Example consistency reports at state 2

of rule definition, i.e., prior to evaluating the rules. We discuss the syntax part of our approach in Chapter 4.

2.3 Finding Inconsistencies

Our consistency checker can perform a consistency check after a check-in to the repository. This means to pinpoint the trouble spots that make a repository inconsistent w.r.t. the rules formalized. Our primary goal is to provide minimum information that precisely characterize when, where, and why a repository is inconsistent. For each rule, we compute a *consistency report* containing a boolean result (representing boolean truth semantics) and a diagnosis set. Each diagnosis consists of a (redundant) consistency flag, a variable assignment, fulfilled atomic formulae, and violated atomic formulae.[3] A diagnosis $(\mathsf{IC}, \eta, ps_t, ps_f)$ reads: "The processed rule is violated (InConsistent) for the variable assignment η, due to fulfilled atomic formulae in the set ps_t and violated atomic formulae in the set ps_f." The assignment η binds variables of rules to concrete values, i.e., repository states, documents, or document content. Due to quantification over repository states, it indicates when and where a rule is violated. The sets ps_t and ps_f give reasons for rule violation.

At the first state, both consistency rules are fulfilled. Hence, our consistency checker generates the report $(\mathsf{True}, \emptyset)$ for each consistency rule.

At the second state, our consistency checker generates the consistency reports shown in Fig. 2.4. The report for rule ϕ_1 reflects the inconsistency introduced by the wrong kind of man1.xml. The report's assignment does not contain the variables d and m: The repository is inconsistent w.r.t. ϕ_1 for all possible bindings to d and m. The report also lacks the atomic formulae $k = \mathtt{key}(d)$ and $\mathtt{dId}(m) = \mathtt{kId}(d)$: The key resolver contains a definition for the key $\mathsf{kaA3}$, but this definition is inconsistent. This means that manuals m with the correct name were found but that they have the wrong kind: The first step of the link is consistent, the second step is inconsistent. The inconsistency introduced by the change of the manual kind is reflected by the report for rule

[3] In the final consistency report, consistency flags are redundant, because they are equal for all diagnoses. We will see, however, that consistency flags are important for generating consistency reports.

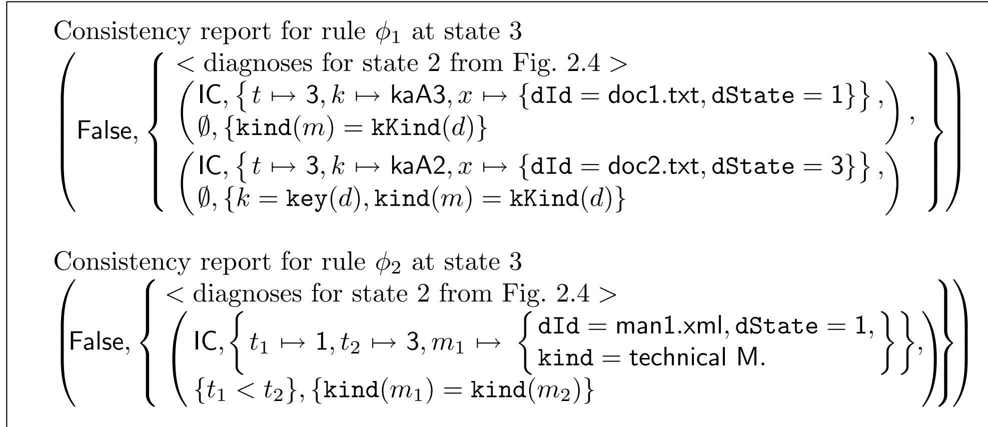


Figure 2.5: Example consistency reports at state 3

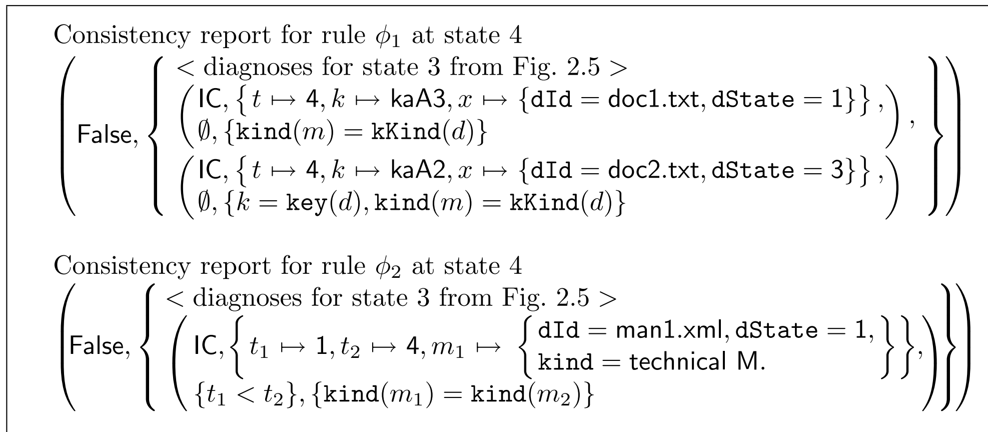


Figure 2.6: Example consistency reports at state 4

ϕ_2: For the manual man1.xml at state 1, no manual at state 2 could be found with the same kind. The report does not include $\mathtt{dId}(m_1) = \mathtt{dId}(m_2)$, because there exists a manual man1.xml at state 2: Its kind is wrong only.

At the third state, our consistency checker generates the reports shown in Fig. 2.5. They contain all diagnoses from the previous consistency check, because our rules quantify over all repository states. Adding the document doc2.txt introduces a new inconsistency w.r.t. rule ϕ_1: There is no key resolver defining the key kaA2. The other diagnoses are already known: They directly follow from inconsistencies introduced in previous repository states. Previous inconsistencies are still present, because we did not resolve them. Incremental consistency checking, discussed in Chapter 6, exploits this natural property.

Consistency reports at state 4 can be found in Fig. 2.6. Fig. 2.7 shows the consistency reports at state 5. At that state, all links are consistent.

We detail our algorithm for generating consistency reports in Chapter 5. As one would expect, speed is a critical issue when generating consistency reports, because the repository must be locked during a consistency check. In

Consistency report for rule ϕ_1 at state 5
$(\mathsf{False}, \{< \text{diagnoses for state 4 from Fig. 2.6} >\})$

Consistency report for rule ϕ_2 at state 5

$$\left(\mathsf{False}, \left\{ \begin{array}{l} < \text{diagnoses for state 4 from Fig. 2.6} > \\ \left(\mathsf{IC}, \left\{ t_1 \mapsto 1, t_2 \mapsto 5, m_1 \mapsto \left\{ \begin{array}{l} \mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 1, \\ \mathtt{kind} = \mathsf{technical\ M.} \end{array} \right\} \right\}, \right. \\ \left. \{t_1 < t_2\}, \{\mathtt{kind}(m_1) = \mathtt{kind}(m_2)\} \right) \end{array} \right\} \right)$$

Figure 2.7: Example consistency reports at state 5

Chapter 6, we develop methods that significantly reduce the time needed for report generation.

2.4 Repairing Inconsistencies

Consistency reports point out inconsistencies precisely. A consistency report is, however, no constructive means — it just shows deficiencies in the repository. In this section, we explore aids to resolve inconsistencies. We use a two-step approach towards generating repairs. The main reason is that we want to efficiently generate a few repairs that make sense. In the first step, we describe for each rule possible repair actions by an S-DAG. In the second step, we derive a single *repair collection* from all S-DAGs. Chapter 7 shows how we integrate this two-step approach into the every-day work with a DMS.

2.4.1 S-DAGs

The structure of an S-DAG resembles that of a consistency rule. Nodes represent logical connectives or atomic formulae; edges target the subformulae of a connective. We distinguish the following kinds of nodes: Universal nodes ($\forall$) and existential nodes ($\exists$) represent universal and existential quantification, respectively. Outgoing edges carry value bindings to the quantified variable. A value represents a repository state, a document, or document content. Conjunction nodes ($\wedge$) and disjunction nodes ($\vee$) stand for conjunctions and disjunctions, respectively.[4] A predicate leaf contains an atomic formula ϕ that causes an inconsistency, the truth value of ϕ, and a predicate suggestion collection.[5] The predicate suggestion collection contains alternative suggestion sets, each of which contains suggestions that indicate how the truth value of ϕ can be changed.

From the user perspective, a universal node represents inconsistencies resulting from *dubious* document content. Each edge blames a value for inconsistencies represented by the edge's target S-DAG. An existential node represents

[4]Implications are replaced by disjunctions: For formulae ϕ and ψ, $\phi \Rightarrow \psi$ is equivalent to $\neg\phi \vee \psi$. Also, we will see that negation nodes are not necessary.

[5]Throughout, we use the term "predicate suggestion collection" to mean a set of sets of predicate suggestions.

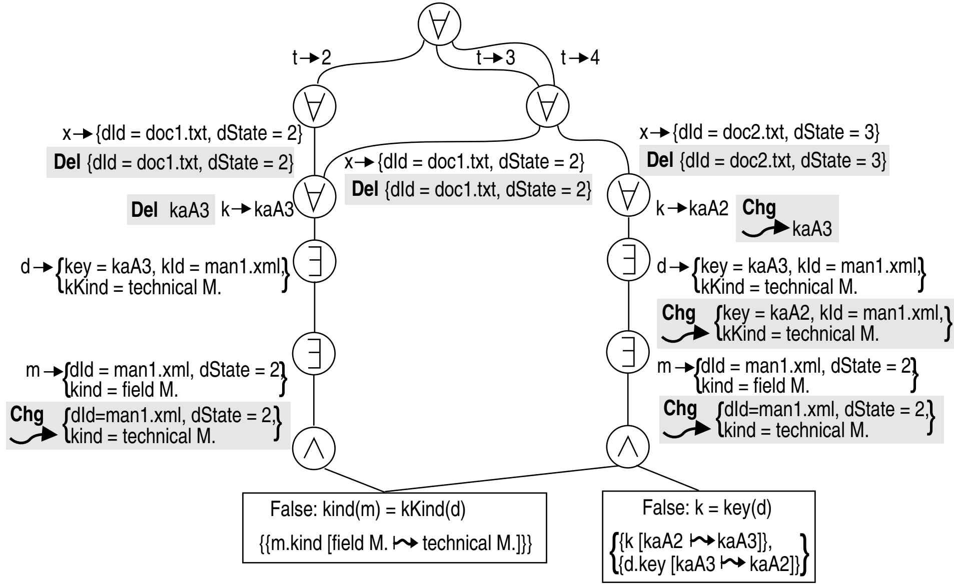


Figure 2.8: S-DAG for rule ϕ_1 at state 4

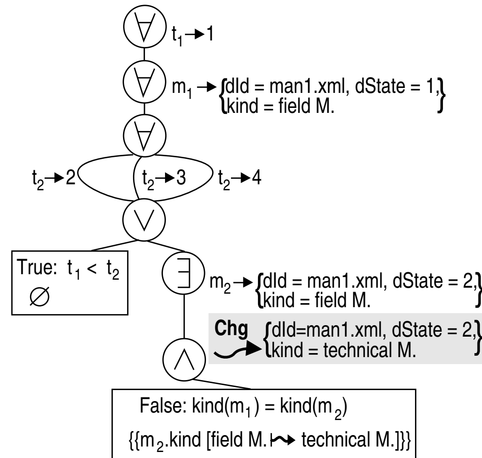


Figure 2.9: S-DAG for rule ϕ_2 at state 4

an inconsistency resulting from *missing* document content. This content could be either really missing or it could be regarded as missing, because one of the edges carries defective content. Below both conjunction nodes and universal nodes each S-DAG must be repaired. In contrast, it is sufficient to repair only one S-DAG below a disjunction node or an existential node.

Fig. 2.8 shows the S-DAG for rule ϕ_1 at state 4. In the leaves, we find inconsistent atomic formulae and the reasons for their failure. For example, the left hand leaf indicates that the atomic formula $\texttt{kind}(m) = \texttt{kKind}(d)$ is violated, if m is bound to the manual man1.xml and d is bound to the key definition for kaA3. We can grasp these bindings by following the paths from the S-DAG root to this leaf. Quantifier edges also carry repair actions marked

$$
\begin{aligned}
rep_{\mathsf{Chgman1}} &= \mathsf{Rep}\ \mathtt{repManDs}(4)\\
&\qquad \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 2, \ldots\}.\mathtt{kind} \rightsquigarrow \mathsf{technical\ M.}\\
rep_{\mathsf{Chgdoc2}} &= \mathsf{Rep}\ \mathtt{refs}(\{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 2\})\\
&\qquad \mathsf{Chg}\ \mathsf{kaA2} \rightsquigarrow \mathsf{kaA3}\\
rep_{\mathsf{Chgkeys}} &= \mathsf{Rep}\ \mathtt{concatMap}(\mathtt{kDefs}, \mathtt{repResDs}(4))\\
&\qquad \mathsf{Chg}\ \{\mathtt{key} = \mathsf{kaA3}, \ldots\}.\mathtt{key} \rightsquigarrow \mathsf{kaA2}\\
rep_{\mathsf{DelkaA2}} &= \mathsf{Rep}\ \mathtt{refs}(\{x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\})\\
&\qquad \mathsf{Del}\ \mathsf{kaA2}\\
rep_{\mathsf{Deldoc2}} &= \mathsf{Rep}\ \mathtt{repDs}(4)\\
&\qquad \mathsf{Del}\ \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}
\end{aligned}
$$

Figure 2.10: Repairs generated for rules ϕ_1 and ϕ_2 at state 4

grey. An action proposes to either add a value to (Add), or change a value within (Chg), or delete a value (Del) from the quantifier sphere. For example, we propose to change the key kaA2 to kaA3 or to delete the document doc1.txt. Fig. 2.9 shows the S-DAG for rule ϕ_2 at state 5. Here, we have replaced the implication $t_1 < t_2 \Rightarrow \exists \ldots$ by the disjunction $\neg(t_1 < t_2) \vee \exists \ldots$. Notice the sharing below the universal node for the variable t_2.

S-DAGs only contain repair actions that propose minimal changes to the repository. For example, the S-DAG for ϕ_1 does not propose to change the name and the kind of the manual man2.xml, in order to repair the broken key definition for kaA3. Also, the S-DAG for ϕ_2 does not propose to change repository states. In Chapter 8, we discuss the challenge of how to generate meaningful S-DAGs.

S-DAGs contain repair actions already. It remains, however, unclear which actions must be applied together, in order to resolve all inconsistencies w.r.t. the corresponding rule. In our interactive approach, authors walk through S-DAGs and choose actions manually. Our system can apply an action to its associated S-DAG, which results in an empty S-DAG if all inconsistencies are resolved. Otherwise, the remaining inconsistencies can be resolved by choosing another action.

2.4.2 Repair Collections

Instead of using the above trial and error approach to determine reasonable repairs, we can also derive a repair collection for all rules automatically. The collection consists of repair sets, each of which is an alternative. Repair sets contain repairs that must be performed together, in order to resolve all inconsistencies for all rules.

Fig. 2.10 shows some repairs derived from our example S-DAGs at state 4. The first component of a repair constitutes its sphere, the second component denotes the proposed action. For example, the repair rep_{Chgman1} proposes to change the kind of the manual man1.xml towards technical M., where man1.xml resides in the sphere calculated via `repManDs` applied to the state 4 (i.e., man1.xml is one of the manuals at state 4).

$$
\begin{array}{ll}
\phi_1 = & \forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\
& \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \exists\, m \in \texttt{repManDs}(t) \bullet \\
& \quad k = \texttt{key}(d) \quad \left\{ \begin{array}{l} \{k \rightsquigarrow \texttt{key}(d)\ \mathsf{False}\ \}, \\ \{d.\texttt{key} \rightsquigarrow k\ \mathsf{False}\ \} \end{array} \right\} \quad \wedge \\
& \quad \texttt{dId}(m) = \texttt{kId}(d) \quad \{\{m.\texttt{dId} \rightsquigarrow \texttt{kId}(d)\ \mathsf{False}\ \}\} \quad \wedge \\
& \quad \texttt{kind}(m) = \texttt{kKind}(d) \quad \{\{m.\texttt{kind} \rightsquigarrow \texttt{kKind}(d)\ \mathsf{False}\ \}\} \\
\\
\phi_2 = & \forall\, t_1^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, m_1^{\mathsf{KEEP}} \in \texttt{repManDs}(t_1) \bullet \forall\, t_2^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \\
& t_1 < t_2 \Rightarrow \\
& \quad \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\
& \quad\quad \texttt{dId}(m_1) = \texttt{dId}(m_2) \quad \{\{m_2.\texttt{dId} \rightsquigarrow \texttt{dId}(m_1)\ \mathsf{False}\ \}\} \quad \wedge \\
& \quad\quad \texttt{kind}(m_1) = \texttt{kind}(m_2) \quad \{\{m_2.\texttt{kind} \rightsquigarrow \texttt{kind}(m_1)\ \mathsf{False}\ \}\}
\end{array}
$$

Figure 2.11: Formal example rules with hints

The sorted repair collection below shows how the repairs in Fig. 2.10 should be best combined, in order to resolve all inconsistencies at state 4.

1.) $\{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}$,
2.) $\{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\}$,
3.) $\{rep_{\mathsf{Chgman1}}, rep_{\mathsf{DelkaA2}}\}$,
4.) $\{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Deldoc2}}\}$

The collection above implies a preference between the repair sets. In our example, we change a document rather than deleting it. Changing a text document is preferred to changing a key definition within a key resolver. In addition, we prefer deleting some content within a document to deleting a whole document.

There are some critical issues to consider when generating repair collections: Is every repair set a new alternative? Within a repair set, can all repairs be applied altogether? Are repairs compatible with the document structure? Throughout, we have assumed that we want to repair all inconsistencies at once; under heavy pressure of time and cost, however, we might want to repair the most troubling inconsistencies only. We tackle these and other challenges in Chapter 9.

2.4.3 Hints

The reader may ask how we could generate the above S-DAGs and repair collections as we support *arbitrary* function symbols and predicate symbols within rules. In order to generate meaningful repairs, we need to incorporate domain knowledge into repair generation. Users may annotate rules by *hints*, which indicate how truth values of atomic formulae can be inverted. Hints also serve to prevent the generation of repair actions that are unwanted in specific situations. In Fig. 2.11, we have annotated our rules by hints. In rule ϕ_1, consider the annotation for the atomic formula $k = \texttt{key}(d)$:

$$
\left\{ \begin{array}{l} \{k \rightsquigarrow \texttt{key}(d)\ \mathsf{False}\ \}, \\ \{d.\texttt{key} \rightsquigarrow k\ \mathsf{False}\ \} \end{array} \right\}
$$

It proposes changes to one of the variables k or d, in order to invert the truth value of $k = \texttt{key}(d)$, if its boolean result is False. We either set the key k to $\texttt{key}(d)$ or set the key of the key definition d to k. Clearly, then k and $\texttt{key}(d)$ are equal. The hint for the atomic formula $\texttt{dId}(m) = \texttt{kId}(d)$ sets the field dId of the manual m only; we would never change the key definition d in case $\texttt{dId}(m) = \texttt{kId}(d)$ is violated.

Annotations for rule ϕ_2 follow the above shape. We have annotated some variables by KEEP, in order to prevent generation of repair actions for them. This is useful, because, in general, our algorithm generates repair actions even for atomic formulae not annotated by hints, e.g., $t_1 < t_2$ in rule ϕ_2. Hints are the basis for high-level repairs. For example, we can react differently to the same inconsistency depending on the development state of a document engineering project.

We discuss hints in detail in Chapter 8 — they are an important prerequisite to deriving meaningful S-DAGs.

Part I

Consistency Checking: Finding Inconsistencies

In the first part of this thesis, we examine how we can find inconsistencies and show them precisely to users. In Chapter 3, we show how we integrate our approach to consistency checking into the every-day work with a DMS. That way we provide users with a tool to check consistency that does not require adaptations to their document engineering practices. In Chapter 4, we show how consistency rules are formalized. We also detail our static type checking algorithm, which ensures well-typedness of rules. In Chapter 5, we define our basic algorithm to generate consistency reports. A report shows precisely to users *when*, *where*, and *why* a consistency rule is violated. Finally, in Chapter 6, we develop methods to speed up consistency checking, which is of major importance, because the repository is locked during a consistency check.

Chapter 3
Consistency-Aware Document Management Systems

In this chapter, we describe how we integrate consistency checking into mainstream DMSs. Fig. 3.1 illustrates the context of this chapter. We do not develop a stand-alone consistency checker but extend a DMS, because DMSs are widely used for document engineering and provide a useful basis for managing, e.g., different document versions, author's access rights, and the document engineering process itself. In addition, a DMS provides the basis for formalizing temporal consistency rules and efficient consistency checking. In this thesis, we follow a conservative approach: Neither do we require end users to change their habits in working with a DMS, nor do we expect special services from a DMS. We design our consistency checker in a way that makes only few assumptions about the DMS. We require only

- access to each version of a document that has been checked in,
- a facility (plug-in mechanism) to call our consistency checker at a check-in,
- a locking mechanism, which prevents check-ins during a consistency check, and
- information about which documents have been modified by a check-in (this is important for efficient consistency checking).

In particular, we make no assumptions about the document model of the DMS, such that our extensions also apply to revision control systems like CVS [$C^{+}02$], subversion [CSFP04], or DARCS [Rou04].[1]

Formalizing consistency rules causes complexity in various problem areas. Therefore, we divide formalization into different tasks and supply tools to every stake-holder (see Fig. 3.2).

[1]Currently, we do not support branching in repositories, which would require an extension to branching time logic.

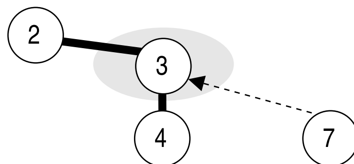


Figure 3.1: Chapter 3 in context

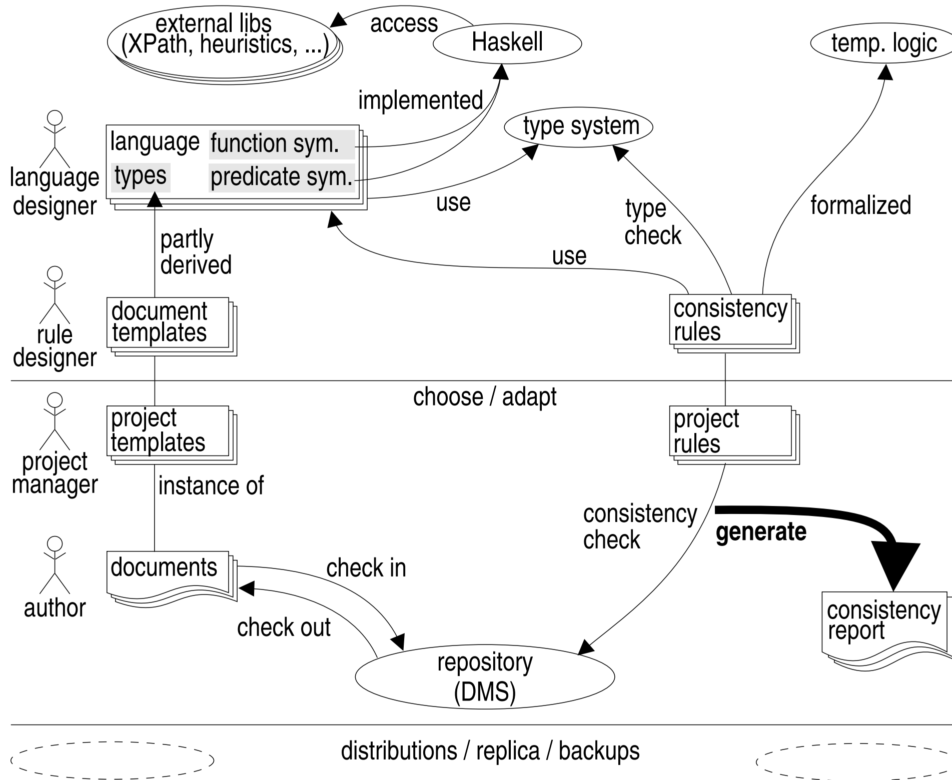


Figure 3.2: Overview of a consistency-aware DMS (ovals mark fixed components; rectangles mark customizable components)

From a repository, *authors* typically check out working copies of documents, modify them, and check them in again. From the repository perspective, authors add, modify, or delete documents. Among other things, a classic DMS manages concurrent check-ins, author's permissions, version control, repository backup, and repository distribution. From the author perspective, a consistency-aware DMS behaves like a traditional DMS, with the only exception that the former provides authors with consistency reports.

The *rule designer* plays an important rôle: He formalizes consistency rules. Rules define what consistency means — they reflect wishes from an "administrator" perspective. We distinguish between *strong* rules, which must be adhered to, and *weak* rules, which may be violated. The repository is checked for consistency at given events, e.g., document check-in. For each rule, our consistency checker generates a *consistency report*, to which we can react in various ways. On violation of a weak rule, the system could inform those authors who have checked out (now) inconsistent documents. If, however, a strong rule is violated, the system rejects the check-in in question. In addition, the rule designer creates templates for documents. These will be DTDs or Schemas, if XML is used as document format [W3C01, W3C04].

Consistency rules use function symbols and predicate symbols from domain-specific languages. This makes the rules independent of concrete document

formats, which can be changed without affecting rules. A static type system ensures that rules are well-typed w.r.t. the language used. For example, the predicate symbol $=$ requires two arguments of the same type. The careless application of $=$ to a number and a document must be rejected, because it is meaningless. Thus, without accessing repository data, our static type system decides on the syntax level whether a consistency rule can possibly make sense. For convenience, we provide a basic language Prelude, which defines fundamental function symbols, predicate symbols, and types.

In user-defined languages, a *language designer* declares domain-specific function symbols, predicate symbols, and types. The semantics of function and predicate symbols is defined in Haskell [PJ03] — a statically typed purely functional programming language.[2] As described in Chapter 6, Haskell's referential transparency supports efficient consistency checking.[3] Also, Haskell provides access to other libraries via a foreign function interface [C$^+$03]. Such libraries offer sophisticated functionality, e.g., for parsing documents or heuristics for semantic content analysis [Sal88, WS99, MPG01, FGLM02, BHQW02, BQBW03, AL03a]. Formal document types (usually corresponding to document templates) can be expressed by record or variant types. This makes our approach independent of any particular document format and facilitates heterogeneous repositories. If XML is used as document format, document types and parser functions can be derived from DTDs [WR99]. Notice that complex programming tasks are hidden from the rule designer who only needs temporal logic.

For specific projects, a *project manager* chooses consistency rules. In some cases adaptations will be necessary, e.g., the document templates use another company sign or some rules are weakened. Of course, major changes to the document templates cause adaptations to the corresponding formal document types, which may involve further changes in the language. But in our experience most projects require layout related adaptations only.

In the next chapter, we focus on the work of the rule designer (formalizing rules) and the language designer (defining languages).

[2] For example, the language designer would declare $=$ to have the polymorphic type $\forall\alpha.\alpha\times\alpha \rightarrow \mathsf{Bool}$ (α denotes a type variable, $\times$ separates argument types, Bool denotes the type for boolean values). A Haskell function then defines the meaning of $=$, e.g., via instances of the type class `Eq`.

[3] The use of Haskell is not obligatory; our approach depends on referential transparency only.

Chapter 4

Formalizing Consistency Rules

In this chapter, we formalize consistency rules by using first-order temporal logic. Our experiments have shown that we need expressive means to formalize consistency rules. We require a temporal component as well as complex functions and predicates. Also, rule design should be comfortable: Once defined, we want to re-use functions and predicates. This is why we have decided to employ *full* first-order temporal predicate logic.[1] We support polymorphic (even higher-order) functions and predicates, which facilitate comfort and reuse. The challenge is, therefore, to smoothly combine a first-order logic with higher-order functions and predicates. We achieve this by our type checker.

We start this chapter with an informal overview in Sect. 4.1. Sect. 4.2 introduces the abstract syntax of consistency rules. In Sect. 4.3, we tackle the problem of "writing proper consistency rules;" we define typing rules and a type checking algorithm. Type checking might appear a rather complex matter; it is, however, essential for a sound semantics for our rules (this is similar to the rôle type checking plays for programming languages). Notice that rule designers and language designers are not affected by the complex formal issues in Sect. 4.3; they just enjoy the features. Sect. 4.3 is not essential for the further understanding of this thesis, but it is necessary to ensure the soundness of our approach.[2] Finally, we summarize this chapter in Sect. 4.4. Fig. 4.1 illustrates the context of this chapter. In the next chapter, we detail how consistency rules are evaluated.

[1] We employ a first-order logic only, because, based on our experiments, we have not seen the need for higher-order logic (see Part III for a case study).

[2] Type checking is, however, important for the type checking of hints, discussed in Sect. 8.2.2.

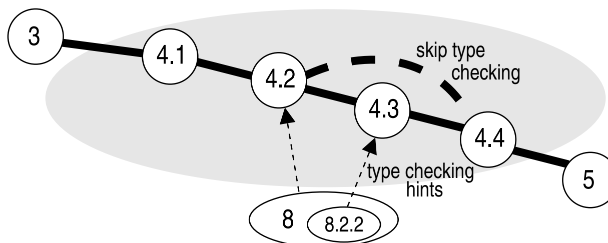
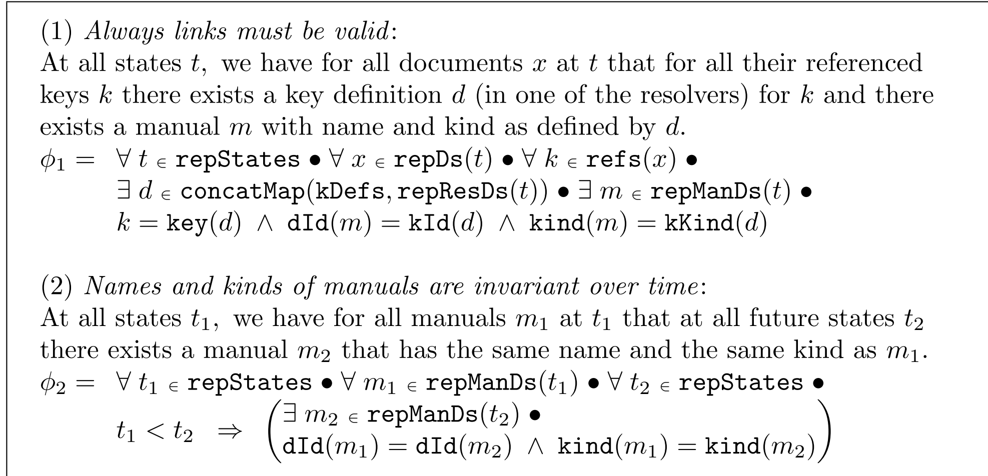


Figure 4.1: Chapter 4 in context


(1) *Always links must be valid*:
At all states t, we have for all documents x at t that for all their referenced keys k there exists a key definition d (in one of the resolvers) for k and there exists a manual m with name and kind as defined by d.

$$\phi_1 = \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \exists\, m \in \texttt{repManDs}(t) \bullet k = \texttt{key}(d) \wedge \texttt{dId}(m) = \texttt{kId}(d) \wedge \texttt{kind}(m) = \texttt{kKind}(d)$$

(2) *Names and kinds of manuals are invariant over time*:
At all states t_1, we have for all manuals m_1 at t_1 that at all future states t_2 there exists a manual m_2 that has the same name and the same kind as m_1.

$$\phi_2 = \forall\, t_1 \in \texttt{repStates} \bullet \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \forall\, t_2 \in \texttt{repStates} \bullet t_1 < t_2 \Rightarrow \left(\begin{array}{l}\exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{dId}(m_1) = \texttt{dId}(m_2) \wedge \texttt{kind}(m_1) = \texttt{kind}(m_2)\end{array}\right)$$

Figure 4.2: Formal example rules

4.1 Informal Overview

Consistency rules may quantify over repository states,[3] which we also call timestamps, because they represent given points in time. Our abstract syntax consists of two parts: (1) the rule designer expresses consistency rules in a first-order temporal logic; (2) the language designer declares function symbols and predicate symbols (that can be used in rules) and implements their semantics in Haskell. Throughout, we use the term “symbols” to refer to both predicate symbols and function symbols.

Rule designers formalize consistency rules in a variant of the two-sorted temporal first-order predicate logic with linear time and equality [Eme90, AHV95, AHV96]. The two-sorts approach to temporal logic introduces a new temporal sort Time. To each non-temporal symbol a timestamp is added. Fully temporal symbols have temporal arguments (and results) only. Quantifiers iterate over variables of the sort Time, too. We use a variant of the two-sorts approach, because (1) timestamp variables make temporal logic more expressive [HWZ00], (2) rule designers do not need to learn temporal connectives, and (3) the introduction of types makes the two-sorts approach straightforward.

We model time by discrete repository states, expressed through natural numbers for simplicity.[4] The type State is interpreted as repository state. To simplify notation, type checking, and our semantics we omit the timestamp parameter of symbols that do not depend on time. A distinction between fully temporal symbols and partially temporal symbols is not necessary in our setting.

The rule designer defines our example consistency rules as shown in Fig. 4.2. Binary predicate symbols are written in infix notation for better readability.

[3]A *state* represents a repository snapshot at a given point in time. A check-in causes a state transition. State transitions in the induced state automaton connect states in ascending order.

[4]We do not model real time. Some DMSs can map real time to repository states.

In rule ϕ_1, we first quantify over all repository states, provided by `repStates`. Then, for each state t we need the current documents x, obtained by $\texttt{repDs}(t)$, and the referenced keys k therein, computed via $\texttt{refs}(x)$. For every referenced key k, there must exist a key definition d, such that the key defined by d (computed via $\texttt{key}(d)$) equals k. In addition, there must exist a manual m whose name equals the identifier mentioned by d and whose kind equals the kind d points to (we obtain the current manuals by $\texttt{repManDs}(t)$). The current key definitions d are computed via $\texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t))$, which gets the current key resolver documents ($\texttt{repResDs}(t)$) and applies `kDefs` to each resolver. Applied to a resolver document, `kDefs` returns a list of key definitions. Finally, all lists are concatenated. Essentially, `concatMap` behaves like a universal quantifier here.[5] Since quantified variables provide pointers to inconsistencies, rule designers must be careful replacing quantifiers by using `concatMap`.

In rule ϕ_2, we twice quantify over repository states, because we have to relate different versions of manuals: the old version at state t_1 and the new version at state t_2.

For the formalized rules, the language designer defines the language Survey, which contains symbols and types as shown in Fig. 4.3. Survey imports the predefined language Prelude, the relevant part of which is also shown in Fig. 4.3. For simplicity, our formalism neglects import relations between languages. We let $<_\mathbf{S}$ denote an explicit subtype relation between record types; a record type may have multiple supertypes.

The variant list type $[\alpha]$ is declared as usual in functional programming [MTH90, PJ03]; the variant constructors are [] (empty list) and (:) (add an element to the front of a list). The record type `Doc` stands for a formal document, carrying a name (of type `String`) and a check-in state (of type `State`). That way we distinguish different document versions. We require that each document type is a subtype of `Doc`. This is important for efficient consistency checking. The record type `ResD` resembles the key resolver structure (see Fig. 2.2 on pg. 10). `ResD` inherits all record labels from its supertype `Doc`. Our notion of subtyping resembles XML Schema subtyping via extension and restriction [W3C01]. Record and variant type definitions induce new function symbols (labels and constructors, respectively). For example, `ResD` induces $\texttt{kDefs} : \texttt{ResD} \to [\texttt{Item}]$.

We regard predicate symbols as function symbols with a boolean result type, denoted by Bool. Function symbols starting with `rep` provide access to documents within the repository at a given state. Notice that these functions return *lists* of documents — in formulae we quantify over lists, instead of sets, because we expect lists to be more familiar to (beginner) Haskell programmers (our language designers) than sets. For a document d, $\texttt{refs}(d)$ returns all referenced keys. $\texttt{concatMap}(f, xs)$ applies the function f to each member of the

[5] `concatMap` is a higher-order symbol, i.e., it takes a function as argument. We apply some restrictions to function types that ensure that our logic remains first-order.

Extract from language *Prelude* (predefined)

Type definitions

`String`			strings
$[\alpha]$	$=$	$[] \mid (:)\ \alpha \times [\alpha]$	Haskell like lists
`Doc`	$=$	`Doc` $\{$`dId` : `String`, `dState` : `State`$\}$	documents

Predicate symbol definitions

$=$	:	$\forall \alpha . \alpha \times \alpha \rightarrow \mathsf{Bool}$	equality
$\leq$	:	$\forall \alpha . \alpha \times \alpha \rightarrow \mathsf{Bool}$	less than or equal
$<$	:	$\forall \alpha . \alpha \times \alpha \rightarrow \mathsf{Bool}$	less than

Function symbol definitions

`concatMap`	:	$\forall \alpha, \beta . (\alpha \rightarrow [\beta]) \times [\alpha] \rightarrow [\beta]$	Haskell like `concatMap`
`repStates`	:	$[\mathsf{State}]$	all repository states

Language *Survey*, defined by the language designer (imports Prelude)

Type definitions

`ManD` $<_{\mathsf{S}}$ $\{$`Doc`$\}$	$=$	`Man` $\{$`kind` : `String`$\}$	manual documents
`ResD` $<_{\mathsf{S}}$ $\{$`Doc`$\}$	$=$	`Res` $\{$`kDefs` : [`KDef`]$\}$	key resolver documents
`KDef`	$=$	`KDef` $\{$`key` : `String`, `kId` : `String`, `kKind` : `String`$\}$	key definitions

Function symbol definitions

`repDs`	:	State $\rightarrow$ [`Doc`]	get all documents in the repository
`repManDs`	:	State $\rightarrow$ [`ManD`]	get all manual documents
`repResDs`	:	State $\rightarrow$ [`ResD`]	get all resolver documents
`refs`	:	`Doc` $\rightarrow$ [`String`]	keys referenced in a document

Figure 4.3: Example types and symbols (the operator : separates a symbol from its type, $\rightarrow$ separates the argument types of a function type or predicate type from the result type, $\times$ separates argument types)

list *xs* and concatenates the result lists. Record labels can serve as parameter for `concatMap`, e.g., in `concatMap`(`kDefs`, `repResDs`(t)).

In addition, the language designer defines the semantics of the symbols in our new language Survey. Currently, symbol semantics is defined in Haskell. Since this programming task depends on the current implementation of our prototype, we shall neglect this matter for the moment. We will discuss the definition of functions and predicates in App. B, which outlines our current prototype implementation.

4.2 Abstract Syntax

In this section, we describe the formal abstract syntax of consistency rules. Rule designers and language designers define rules and languages via some concrete syntax, which is converted to the abstract syntax described in this section. The formal abstract syntax is a fundamental prerequisite for *typing* consistency rules, which we describe in the next section, and *evaluating* consistency rules, which we describe in Chapter 5.

Syntax	Description
Formulae $\mathcal{F}$	
$\phi, \psi ::= p(e_1, \ldots, e_n) \mid \neg\phi$	atomic formula $\mathcal{F}_{\text{at}}$; negation
$\mid \; \phi \vee \psi \mid \phi \wedge \psi \mid \phi \Rightarrow \psi$	disjunction; conjunction; implication
$\mid \; \exists\, x \in e \bullet \phi$	existentially quantified formula
$\mid \; \forall\, x \in e \bullet \phi$	universally quantified formula
Terms $\mathcal{T}$	
$e \quad ::= x \mid s$	variable $\mathcal{X}$; symbol $\mathcal{S}$ $(s \in \{f, l, k\})$
$\mid \; s(e_1, \ldots, e_n)$	symbol application
$\mid \; K_R \; \{\overline{l_i = e_i}\}$	record construction
$\mid \; \mathsf{case}(e, \{\overline{k_i \to s_i}\})$	variant deconstruction
$\mid \; \mathsf{case}(e, V, \{\overline{k_i \to s_i}\})$	variant deconstruction for variant constructor V
$\mid \; e :: \tau$	type annotation with type τ

Figure 4.4: Abstract syntax of formulae $\mathcal{F}$ and terms $\mathcal{T}$ (types τ are defined in Fig. 4.6 on pg. 32)

Fig. 4.4 summarizes the abstract syntax of consistency rules — the rule designer's tool set. More often than not, consistency rules are standard first-order formulae.[6] A quantifier ($\forall$ or $\exists$) introduces a bound variable x, restricted by a term e that evaluates to the sphere of x. We use terms for quantifier restrictions, in order to easily identify variable spheres.

Restricting quantifier spheres is known from many applications of logic, e.g., via access paths in Access Limited Logic [CK91] or formulae in applications of temporal logic for planning [BK00]. The consistency management toolkit xlinkit [NCEF02] uses XPath statements for restricting quantifiers. We generalize this approach towards terms. We argue that explicit notation of quantifier spheres by terms[7] has several advantages in our setting:

- Rule designers can easily identify variable spheres in a formula at the point where variables are introduced and do not need to "search" for implicit bounds.
- Explicit quantifier spheres accelerate consistency checking.
- Restriction of quantifier spheres via terms is already used in XPath [W3C99b, W3C03]; therefore, our rule syntax should appear natural to rule designers.

Without restriction to generality, we only permit formulae in which each quantifier binds a different variable. Such formulae are commonly called *rectified.* In our sense, closed rectified formulae are *consistency rules.* We call a non-empty finite set of consistency rules a *rule system.* Notice that a consistency rule

[6] Similar to classic predicate logic, we could replace binary logical connectives and negation by implication only. For better comprehension, we choose not to do so.

[7] Our abstract syntax provides no means to quantify over the complete domain of a type. This has to be accomplished by a special function symbol, which of course may be derived automatically. We regard this feature, however, as syntactic sugar, which usually is neglected in abstract syntax.

corresponds to a sentence and a rule system corresponds to a theory in classic logic. Typically, a rule contains additional metadata, e.g., whether it may be violated and its priority. We let $\mathcal{F}$ denote the set of all formulae, $\mathcal{F}_{\text{at}} \subset \mathcal{F}$ is the set of all atomic formulae. For easy lookup, App. A lists references to the definitions of all notations.

In the logic used here, it is *undecidable* whether a formula is satisfiable. This is, however, no issue in our setting, because rules are evaluated against concrete repositories, i.e., finite structures. What we are interested in are concrete inconsistency pointers generated from complex consistency rules. Therefore, we sacrifice decidability for expressivity. For some applications, it might be interesting to know whether the rules are satisfiable or whether some rules are implied by others. Both questions reduce to the implication problem, which has been proven undecidable in full first-order predicate logic (see, e.g., [Men87]).[8] Thus, we only perform analyses that detect some cases of rule contradiction and implication, but not all of them (see Sect. 6.2.2).

We define terms similar to Nordlander's Haskell extension O'Haskell [Nor98, Nor99, Nor02], which extends the Haskell type system with *subtypes*. Thus, we regard function symbols f, record labels l, and variant constructors k as terms, too. In contrast to O'Haskell, we let an explicit record constructor K_R construct a record of type R, in order to guide type inference; K_R takes a set of label bindings $\{\overline{l_i = e_i}\}$ as argument. In a case statement, the case scrutinee e must have a variant type. The second argument of a case statement is a set of bindings $\{\overline{k_i \to s_i}\}$ that associate each of e's variant constructors k_i with a symbol s_i: If e is constructed by k_i, then s_i is applied to the arguments of k_i.[9] An optional variant type constructor V uniquely identifies the type of e, thus avoiding ambiguities that may arise from subtyping. Any expression can be annotated with a type, in order to guide type inference. We denote the set of all terms by $\mathcal{T}$, where $\mathcal{X} \subset \mathcal{T}$ is the set of all variables and $\mathcal{S} \subset \mathcal{T}$ is the set of all symbols.

Next, we discuss type checking of consistency rules, which is a fundamental prerequisite for the soundness of consistency checking. Type checking might appear a rather complex matter. It is, however, not essential for the further understanding of this thesis. Thus, readers disinterested in these technical details might want to skip the following section and to continue with the chapter summary (Sect. 4.4, pg. 44).

4.3 Writing Proper Consistency Rules

The motivation behind type checking consistency rules is that we can give a reasonable semantics for well-typed terms and formulae only. Since we permit language designers to define complex domain-specific functions and predicates,

[8] There exist decidable subsets in temporal predicate logic [HWZ00]. A restriction to these subsets would, however, severely limit the range of applications using our approach.

[9] We associate variant constructors with symbols rather than with expressions for technical (and implementation) reasons. Since we prohibit rule designers to define new symbols (e.g., via let or λ expressions), this might require to define new symbols in a language.

the rule designer's task of writing meaningful consistency rules becomes more complicated and must be supported. Our example rules in Fig. 4.2 are well-typed. If, however, we erroneously omit the function symbol `key` within $k = \mathtt{key}(d)$ then a type checker ought to warn us that this atomic formula is ill-typed. The type checker we have developed supports basic subtyping and higher-order functions, and guarantees first-order properties of formulae.

Our combined type checking and type inference algorithm assigns a monomorphic type to each term. Monomorphism is important since it facilitates to treat types like primitive sorts, which can be omitted. In conjunction with some restrictions we put on function types, this retains compatibility with traditional first-order logic and supports simple set theoretic semantics. As usual, we separate well-typedness rules (Sect. 4.3.2) from the type inference algorithm (Sect. 4.3.3). Both are extensions to [Nor99], the major adaptations being:

- We extend the algorithm to the typing of formulae; a quantified formula roughly corresponds to a `let` construct in a programming language.
- We require that each term has a monomorphic type, in order to support simple set theoretic semantics.
- We prohibit partial application.

Before we discuss type checking in detail, we introduce some formal means necessary.

4.3.1 Preliminaries: Signature, Types, Type Declarations, Environments

In this section, we define the formal notion of *signature*, which carries predicate symbols, function symbols, and types. Our system derives a signature from the languages defined by the language designer. From the user perspective, a signature appears unnecessary. It includes, however, information about symbols and types that are essential for type checking. In addition, we define our notion of *type* formally and outline user-defined subtyping. The following subsections contain our definitions in a top down manner.

Signature

Within a signature, the type constructor set $\mathbf{T}$ carries all type constructors defined in the corresponding languages; $\mathbf{T}$ also contains document type constructors, modeling document templates. The type structure $\Omega(\mathbf{T})$ contains all types properly constructed from the type constructors in $\mathbf{T}$ and type variables; usually, $\Omega(\mathbf{T})$ is infinite. The subtype theory $\mathbf{S}$ contains subtype axioms, which define subtype relationships between record and variant types, respectively. $\mathbf{S}$ is induced by user-defined subtype relationships, which we will explain below. A record type constructor definition R induces a record constructor K_R and a label environment $\prod_{\widehat{R}}$ containing record labels and their

$$
\begin{array}{lll}
\mathbf{T} & = \{[_], \mathtt{Doc}, \mathtt{String}, \mathsf{State}\} & \\
\mathbf{S} & = \emptyset & \\
\prod & = \prod_{\widehat{\mathtt{Doc}}} & \\
\prod_{\widehat{\mathtt{Doc}}} & = \{\mathtt{dId} : \mathtt{Doc} \to \mathtt{String}, \mathtt{dState} : \mathtt{Doc} \to \mathsf{State}\} & \\
\sum & = \sum_{\widehat{[_]}} & \\
\sum_{\widehat{[_]}} & = \{[] : [\alpha], (:) : \alpha \times [\alpha] \to [\alpha]\} & \\
\mathbf{P} & = \mathbf{P}_{\alpha\times\alpha\to\mathsf{Bool}} \cup \mathbf{P}_{[\alpha]\to\mathsf{Bool}} & \\
\mathbf{P}_{\alpha\times\alpha\to\mathsf{Bool}} & = \{=, \leq\} & \\
\mathbf{P}_{[\alpha]\to\mathsf{Bool}} & = \{\mathtt{null}\} & \text{is a list empty?} \\
\mathbf{F} & = \mathbf{F}_{[\mathsf{State}]} \cup \mathbf{F}_{\mathsf{State}} \cup \mathbf{F}_{\mathsf{State}\to\mathsf{State}} \cup \mathbf{F}_{\mathsf{State}\to[\mathsf{Doc}]} \cup \mathbf{F}_{\mathtt{Doc}\to\mathtt{String}} & \\
\mathbf{F}_{[\mathsf{State}]} & = \{\mathtt{repStates}\} & \text{all valid states in the repository} \\
\mathbf{F}_{\mathsf{State}} & = \{\mathtt{repHead}, \mathtt{repInit}\} & \text{last (first) valid state in the repository} \\
\mathbf{F}_{\mathsf{State}\to\mathsf{State}} & = \{\mathtt{next}, \mathtt{prev}\} & \text{next (previous) repository state} \\
\mathbf{F}_{\mathsf{State}\to[\mathsf{Doc}]} & = \{\mathtt{repDocs}\} & \text{all documents at a given repository state} \\
\mathbf{F}_{\mathtt{Doc}\to\mathtt{String}} & = \{\mathtt{docContent}\} & \text{content of a document}
\end{array}
$$

Figure 4.5: Base signature

types; this also includes the labels from the supertypes. For example, the label environment $\prod_{\widehat{\mathtt{ResD}}}$ contains $\mathtt{kDefs} : \mathtt{ResD} \to [\mathtt{KDef}]$, $\mathtt{dId} : \mathtt{ResD} \to \mathtt{String}$, and $\mathtt{dState} : \mathtt{ResD} \to \mathsf{State}$. A variant type constructor definition V induces a constructor environment $\sum_{\widehat{V}}$ containing variant constructors; this also includes constructors from subtypes. For example, the constructor environment $\sum_{\widehat{[_]}}$ contains $[] : [\alpha]$ and $(:) : \alpha\times[\alpha] \to [\alpha]$. We will define environments shortly. The set $\mathbf{P}$ carries predicate symbols; $\mathbf{F}$ carries function symbols. As usual in many sorted logics, we partition symbol sets w.r.t. the type of the symbols they include. This is necessary for our semantics.

Definition 4.1 (Signature) A *signature* $\Sigma = (\mathbf{T}, \mathbf{S}, \prod, \mathbf{K}, \sum, \mathbf{P}, \mathbf{F})$ contains:

- a type constructor set $\mathbf{T}$;
- a subtype theory $\mathbf{S}$ holding subtype axioms over $\Omega(\mathbf{T})$;
  - $\mathbf{S}$ is a set of axioms $\tau_1 <_{\mathbf{S}} \tau_2$, where either
  - both τ_1 and τ_2 are record types, or
  - both τ_1 and τ_2 are variant types.
- a record type label environment $\prod_{\widehat{R}} \in \prod$ and a record constructor $K_R \in \mathbf{K}$ for each record type constructor $R \in \mathbf{T}$;
- a variant type constructor environment $\sum_{\widehat{V}} \in \sum$ for each variant type constructor $V \in \mathbf{T}$;
- predicate symbols $p \in \mathbf{P}$; where $\mathbf{P} = \bigcup \mathbf{P}_{\tau_1\times\ldots\times\tau_n\to\mathsf{Bool}} (\tau_i \in \Omega(\mathbf{T}))$ contains predicate symbols of type $\tau_1\times\ldots\times\tau_n \to \mathsf{Bool}$;
- function symbols $f \in \mathbf{F}$; where $\mathbf{F} = \bigcup \mathbf{F}_{\tau_1\times\ldots\times\tau_n\to\tau_g}$ $(\tau_i, \tau_g \in \Omega(\mathbf{T}))$ contains function symbols of type $\tau_1\times\ldots\times\tau_n \to \tau_g$. □

Each signature includes types and symbols shown in Fig. 4.5, i.e., it is an extension of the *base signature*. In Sect. 5.5, we assign a fixed meaning to

T	$::= A_0 \mid R_n \mid V_n$	type constructors (atomic; record; variant)
τ_g, ρ_g	$::= \alpha \mid \beta$	ground type variable
	$\mid\ T_n\ \tau_{g_1} \ldots \tau_{g_n}$	constructed type
	$\mid\ \mathsf{State} \mid \mathsf{Top}$	repository state; supertype of all types
	$\mid\ \mathsf{Bool}$	boolean type
τ, ρ	$::= \tau_g \mid \nu$	ground type; type variable
	$\mid\ \tau_1 \times \ldots \times \tau_n \to \tau_g$	function type
τ_p	$::= \tau_1 \times \ldots \times \tau_n \to \mathsf{Bool}$	predicate type
D, C	$::= \{\overline{\tau_i \leq \rho_i}\}$	subtype constraints
σ	$::= \forall \bar{\alpha}.(\tau \mid C)$	function type scheme
σ_p	$::= \forall \bar{\alpha}.(\tau_p \mid C)$	predicate type scheme

Figure 4.6: Abstract syntax of types

the symbols in **P** and **F**, respectively. The function symbols are necessary for accessing the repository. For the type State, the predicate symbol $\leq$ is interpreted by the natural linear ordering relation on natural numbers. That way we support linear time.

Types

We now introduce the formal notion of *types*. In the following sections, we describe how type constructors in the set **T** are defined, and how constructor environments $\prod_{\widehat{R}}$ and label environments $\sum_{\widehat{R}}$ are built. We need these environments for type checking. Our static type system is based on functional programming languages, such as Haskell [PJ03] or ML [MTH90]. Since many ideas carry over from [Nor99], we concentrate on important aspects and show our adaptations only. Our type system slightly differs from ML and Haskell, particularly, because we prohibit overloading and partial application.[10]

Fig. 4.6 summarizes our type language. Type constructors $T \in \mathbf{T}$ have a fixed arity. Atomic type constructors A introduce atomic types (like `String`); we identify nullary type constructors by the types they represent. Record type constructors R build record types, variant type constructors V introduce variant types — see below for a detailed description. A ground type may be either a ground type variable, or it may be constructed by a type constructor. The type State models discrete repository states. We distinguish a special type Top, which is supertype of all types.

We apply some restrictions, in order to ensure that (higher-order) function symbols result in ground types τ_g only and thus the type system supports first-order logic properties: We do not quantify over functions. In contrast to many functional programming languages, we restrict ourselves to full application of

[10] Haskell permits to apply the binary function `(+)` to one argument only: `(+) 0.0 :: Float -> Float`. In our type system, however, `(+)` requires exactly two arguments: `(+Float) (0.0,1.0) :: Float`. In addition, we do not support ad-hoc polymorphism, which is achieved via type classes in Haskell. Therefore, we annotate `(+)` with its result type — here `Float`.

a type constructor to argument types[11] and full application of symbols to argument terms. To simplify notation, we use uncurried syntax in function and predicate types. We distinguish between ground types τ_g and general types τ, in order to ensure that the result type of a function symbol is not a function type. Notice the difference between ground type variables α and general type variables ν. General type variables ν are reserved for type inference only; ground type variables can be used by the language designer to define polymorphic symbols. This restriction is necessary to guarantee that polymorphic functions symbols have a ground type as result type. As usual, function and predicate type schemes (σ and σ_p) are universally quantified over all their type variables. A subtype constraint $\tau \leq \rho$ requires that the type τ is a subtype of the type ρ. We use the shorthand notation $\{\overline{\tau_i \leq \rho_i}\}$ to denote a set of subtype constraints.

Type Declarations

Records and variants are well known from programming languages and are (in conjunction with user declared subtyping) a key ingredient to make our approach to typed formal consistency rules viable. In this section, we shortly review the abstract syntax of type declarations.

A *record type* is defined via some concrete syntax, which we abstract as follows ($\mathsf{tv}(\tau)$ denotes the free type variables of the type τ, $\mathsf{tv}(\bar{\tau})$ denotes the free type variables of a family of types $\bar{\tau}$):

$$R_n\ \bar{\alpha} <_{\mathsf{S}} \{\overline{S_i\ \bar{\rho}_{gi}}\} \ = \ K_R\ \{\overline{l_j : \tau_{g_j}}\} \qquad \text{where } \bigcup \mathsf{tv}(\overline{\bar{\rho}_{gi}}) \cup \bigcup \mathsf{tv}(\overline{\tau_{g_j}}) \subseteq \bar{\alpha}.$$

The above declaration introduces the n-ary record type constructor R_n, parameterized over type variables $\bar{\alpha}$; only these type variables can be used within the record type declaration. The type $R\ \bar{\alpha}$ is a subtype of each record type $S_i\ \bar{\rho}_{gi}$ — we support multiple inheritance. Each record label l_j has a ground type τ_{g_j} and can be used as a regular function symbol, which selects a component from the record. The so declared record type inherits all labels from its supertypes. If new labels are defined, we say that the declared record type is a specialization of its supertypes. If no new labels are defined and $R\ \bar{\alpha}$ has only one supertype, $R\ \bar{\alpha}$ is considered isomorphic to its supertype. The record constructor K_R, applied to a set of label bindings, constructs a record term. In contrast to O'Haskell, we require an explicit constructor for a record, in order to prevent ambiguity that may occur if we define record subtypes without adding new labels. This situation may occur frequently when document types are defined. Also, explicit record constructors are more in the spirit of traditional Haskell requiring explicit record constructors, too.

Our abstract syntax of *variant type* declarations closely follows the O'Haskell syntax for data type declarations:

$$V_n\ \bar{\alpha} >_{\mathsf{S}} \{\overline{S_i\ \bar{\rho}_{gi}}\} \ = \ \overline{k_j : \bar{\tau}_{gj}} \qquad \text{where } \bigcup \mathsf{tv}(\overline{\bar{\rho}_{gi}}) \cup \bigcup \mathsf{tv}(\overline{\bar{\tau}_{gj}}) \subseteq \bar{\alpha}.$$

[11] Partial type constructor application would require a kind system similar to that of Haskell.

The above declaration introduces the n-ary variant type constructor V_n, parameterized over type variables $\bar{\alpha}$; again, only these variables can be used within the declaration. The type $V\ \bar{\alpha}$ is a supertype of each variant type $S_i\ \bar{\rho}_{gi}$. A variant constructor k_j builds a term of the type $V\ \bar{\alpha}$ and can be used like a regular function symbol. The subtype/supertype relation for variant type declarations is "reversed," compared to record type declarations. Since the declared variant type inherits all constructors from its subtypes, it is a superset of all its subtypes. Variants can be deconstructed by one of the `case` statements.

Record labels, record constructors, and variant constructors share the name space of all symbols and must be globally unique. Notice, that we only support subtype relationships between record types and variant types, respectively. We prohibit subtype relationships involving atomic types, in order to simplify our semantics.

Label Environments and Constructor Environments

We now outline how record label environments, variant constructor environments, and function symbols are induced by record type declarations and variant type declarations, respectively.

Consider a record type declaration $R\ \bar{\alpha} <_{\mathbf{S}} \{\overline{S_i\ \bar{\rho}_{gi}}\} = K_R\ \{\overline{l_j : \tau_{g_j}}\}$, where $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, i.e., the type $R\ \bar{\alpha}$ has n supertypes and m new labels. Then we define the following:

1. A type scheme for each label, such that $l_j : \forall \bar{\alpha}. R\ \bar{\alpha} \to \tau_{g_j}$ can be used as a regular function symbol (with the built-in semantics to select a component from a record value).

2. A set $\mathbf{S}_R \subset \mathbf{S}$ of transitively closed subtype axioms:

$$\mathbf{S}_R = \left\{ \begin{matrix} R\ \bar{\alpha} <_{\mathbf{S}} S_i\ \bar{\rho}_{gi} \\ \mid i \in \{1, \ldots, n\} \end{matrix} \right\} \cup \bigcup \left\{ \begin{matrix} R\ \bar{\alpha} <_{\mathbf{S}} [\bar{\rho}_{gi}/\bar{\beta}](S'_j\ \bar{\tau}_g) \\ \mid (S_i\ \bar{\beta} <_{\mathbf{S}} S'_j\ \bar{\tau}_g) \in \mathbf{S}_{S_i}, i \in \{1, \ldots, n\} \end{matrix} \right\}$$

   The right hand set above represents the subtype axioms induced by transitivity; S'_j denote the supertypes of R's supertypes S_i. These axioms are instantiated with the argument types of S_i in the subtype declaration. By $[\bar{\rho}_{gi}/\bar{\beta}]$ we denote a type substitution that replaces the type variables $\bar{\beta}$ with their corresponding argument types $\bar{\rho}_{gi}$. As done in [Nor99], we prohibit cycles in the subtyping axioms and require that the subtype theory $\mathbf{S}$ is closed under transitivity.

3. A label closure $\widehat{R} = \{l_j \mid j \in \{1, \ldots, m\}\} \cup \bigcup \{\widehat{S_i} \mid i \in \{1, \ldots, n\}\}$, which represents the full record including inherited record labels from the supertypes S_i.

4. A label environment

$$\prod\nolimits_{\widehat{R}} = \left\{ \begin{matrix} l_j : \forall \bar{\alpha}. R\ \bar{\alpha} \to \tau_{g_j} \\ \mid j \in \{1, \ldots, m\} \end{matrix} \right\} \cup \bigcup \left\{ \begin{matrix} l : \forall \bar{\alpha}. R\ \bar{\alpha} \to [\bar{\rho}_{gi}/\bar{\beta}]\tau \\ \mid (l : \forall \bar{\beta}. S_i\ \bar{\beta} \to \tau) \in \prod\nolimits_{\widehat{S_i}}, i \in \{1, \ldots, n\} \end{matrix} \right\}$$

representing the instantiated type schemes of the labels in R's label closure $\widehat{R}$. Similar to above, the right hand set denotes record labels induced by transitivity from the supertypes S_i. In the type schemes of inherited labels, we replace the argument types $S_i\ \bar{\beta}$ with the declared type $R\ \bar{\alpha}$.

Consider a variant type declaration $V\ \bar{\alpha} >_{\mathbf{S}} \{\overline{S_i\ \bar{\rho}_{gi}}\} \; = \; \overline{k_j : \bar{\tau}_{gj}}$, where $i \in \{1,\dots,n\}$ and $j \in \{1,\dots,m\}$, i.e., the type $V\ \bar{\alpha}$ has n subtypes and m new variant constructors. Then we define the following:

1. A type scheme $k_j : \forall\bar{\alpha}.\tau_{g_1} \times \dots \times \tau_{g_n} \rightarrow V\ \bar{\alpha}$ for each new variant constructor, which can be used as a regular function symbol (with the built-in semantics to construct a variant).
2. A set $\mathbf{S}_V \in \mathbf{S}$ of transitively closed subtype axioms:

   $$\mathbf{S}_V = \left\{ \begin{array}{l} S_i\ \bar{\rho}_{gi} <_{\mathbf{S}} V\ \bar{\alpha} \\ \mid i \in \{1,\dots,n\} \end{array} \right\} \cup \bigcup \left\{ \begin{array}{l} [\bar{\rho}_{gi}/\bar{\beta}](S'_i\ \bar{\tau}_g) <_{\mathbf{S}} V\ \bar{\alpha} \\ \mid (S'\ \bar{\tau}_g <_{\mathbf{S}} S_i\ \bar{\beta}) \in \mathbf{S}_{S_i}, i \in \{1,\dots,n\} \end{array} \right\}$$

   The right hand set above represents the subtype axioms induced by transitivity. These axioms are instantiated with the argument types of S_i in the supertype declaration.
3. A constructor closure $\widehat{V} = \{k_j \mid j \in \{1,\dots,m\}\} \cup \bigcup\{\widehat{S_i} \mid i \in \{1,\dots,n\}\}$, which represents the full variant type including inherited constructors.
4. A constructor environment

   $$\begin{array}{l} \sum_{\widehat{V}} = \left\{ k_j : \forall\bar{\alpha}.\tau_{g_1} \times \dots \times \tau_{g_{n_{k_j}}} \rightarrow V\ \bar{\alpha} \mid j \in \{1,\dots,m\} \right\} \cup \\ \quad \bigcup \left\{ \begin{array}{l} k : \forall\bar{\alpha}.[\bar{\rho}_{gi}/\bar{\beta}](\tau_{g'_1} \times \dots \times \tau_{g'_{n_k}}) \rightarrow V\ \bar{\alpha} \\ \Big| \ (k : \forall\bar{\beta}.\tau_{g'_1} \times \dots \times \tau_{g'_{n_k}} \rightarrow S_i\ \bar{\beta}) \in \sum_{\widehat{S_i}}, i \in \{1,\dots,n\} \end{array} \right\} \end{array}$$

   representing the instantiated type schemes of the variant constructors in $\widehat{V}$. In the type schemes of inherited constructors, the result types $S_i\ \bar{\beta}$ are replaced with the declared type $V\ \bar{\alpha}$.

4.3.2 Typing Consistency Rules

Fig. 4.7 shows the well-typedness rules for terms and formulae, which follow the shape

$\dfrac{A_1,\dots,A_n}{B}$	The judgements A_1 through A_n must hold, in order to infer the judgement B.

Well-typedness rules define a desired property of terms and formulae. This is in contrast to type checking, which tries to infer types for terms. A well-typedness judgement for a term follows the pattern $C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \sigma$ and reads: "In the subtype theory $\mathbf{S}$, under subtype constraints C, variable assumptions Γ, and symbol assumptions Δ the term e has the type scheme σ." A judgement for a formula $C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \phi$ ensures that the formula ϕ is well-typed. The context Δ holds the types of all symbols in a signature Σ. Γ holds the types for variables

$$\frac{}{C,\ \Gamma \cup \{x : \sigma_g\},\ \Delta\ \vdash_{\mathbf{S}}\ x : \sigma_g}$$ TypVar

$$\frac{}{C,\ \Gamma,\ \Delta \cup \{s : \sigma\}\ \vdash_{\mathbf{S}}\ s : \sigma}$$ TypSym

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ s : \tau_1 \times \ldots \times \tau_n \rightarrow \tau_g \qquad C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e_i : \tau_i}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ s(e_1, \ldots, e_n) : \tau_g}$$ TypSymApp

$$\frac{\begin{array}{c}\prod_{\widehat{R}} = \{\overline{l_i : \forall \bar{\alpha}.R\ \bar{\alpha} \rightarrow \tau_{g_i}}\} \\ C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e_i : [\bar{\rho}/\bar{\alpha}]\tau_{g_i}\end{array}}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ K_R\{\overline{l_i = e_i}\} : [\bar{\rho}/\bar{\alpha}](R\ \bar{\alpha})}$$ TypStruct

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : [\bar{\rho}/\bar{\alpha}](V\ \bar{\alpha}) \\ \sum_{\widehat{V}} = \{\overline{k_i : \forall \bar{\alpha}.\tau_{g_1} \times \ldots \times \tau_{g_{n_i}} \rightarrow V\ \bar{\alpha}}\} \\ C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ s_i : [\bar{\rho}/\bar{\alpha}](\tau_{g_1} \times \ldots \times \tau_{g_{n_i}}) \rightarrow \tau_g\end{array}}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \mathsf{case}(e, \{\overline{k_i \rightarrow s_i}\}) : \tau_g}$$ TypCase

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : [\bar{\rho}/\bar{\alpha}](V\ \bar{\alpha}) \\ \sum_{\widehat{V}} = \{\overline{k_i : \forall \bar{\alpha}.\tau_{g_1} \times \ldots \times \tau_{g_{n_i}} \rightarrow V\ \bar{\alpha}}\} \\ C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ s_i : [\bar{\rho}/\bar{\alpha}](\tau_{g_1} \times \ldots \times \tau_{g_{n_i}}) \rightarrow \tau_g\end{array}}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \mathsf{case}(e, V, \{\overline{k_i \rightarrow s_i}\}) : \tau_g}$$ TypCase$V$

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \tau \qquad \mathsf{tv}(\tau) = \emptyset}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ (e :: \tau) : \tau}$$ TypAnno

$$\frac{C \cup D,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \tau \qquad \bar{\alpha} \notin \mathsf{tv}(C) \cup \mathsf{tv}(\Gamma)}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \forall \bar{\alpha}.\tau \mid D}$$ TypGen

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \forall \bar{\alpha}.\tau \mid D \qquad C\ \vdash_{\mathbf{S}}\ [\bar{\tau}/\bar{\alpha}]D}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : [\bar{\tau}/\bar{\alpha}]\tau}$$ TypInst

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \tau \qquad C\ \vdash_{\mathbf{S}}\ \tau \leq \tau'}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \tau'}$$ TypSub

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ p : \tau_1 \times \ldots \times \tau_n \rightarrow \mathsf{Bool} \qquad C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e_i : \tau_i}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ p(e_1, \ldots, e_n)}$$ TypPredApp

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \phi}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \neg\phi}$$ TypNot

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \phi \qquad C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \psi}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ \phi \cdot \psi}$$ TypBin

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : [\tau] \qquad C,\ \Gamma \cup \{x : \tau\},\ \Delta\ \vdash_{\mathbf{S}}\ \phi}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ Q\ x \in e \bullet \phi}$$ TypQuant

Figure 4.7: Well-typedness rules for terms and formulae

introduced by quantifiers. For brevity, we denote multiple judgements by the shape $C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e_i : \sigma_i$, where i is clear from the context.

Next, we discuss some of the typing rules. TypVar, TypGen, TypInst, and TypSub directly carry over from [Nor99], where TypSub defines subsumption through subtyping (see below for a definition of the subtyping relation $\leq$). TypSym introduces typings of function symbols and predicate symbols, respectively; recall that we regard record labels and variant constructors as regular function symbols. Symbol application in rule TypSymApp corresponds to general term application in [Nor99] (where it is called TypApp). We only support full application of a symbol to argument terms, in order to ensure the first-order properties of our logic. The rules TypStruct, TypCase, and TypCase$V$ deviate from their corresponding rules in [Nor99], due to the use of explicit record constructors and variant type constructors. We use explicit record constructors, in order to uniquely determine the type of a constructed record. In rule TypCase$V$, the variant type constructor V determines the type of the `case` scrutinee e. We only permit type annotations with monomorphic types, because, finally, each term must have a monomorphic type, in order to support first-order semantics of our rules. We reflect this in rule TypAnno, which corresponds to TypSig in [Nor99].

Typing rules for formulae are straightforward. For negated formulae (TypNot) and binary connectives (TypBin), we require that their subformulae are well typed. We use the symbol "$\cdot$" as an abbreviation for logical binary connectives, namely $\vee$, $\wedge$, and $\Rightarrow$. From the typing perspective, a quantified formula roughly behaves like a `let` construct in a functional programming language. A quantifier introduces a new variable x and makes this variable available in the subformula ϕ. When the quantifier is evaluated, x iterates over all values in the list, calculated by the term e. Rule TypQuant deviates from rule TypLet in [Nor99], because the quantifier sphere term e must have the *list* type $[\tau]$, if the quantified variable x has the type τ. Clearly, e can be regarded as a "container" holding the values for x.[12] In contrast, in a `let` construct x and e must have the same type. In rule TypQuant, we let Q denote either a universal or an existential quantifier.

The well-typedness rule TypSub uses the subtype relation $\leq$, in order to determine whether two types are related by subtyping. As usual, $\leq$ is an ordering relation, induced by the subtype axioms in a signature. Since $\leq$ carries over from [Nor99], we give its derivation rules for completeness only. A judgement in Fig. 4.8 follows the shape $C\ \vdash_{\mathbf{S}}\ \tau \leq \rho$ reading: "In the subtype theory $\mathbf{S}$, under the subtype constraints in C, τ is a subtype of ρ." Notice the overloading of the symbol $\vdash_{\mathbf{S}}$, which is used for both well-typedness judgements and subtyping relation judgements. We do so for simpler notation.

In our setting, all type constructors are covariant; they comprise record types and variant types only. As usual, function types are covariant in their result types and contravariant in their argument types. Recall that we regard predicate types as special function types with result type Bool. Rule SubConst

[12] Recall, that we expect Haskell programmers more familiar with lists than with sets.

$$\frac{}{C \cup \{\tau \leq \rho\} \vdash_{\mathbf{S}} \tau \leq \rho} \quad \text{SubHyp}$$

$$\frac{}{C \vdash_{\mathbf{S}} \tau \leq \tau} \quad \text{SubRefl}$$

$$\frac{C \vdash_{\mathbf{S}} \tau \leq \rho \qquad C \vdash_{\mathbf{S}} \rho \leq \tau'}{C \vdash_{\mathbf{S}} \tau \leq \tau'} \quad \text{SubTrans}$$

$$\frac{C \vdash_{\mathbf{S}} \rho_i \leq \tau_i \qquad C \vdash_{\mathbf{S}} \tau_g \leq \rho_g}{C \vdash_{\mathbf{S}} \tau_1 \times \ldots \times \tau_n \to \tau_g \ \leq \ \rho_1 \times \ldots \times \rho_n \to \rho_g} \quad \text{SubFunPred}$$

$$\frac{C \vdash_{\mathbf{S}} \tau_{g_i} \leq \rho_{g_i}}{C \vdash_{\mathbf{S}} T\, \tau_{g_1} \ldots \tau_{g_n} \ \leq \ T\, \rho_{g_1} \ldots \rho_{g_n}} \quad \text{SubDepth}$$

$$\frac{\tau <_{\mathbf{S}} \rho}{C \vdash_{\mathbf{S}} \theta\tau \leq \theta\rho} \quad \text{SubConst}$$

Figure 4.8: Subtype relation

relates the subtype relation $\leq$ to the subtype axioms in the subtype theory $\mathbf{S}$. If a type τ is *defined* to be a subtype of a type ρ, then any instantiation $\theta\tau$ is a subtype of the instantiation $\theta\rho$, where θ is a type substitution.

4.3.3 Type Inference

Our type inference algorithm detects whether a consistency rule is well typed. With each term the algorithm tries to associate a monomorphic type. Although Nordlander's type inference algorithm is *incomplete* [Nor99], we use it because it is fast and easy to comprehend. The algorithm does not involve sophisticated constraint solving, as proposed by many other authors [Smi91, Smi94, Hen96, MW97, Pot98, HPT98, Pot01]. Whereas complete subtype inference is NP-hard [LM92], Nordlander's quasi-linear algorithm is almost as fast as standard Hindley-Milner type inference. The major source for the algorithm's incompleteness is partial application of terms, which we exclude. In addition, we use explicit record constructors and an annotated `case` statement, providing further guidance. Notice that our type inference algorithm is still incomplete; we show an example below.

Applied to a formula ϕ, our type inference algorithm returns a type substitution θ that should instantiate all type variables in ϕ to monomorphic types, which is not always possible for well-typed formulae (see below). Fig. 4.9 shows our type inference algorithm. Rule judgements for terms follow the pattern

$$\overset{\downarrow}{C},\ \overset{\uparrow}{\Gamma},\ \overset{\uparrow}{\Delta} \Vdash_{\mathbf{S}} \overset{\uparrow}{e}\ :\ \overset{\downarrow}{\theta}\ (\overset{\uparrow}{\sigma})$$

where up-arrows denote inputs to the algorithm and down-arrows denote outputs. It is easy to follow these arrows: in a derivation, inputs move up and outputs move down the derivation tree, where the root is at the bottom of a derivation. The above rule judgement reads: “Given variable assumptions Γ, symbol assumptions Δ, a subtype theory $\mathbf{S}$, an expression e, and an *expected*

type scheme σ, return subtype constraints C and a type substitution θ, instantiating σ." Then, we assign the (preliminary) type scheme $\theta\sigma$ to e. This type scheme may become more concrete by other type inference steps. The subtyping constraint set C is built by variables that are bound to polymorphic terms. This allows to instantiate the types for such variables "later" (see the discussion about the rule ChkQuant below). In rule judgements for formulae, we omit expected type schemes; these judgements indicate that type inference has succeeded. For our type inference algorithm, we regard type variables ν as fresh, i.e., they do not occur in the derivation before. Similar to the previous section, Δ holds assumptions about symbols in the signature, Γ holds assumptions about variables introduced by quantifiers. As usual, every well-typedness rule from the previous section has a corresponding type inference rule. Notable exceptions are the well-typedness rules TypGen, TypInst, and TypSub. They are represented by solving subtyping constraints, built by the other type inference rules; we define resolution of subtyping constraints below.

The rules ChkVar and ChkSym correspond to the rule ChkVar in [Nor99]. For better comprehension, we delay a detailed discussion about ChkVar until after we have discussed the rule ChkQuant (see below). Function symbols have a fully quantified type, consequently they do not introduce new subtyping constraints. ChkSymApp infers the type for a symbol application, where we permit full application only. Type checking record constructions and variant deconstructions is straightforward, where the only critical rule is ChkCase. In ChkCase, we "guess" the variant type $V_j\ \bar{\alpha}$ by its environment. Of course, if a signature includes multiple variant types with the same symbols in the environment, by ChkCase we cannot infer a correct typing although the term may be well typed. Explicit annotation of a `case` construct can avoid this confusion. In a type annotation, we require that the annotated type is monomorphic. This simplifies type checking since it prohibits subtyping constraints introduced by user type annotations. Similar to [Nor99], user supplied type annotations are propagated down the term structure as an aid to type inference.

Type checking formulae is straightforward. Predicate symbol introduction is covered by function symbol introduction (ChkSym); predicate symbol application (ChkPredApp) corresponds to symbol application. Our *monomorphic* rule ChkQuant deviates from the polymorphic ChkLet rule in [Nor99]. We infer well-typedness of a quantified formula, if its sphere term e has a list type $[\nu]$ and we can infer well-typedness of the subformula ϕ under the assumption that the bound variable x has the type $\theta\nu$. The type substitution θ (returned from inferring e's type) instantiates ν. Similar to [Nor99], we solve new subtype constraints that reference type variables free in Γ.[13] Without generalizing the type of the bound variable x, we permit the subformula ϕ to instantiate the type of x to a monomorphic type. This is not possible in Nordlander's polymorphic ChkLet rule. For example, in $\forall\, t \in [] \bullet \texttt{repDs}(t) = []$ the type of t is instantiated to `State` by the subformula $\texttt{repDs}(t) = []$. In $\forall\, t \in [] \bullet t = []$,

[13] C_Γ returns subtype constraints that do not reference type variables free in Γ:
$C_\Gamma = \{\tau \leq \rho \mid \tau \leq \rho \in C \text{ and } (\mathsf{tv}(\tau) \cup \mathsf{tv}(\rho)) \cap \mathsf{tv}(\Gamma) = \emptyset\}$

$$\dfrac{\begin{array}{c}\bar{\beta} = \mathsf{tv}(\tau) \qquad C = \{\overline{\beta \leq \nu}\} \\ \theta \Vdash_{\mathbf{S}} \{[\bar{\nu}/\bar{\beta}]\tau \leq \tau'\}\end{array}}{\theta C,\ \Gamma \cup \{x : \tau\},\ \Delta \Vdash_{\mathbf{S}} x : \theta(\tau')} \quad \text{ChkVar}$$

$$\dfrac{\theta \Vdash_{\mathbf{S}} \{[\bar{\nu}/\bar{\alpha}]\tau \leq \tau'\}}{\emptyset,\ \Gamma,\ \Delta \cup \{s : \forall\bar{\alpha}.\tau\} \Vdash_{\mathbf{S}} s : \theta(\tau')} \quad \text{ChkSym}$$

$$\dfrac{\begin{array}{c}C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e_i : \theta_i(\nu_i) \\ C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} s : \theta(\theta_1\nu_1 \times \ldots \times \theta_n\nu_n \to \tau)\end{array}}{C \cup \bigcup \theta C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} s(e_1, \ldots, e_n) : \theta(\tau)} \quad \text{ChkSymApp}$$

$$\dfrac{\begin{array}{c}\prod_{\widehat{R}} = \{\overline{l_i : \forall\bar{\alpha}.R\ \bar{\alpha} \to \tau_{g_i}}\} \\ \theta \Vdash_{\mathbf{S}} \{[\bar{\nu}/\bar{\alpha}](R\ \bar{\alpha}) \leq \rho_g\} \qquad \overline{\rho_{g_i} = \theta[\bar{\nu}/\bar{\alpha}]\tau_{g_i}} \\ C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e_i : \theta_i(\rho_{g_i}) \qquad \theta' \Vdash_{\mathbf{S}} \{\overline{\theta_i\rho_{g_i} \leq \rho_{g_i}}\}\end{array}}{\bigcup \theta' C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} K_R\{\overline{l_i = e_i}\} : \theta'\theta(\rho_g)} \quad \text{ChkStruct}$$

$$\dfrac{\begin{array}{c}\sum_{\widehat{V_j}} = \{\overline{k_{j,i} : \forall\bar{\alpha}.\tau_{g_{j,i,1}} \times \ldots \times \tau_{g_{j,i,n_{j,i}}} \to V_j\ \bar{\alpha}}\} \\ C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e : \theta(\nu) \\ \theta' \Vdash_{\mathbf{S}} \{\theta\nu \leq [\bar{\nu}/\bar{\alpha}](V_j\ \bar{\alpha})\} \text{ for exactly one } j \\ \theta_i \Vdash_{\mathbf{S}} \{\nu_i \leq [\bar{\nu}/\bar{\alpha}](\tau_{g_{i,1}} \times \ldots \times \tau_{g_{i,n_i}}) \to \rho_g\} \\ \rho_i = \theta_i\nu_i \\ C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} s_i : \theta'_i(\rho_i) \qquad \theta'' \Vdash_{\mathbf{S}} \{\overline{\theta'_i\rho_i \leq \rho_i}\}\end{array}}{\theta'(C \cup \bigcup C_i),\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \mathsf{case}(e, \{\overline{k_i \to s_i}\}) : \theta''\theta'(\rho_g)} \quad \text{ChkCase}$$

$$\dfrac{\begin{array}{c}\sum_{\widehat{V}} = \{\overline{k_i : \forall\bar{\alpha}.\tau_{g_{i,1}} \times \ldots \times \tau_{g_{i,n_i}} \to V\ \bar{\alpha}}\} \\ C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e : \theta(\nu) \qquad \theta' \Vdash_{\mathbf{S}} \{\theta\nu \leq [\bar{\nu}/\bar{\alpha}](V\ \bar{\alpha})\} \\ \theta_i \Vdash_{\mathbf{S}} \{\nu_i \leq [\bar{\nu}/\bar{\alpha}](\tau_{g_{i,1}} \times \ldots \times \tau_{g_{i,n_i}}) \to \rho_g\} \\ \rho_i = \theta_i\nu_i \\ C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} s_i : \theta'_i(\rho_i) \qquad \theta'' \Vdash_{\mathbf{S}} \{\overline{\theta'_i\rho_i \leq \rho_i}\}\end{array}}{\theta'(C \cup \bigcup C_i),\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \mathsf{case}(e, V, \{\overline{k_i \to s_i}\}) : \theta''\theta'(\rho_g)} \quad \text{ChkCase}V$$

$$\dfrac{\mathsf{tv}(\tau) = \emptyset \qquad C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e : \theta(\tau) \qquad \theta' \Vdash_{\mathbf{S}} \tau \leq \tau'}{\theta C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} (e :: \tau) : \theta'(\tau')} \quad \text{ChkAnno}$$

$$\dfrac{\begin{array}{c}C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e_i : \theta_i(\nu_i) \\ C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} p : \theta(\theta_1\nu_1 \times \ldots \times \theta_n\nu_n \to \mathsf{Bool})\end{array}}{C \cup \bigcup \theta C_i,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} p(e_1, \ldots, e_n),\ \theta} \quad \text{ChkPredApp}$$

$$\dfrac{C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \phi,\ \theta}{C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \neg\phi,\ \theta} \quad \text{ChkNot}$$

$$\dfrac{C_1,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \phi,\ \theta_1 \qquad C_2,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \psi,\ \theta_2}{\theta_1\theta_2(C_1 \cup C_2),\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} \phi \cdot \psi,\ \theta_1 \circ \theta_2} \quad \text{ChkBin}$$

$$\dfrac{\begin{array}{c}C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e : \theta([\nu]) \qquad C',\ \Gamma \cup \{x : \theta\nu\},\ \Delta \Vdash_{\mathbf{S}} \phi,\ \theta' \\ \theta'' \Vdash_{\mathbf{S}} C' \setminus C'_{\Gamma}\end{array}}{\theta''(C \cup C'_{\Gamma}),\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} Q\ x \in e \bullet \phi,\ \theta'' \circ \theta'} \quad \text{ChkQuant}$$

Figure 4.9: Combined type inference and type checking algorithm

$$\frac{}{[] \Vdash_{\mathbf{S}} \emptyset} \quad \text{CTriv}$$

$$\frac{\theta \Vdash_{\mathbf{S}} C \cup \{\overline{\tau_i \leq \rho_i}\}}{\theta \Vdash_{\mathbf{S}} C \uplus \{T\ \tau_1 \ldots \tau_n \leq T\ \rho_1 \ldots \rho_n\}} \quad \text{CDepth}$$

$$\frac{\theta \Vdash_{\mathbf{S}} C \cup \{\overline{\rho_i \leq \tau_i}\} \cup \{\tau_g \leq \rho_g\}}{\theta \Vdash_{\mathbf{S}} C \uplus \{\tau_1 \times \ldots \times \tau_n \to \tau_g \leq \rho_1 \times \ldots \times \rho_n \to \rho_g\}} \quad \text{CFunPred}$$

$$\frac{T\ \bar{\tau}' <_{\mathbf{S}} S\ \bar{\rho}' \qquad \bar{\alpha} = \mathsf{tv}(\bar{\tau}') \cup \mathsf{tv}(\bar{\rho}') \qquad \theta \Vdash_{\mathbf{S}} C \cup \{T\ \bar{\tau} \leq [\bar{\nu}/\bar{\alpha}](T\ \bar{\tau}')\} \cup \{[\bar{\nu}/\bar{\alpha}](S\ \bar{\rho}') \leq S\ \bar{\rho}\}}{\theta \Vdash_{\mathbf{S}} C \uplus \{T\ \bar{\tau} \leq S\ \bar{\rho}\}} \quad \text{CSub}$$

$$\frac{\alpha \notin \mathsf{tv}(\tau_i) \cup \mathsf{tv}(\rho_i) \qquad \tau = \mathsf{smallest}(\bigsqcup\{\overline{\tau_i}\}, \bigsqcap\{\overline{\rho_j}\}) \qquad \bar{\alpha} = \mathsf{tv}(\tau) \qquad \tau' = [\bar{\nu}/\bar{\alpha}]\tau \qquad \theta \Vdash_{\mathbf{S}} [\tau'/\alpha]C \cup \{\overline{\tau_i \leq \tau'}\} \cup \{\overline{\tau' \leq \rho_j}\}}{\theta \circ [\tau'/\alpha] \Vdash_{\mathbf{S}} C \uplus \{\overline{\tau_i \leq \alpha}\} \cup \{\overline{\alpha \leq \rho_j}\}} \quad \text{CMerge}$$

$$\frac{\theta \Vdash_{\mathbf{S}} [\beta/\alpha]C}{\theta \circ [\beta/\alpha] \Vdash_{\mathbf{S}} C \uplus \{\alpha \leq \beta\}} \quad \text{CVar}$$

Figure 4.10: Solving subtype constraints

however, the type of t cannot be instantiated to a monomorphic type; we can infer $t : [\alpha]$ only. A monomorphic type for t can be achieved with the help of explicit type annotation, e.g., $\forall\, t \in [] \bullet t : [\texttt{String}] \;=\; []$.

The rule ChkVar is quite simple in our setting, because variables cannot have a function type and their type is not generalized. Recall that variables are introduced by quantifiers only and we prohibit quantification over functions. The rule ChkVar introduces new subtyping constraints for a variable x, if its typing in the context Γ is polymorphic, i.e., if the quantifier sphere term for x is polymorphic. In that case the subtype constraints move down the derivation tree. They are caught by the quantifier for x as explained above. The reason for this rather complicated behavior is that the type of a bound variable x can be instantiated by the subformula of a quantifier.

We only make slight adaptations to the solving procedures for subtyping constraints (see Fig. 4.10). A constraint solving judgement follows the shape

$$\overset{\downarrow}{\theta} \Vdash_{\mathbf{S}} \overset{\uparrow}{C}$$

reading: "Given a constraint set C and a subtype theory $\mathbf{S}$, return the type substitution θ." For simpler notation, we overload the symbol $\Vdash_{\mathbf{S}}$, which is used for both type inference judgements and constraint solving judgements.

Nordlander's rule CDepth splits into CDepth for covariant type constructors and CFunPred for function types. The rule CFunPred also solves predicate type constraints. We choose to make this adaptation since the only types with (partially) contravariant behavior are function types and predicate types. All other type constructors are covariant. In CMerge, the auxiliary function smallest returns the smaller of two given types. We denote the least upper bound of some types by $\bigsqcup$ and the greatest lower bound by $\bigsqcap$. Least upper

bounds and greatest lower bounds, respectively, may not exist in certain situations, which causes type inference to fail due to ambiguity. This ambiguity can be resolved by explicit type annotation. The symbol $\uplus$ denotes the disjoint union between sets. The rule CVar is the major source for the algorithm's incompleteness, because in variable-variable constraints type variables are simply unified. Notice that the order of the solving rules is important. In particular, CSub must be applied prior to CMerge, which in turn must be applied prior to CVar.

4.3.4 Examples

In this section, we illustrate our combined type checking and type inference algorithm. We infer well-typedness of our example rule ϕ_2:

$$\begin{array}{l}\forall\, t_1 \in \texttt{repStates} \bullet \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \forall\, t_2 \in \texttt{repStates} \bullet \\ t_1 < t_2 \;\;\Rightarrow\;\; \begin{pmatrix}\exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{dId}(m_1) = \texttt{dId}(m_2) \;\wedge\; \texttt{kind}(m_1) = \texttt{kind}(m_2)\end{pmatrix}\end{array}$$

We follow the derivation from the root to the leaves from an initially empty context Γ. Recall that the contexts Γ and Δ, the expression or formula to type check, and expected types move *up* the derivation tree. In contrast, type substitutions θ move *down* the derivation tree. Since all quantifier sphere terms in ϕ_2 are monomorphic, we omit the subtyping constraints C for better comprehensibility. We enumerate type variables ν, introduced during type inference, by consecutive numbers. In addition, we mark the different type substitutions θ by the symbols or variables, which "produced" these type substitutions.

The root of the derivation employs the rule ChkQuant:

$$\frac{\begin{array}{c}\emptyset,\ \Delta \Vdash_{\mathbf{S}} \texttt{repStates} : \theta_{t_1}([\nu_0]) \\ \{t_1 : \theta_{t_1}\nu_0\},\ \Delta \Vdash_{\mathbf{S}} \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \ \ldots,\ \theta'\end{array}}{\emptyset,\ \Delta \Vdash_{\mathbf{S}} \forall\, t_1 \in \texttt{repStates} \bullet \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \ \ldots,\ \theta'}\ \text{ChkQuant}$$

The premises of the above rule require for two steps: First, we infer the type of the term `repStates`; second, we infer well-typedness of the subformula $\forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \ \ldots$.

1. Type inference for the term `repStates` succeeds with the help of the rule ChkSymApp, which reduces to ChkSym, because `repStates` has no argument terms:

$$\frac{\dfrac{\theta_{t_1} \ \Vdash_{\mathbf{S}} \ \{[\mathsf{State}] \leq [\nu_0]\}}{\emptyset,\ \Delta \cup \{\texttt{repStates} : [\mathsf{State}]\} \Vdash_{\mathbf{S}} \texttt{repStates} : \theta_{t_1}([\nu_0])}\ \text{ChkSym}}{\emptyset,\ \Delta \Vdash_{\mathbf{S}} \texttt{repStates} : \theta_{t_1}([\nu_0])}\ \text{ChkSymApp}$$

   The type substitution θ_{t_1} results from solving the constraint set $\{[\mathsf{State}] \leq [\nu_0]\}$ as follows (the constraint set on the right hand side of a constraint

solving judgement moves up the derivation tree; the type substitution on the left hand side moves down the derivation tree):

$$\begin{array}{c} \hline [\,]\ \Vdash_{\mathsf{S}}\ \emptyset \\ \hline [\,]\ \Vdash_{\mathsf{S}}\ \{\mathsf{State} \leq \mathsf{State}\} \\ \hline [\mathsf{State}/\nu_0]\ \Vdash_{\mathsf{S}}\ \{\mathsf{State} \leq \nu_0\} \\ \hline [\mathsf{State}/\nu_0]\ \Vdash_{\mathsf{S}}\ \{[\mathsf{State}] \leq [\nu_0]\} \end{array} \qquad \begin{array}{l} \text{CTriv} \\ \text{CSub} \\ \text{CMerge} \\ \text{CDepth} \end{array}$$

We obtain $\theta_{t_1} = [\mathsf{State}/\nu_0]$ and assign the type $[\mathsf{State}]$ to `repStates`.

2. We infer well-typedness of the formula $\forall\ m_1 \in \texttt{repManDs}(t_1) \bullet \ldots$ by employing the rule ChkQuant again. The derivation proceeds similar to above; therefore, we omit most of it. Instead, we get into type inference of the atomic formula $\texttt{dId}(m_1) = \texttt{dId}(m_2)$, i.e., the left hand side of the conjunction below the existential quantifier for m_2. At this point, the context Γ contains the typings of all quantified variables:

   $$\Gamma = \{t_1 : \mathsf{State}, m_1 : \texttt{ManD}, t_2 : \mathsf{State}, m_2 : \texttt{ManD}\}$$

   We proceed using the rule ChkPredApp

   $$\frac{\begin{array}{l} \Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ \texttt{dId}(m_1) : \theta_1(\nu_1) \\ \Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ \texttt{dId}(m_2) : \theta_2(\nu_2) \\ \Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ = \qquad : \theta_{=}(\theta_1\nu_1 \times \theta_2\nu_2 \to \mathsf{Bool}) \end{array}}{\Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ \texttt{dId}(m_1) = \texttt{dId}(m_2),\ \theta_{=}}\ \text{ChkPredApp}$$

   which requires three premises.

   (a) We infer the type of the term $\texttt{dId}(m_1)$, which results in the type substitution θ_1:

   $$\frac{\begin{array}{l} \Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ m_1 : \theta_{m_1}(\nu_3) \\ \Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ \texttt{dId} : \theta_1(\theta_{m_1}\nu_3 \to \nu_1) \end{array}}{\Gamma,\ \Delta\ \Vdash_{\mathsf{S}}\ \texttt{dId}(m_1) : \theta_1(\nu_1)}\ \text{ChkSymApp}$$

   The type of m_1 is inferred by ChkVar (m_1 is already included in the context Γ)

   $$\frac{\theta_{m_1}\ \Vdash_{\mathsf{S}}\ \{\texttt{ManD} \leq \nu_3\}}{\Gamma \cup \{m_1 : \texttt{ManD}\},\ \Delta\ \Vdash_{\mathsf{S}}\ m_1 : \theta_{m_1}(\nu_3)}\ \text{ChkVar}$$

   where the constraint set $\{\texttt{ManD} \leq \nu_3\}$ is solved by applying CMerge, CSub, and CTriv, similar to above. This results in the type substitution $\theta_{m_1} = [\texttt{ManD}/\nu_3]$. Notice that, if m_1 had a polymorphic type in Γ, we would obtain a non-empty constraint set C from ChkVar.

   Now we can infer the type of `dId`, which involves subtyping, because we apply `dId` to a manual (not to a document).

   $$\frac{\theta_1\ \Vdash_{\mathsf{S}}\ \{\texttt{Doc} \to \texttt{String} \leq \texttt{ManD} \to \nu_1\}}{\Gamma,\ \Delta \cup \{\texttt{dId} : \texttt{Doc} \to \texttt{String}\}\ \Vdash_{\mathsf{S}}\ \texttt{dId} : \theta_1(\texttt{ManD} \to \nu_1)}\ \text{ChkSym}$$

We infer θ_1 as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{[] \Vdash_{\mathsf{S}} \emptyset}\text{CTriv}}{[] \Vdash_{\mathsf{S}} \{\mathtt{ManD} \leq \mathtt{ManD}\} \cup \{\mathtt{Doc} \leq \mathtt{Doc}\}}\text{CDepth (twice)}}{[] \Vdash_{\mathsf{S}} \{\mathtt{ManD} \leq \mathtt{Doc}\}}\text{CSub}}{[] \Vdash_{\mathsf{S}} \{\mathtt{ManD} \leq \mathtt{Doc}\} \cup \{\mathtt{String} \leq \mathtt{String}\}}\text{CDepth}}{[\mathtt{String}/\nu_1] \Vdash_{\mathsf{S}} \{\mathtt{ManD} \leq \mathtt{Doc}\} \cup \{\mathtt{String} \leq \nu_1\}}\text{CMerge}}{[\mathtt{String}/\nu_1] \Vdash_{\mathsf{S}} \{\mathtt{Doc} \rightarrow \mathtt{String} \leq \mathtt{ManD} \rightarrow \nu_1\}}\text{CFunPred}$$

The above derivation infers $\theta_1 = [\mathtt{String}/\nu_1]$.

(b) Similar to (a) we infer $\theta_2 = [\mathtt{String}/\nu_2]$.

(c) Finally, we infer the type of the predicate symbol $=$ with the help of ChkSym:

$$\cfrac{\theta_= \Vdash_{\mathsf{S}} \{\nu_4 \times \nu_4 \rightarrow \mathsf{Bool} \leq \mathtt{String} \times \mathtt{String} \rightarrow \mathsf{Bool}\}}{\emptyset,\ \Delta \cup \{= : \forall \alpha.\alpha \times \alpha \rightarrow \mathsf{Bool}\} \Vdash_{\mathsf{S}} \ = : \theta_=(\mathtt{String} \times \mathtt{String} \rightarrow \mathsf{Bool})}\text{ChkSym}$$

Our constraint solver infers the type substitution $\theta_= = [\mathtt{String}/\nu_4]$ with help of the rules CFunPred, CMerge, CDepth, and CTriv.

4.4 Summary

This chapter is concerned with the syntactic aspects of our approach. Rule designers formalize consistency rules in a variant of first-order temporal predicate logic with linear time. Our logic comprises a rich vocabulary including record construction, variant deconstruction via `case` statements, and higher-order functions. Parametric polymorphism and Cardelli-style subtyping facilitate the re-use of function and predicate symbols. Language designers define symbols used in rules. Besides symbols, languages also contain type declarations.

The languages induce a signature, which contains theoretical means that support type checking, e.g., record label environments, variant constructor environments, and subtyping axioms. For type checking consistency rules, we adapt algorithms from Nordlander's Haskell extension O'Haskell [Nor99]. O'Haskell supports parametric polymorphism and subtyping, which aids rule formalization and reduces the number of symbols to be defined. It turns out, however, that careful adaptations to Nordlander's type checking algorithm are necessary, in order to ensure the first-order properties of our logic.

The formal background behind the formalization of consistency rules may appear a rather complex matter. Notice, however, that rule designers and language designers are not affected by these issues. They just define rules and languages. Our tools derive a signature from these languages and ensure that well-typed rules can be formalized only. We consider type checking a key ingredient in our approach, because syntactical well-typedness of consistency rules is a vital prerequisite for consistency checking, which we discuss in the next chapter.

Chapter 5
Finding Inconsistencies

In this chapter, we present a *tolerant*[1] semantics for consistency rules that pinpoints the trouble spots making a repository inconsistent w.r.t. a rule system. We do not restrict ourselves to finding out whether a repository violates certain rules. Instead, we generate *consistency reports*, which indicate *when*, *where*, and *why* documents in the repository are inconsistent. Presenting consistency reports to authors, instead of simply rejecting a check-in, is the basis for consistency-aware DMSs.

Classic set theoretic semantics provides a boolean result only and is, therefore, insufficient for our purposes. Evaluation of a predicate logic formula binds, however, concrete values to quantified variables. The basic idea behind our tolerant semantics is to exploit these bindings, in order to indicate inconsistencies. We have designed our consistency checking algorithm to provide compact information that precisely characterize inconsistencies. Our tolerant semantics follows xlinkit [NCEF02], which calculates a set of links, each containing a consistency flag and values that are responsible for the violation of a formula. A major difference is that we also return those atomic formulae that cause rule violation. Authors need these details to precisely identify inconsistencies. Consistency rules can grow large and only some of many predicates can falsify a rule.

In this chapter, we proceed as follows: Sect. 5.1 gives an informal overview about how consistency reports indicate inconsistencies. In Sect. 5.2, we show the classic boolean semantics for our rules, i.e., we say when a rule is fulfilled in a repository. In Sect. 5.3, we define a basic consistency checking algorithm, which conforms to the boolean semantics. We illustrate report generation by our running example in Sect. 5.4. In Sect. 5.5, we introduce the formal structures needed for our approach and show how our tolerant semantics is related to classic predicate logic. Consistency checking relies on the computation of values from terms, which we review in Sect. 5.6. The formal details in Sect. 5.5

[1]Since we borrow many ideas from xlinkit [NCEF02], we adopt the term "tolerant."

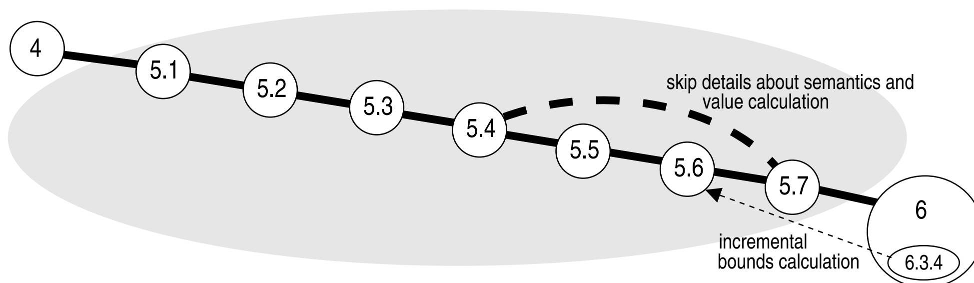


Figure 5.1: Chapter 5 in context

Check-in	Document	Modifications	
		Kind	Details
1	doc1.txt	add	...as shown in manual kaA3 ...
	keys.xml	add	<kDef key="kaA3"
			kId="man1.xml"
			kKind="technical M."/>
	man1.xml	add	<man kind="technical M."> ...
2	man1.xml	change	<man kind="field M."> ...
3	doc2.txt	add	...as shown in manual kaA2 ...
4	man2.xml	add	<man kind="field M."> ...
	man3.xml	add	<man kind="field M."> ...
	man4.xml	add	<man kind="field M."> ...

Table 5.1: Example repository up to state 4

and 5.6 are given for completeness; they are not essential for the further understanding of this thesis.[2] Finally in Sect. 5.7, we summarize this chapter. Fig. 5.1 illustrates the context of this chapter. For the moment, we shall neglect efficiency of consistency checking. In Chapter 6, we will develop methods to speed-up the basic algorithm introduced in this chapter.

5.1 Informal Overview

Given a repository state, for each consistency rule we generate a *consistency report* containing a boolean result (representing boolean truth semantics) and a diagnosis set. A diagnosis consists of a consistency flag, a variable assignment, and two sets of atomic formulae. The diagnosis $(\mathsf{IC}, \eta, \mathcal{F}_t, \mathcal{F}_f)$ reads: "The processed formula is violated (InConsistent) for the variable assignment η, due to fulfilled atomic formulae in the set $\mathcal{F}_t$ and violated atomic formulae in the set $\mathcal{F}_f$." The diagnosis $(\mathsf{C}, \eta, \mathcal{F}_t, \mathcal{F}_f)$ reads: "The processed formula is fulfilled (Consistent) for the variable assignment η, due to fulfilled atomic formulae in the set $\mathcal{F}_t$ and violated atomic formulae in the set $\mathcal{F}_f$." The variable assignment η binds variables to concrete values, i.e., repository states, documents, or document content. Thus, η indicates *when* and *where* a formula is violated.[3] If the consistency flag is IC, the sets $\mathcal{F}_t$ and $\mathcal{F}_f$ describe *why* a formula is violated; otherwise, $\mathcal{F}_t$ and $\mathcal{F}_f$ indicate why a formula is fulfilled.

Recall our example repository at the fourth state (shown in Tab. 5.1). The repository contains five inconsistencies w.r.t. rule ϕ_1:

- At state 2, the key reference to kaA3 in the document doc1.txt is inconsistent, because the definition of kaA3 in the key resolver keys.xml points to a technical manual man1.xml, which is, however, a field manual.

[2]Value calculation is, however, a prerequisite for incremental calculation of quantifier spheres, discussed in Sect. 6.3.4.

[3]Recall that we permit rectified formulae only, in which each quantifier binds a different variable.

$$\phi_1 = \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet$$
$$\exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \exists\, m \in \texttt{repManDs}(t) \bullet$$
$$k = \texttt{key}(d) \ \wedge\ \texttt{dId}(m) = \texttt{kId}(d) \ \wedge\ \texttt{kind}(m) = \texttt{kKind}(d)$$

$$\left(\mathsf{False}, \left\{\begin{array}{l}
\left(\begin{array}{l}\mathsf{IC}, \left\{t \mapsto 2, k \mapsto \mathsf{kaA3}, x \mapsto \{\texttt{dId} = \mathsf{doc1.txt}, \texttt{dState} = 1\}\right\}, \\ \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right), \\
\left(\begin{array}{l}\mathsf{IC}, \left\{t \mapsto 3, k \mapsto \mathsf{kaA3}, x \mapsto \{\texttt{dId} = \mathsf{doc1.txt}, \texttt{dState} = 1\}\right\}, \\ \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right), \\
\left(\begin{array}{l}\mathsf{IC}, \left\{t \mapsto 3, k \mapsto \mathsf{kaA2}, x \mapsto \{\texttt{dId} = \mathsf{doc2.txt}, \texttt{dState} = 3\}\right\}, \\ \emptyset, \{k = \texttt{key}(d), \texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right), \\
\left(\begin{array}{l}\mathsf{IC}, \left\{t \mapsto 4, k \mapsto \mathsf{kaA3}, x \mapsto \{\texttt{dId} = \mathsf{doc1.txt}, \texttt{dState} = 1\}\right\}, \\ \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right), \\
\left(\begin{array}{l}\mathsf{IC}, \left\{t \mapsto 4, k \mapsto \mathsf{kaA2}, x \mapsto \{\texttt{dId} = \mathsf{doc2.txt}, \texttt{dState} = 3\}\right\}, \\ \emptyset, \{k = \texttt{key}(d), \texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right)
\end{array}\right\}\right)$$

Figure 5.2: Example consistency report for rule ϕ_1 at state 4

- At state 3, the above inconsistency is still present. In addition, the key reference to kaA2 in the document doc2.txt is inconsistent, because no key resolver contains a definition for kaA2.
- At state 4, both inconsistencies from state 3 are still present.

The automatically generated report for ϕ_1 shown in Fig. 5.2 reflects these inconsistencies. The report's diagnoses do not include bindings to the variables d and m: The repository is inconsistent w.r.t. ϕ_1 for all possible assignments to d and m, respectively. The first diagnosis does not include the atomic formulae $k = \texttt{key}(d)$ and $\texttt{dId}(m) = \texttt{kId}(d)$: The key resolver contains kaA3, but part of its definition is inconsistent. At least one manual m with the correct name was found, but this manual has the wrong kind. In other words, the first step of the link is consistent, the second step is partly inconsistent. The second diagnosis indicates that the above inconsistency is still present at state 3. The third diagnosis indicates that there is no key resolver containing a definition for the key kaA2, which is referenced in the document doc2.txt. The fourth diagnosis corresponds to the second diagnosis; the fifth diagnosis corresponds to the third diagnosis.

Notice that variable assignments within diagnoses contain quantified variables only. Consequently, the rule designer must be careful when replacing quantifiers by applications of `concatMap`.

Authors can react in various plausible ways to the above report. An appropriate action would be to lookup the key definition for the key kaA3 in the key resolvers and to correct the kind of the manual this definition points to. In addition, a new key definition for the key kaA2 could be added to one of the key resolvers. Clearly, this requires good knowledge about the document structure and the consistency rule ϕ_1. In Part II, we present an approach to generate concrete repairs, which can resolve inconsistencies. For the moment, however, we concentrate on pinpointing inconsistencies only.

5.2 Validity of Rules

In this section, we define a truth value semantics for consistency rules, which employs an approach adopted from partial first-order predicate logic PFOL [Far01]. We have to use a *partial* logic, because function symbols may be interpreted by partial functions, which in turn means that evaluation of terms may fail.

As usual, we interpret consistency rules in a first-order structure A induced by the definitions from the language designer and the check-ins to the repository. The structure A interprets types by sets, predicate symbols by relations (which correspond to total characteristic functions), and function symbols by *partial* functions. A record label selects a component from a record value, a variant constructor builds a variant value. We interpret the type State by the set of natural numbers $\mathbb{N}$, representing repository states; we interpret the predicate symbol $\leq$ by the natural "less than or equal" ordering and the symbol $=$ by structural equality (if both argument terms are defined; otherwise, $e_1 = e_2$ and $e_1 \leq e_2$ do not hold). Since we facilitate temporal predicate logic, the structure A changes over time, which we discuss in Sect. 5.5.

Fig. 5.3 defines when a formula is valid in a structure A. As usual, we denote validity by the relation $\models$, where $A \models_\eta \phi$ means that the formula ϕ is valid in the structure A under the variable assignment η. As usual in predicate logic, the *variable assignment* η binds free variables in ϕ to values, i.e., repository states, documents, or document content. Type checking guarantees that η contains a binding of each variable free in ϕ. We define A and η formally in Sect. 5.5.

Let p^A denote the relation interpreting the predicate symbol p. Then an atomic formula $p(e_1, \ldots, e_n)$ is valid iff the values resulting from evaluating the argument terms e_i are in the relation p^A. If any argument term e_i is not defined, we follow [Far01] to define $p(e_1, \ldots, e_n)$ as not valid. Since we support higher-order predicates but want to keep our semantics simple, i.e., set theoretic, we have to distinguish symbol arguments from other arguments. Symbols s_i are interpreted by their implementation in the structure A, denoted by s_i^A. Other terms e_i are evaluated using the term evaluation function $\mathcal{V}_A[\![e_i]\!]\eta$, which evaluates the term e_i w.r.t. the structure A and the assignment η; see Sect. 5.6 for a formal definition. The result of $\mathcal{V}_A[\![e_i]\!]\eta$ is called *value*. Notice that, due to support for partial functions, $\mathcal{V}$ is partial.

For report generation, we need a special predicate symbol $\downarrow : \alpha$ that indicates whether a term is defined: $\downarrow(e)$ holds iff the value of e can be calculated.[4]

A negated formula is valid iff its subformula is not valid. A conjunction is valid iff both subformulae are valid. In contrast, a disjunction is valid already, if at least one subformula is valid. For validity of an implication $\phi \Rightarrow \psi$, we require that, if the first subformula ϕ is valid, then also the subformula ψ must be valid — this is equivalent to validity of the formula $\neg\phi \vee \psi$.

---

[4] We adopt the down arrow $\downarrow$ from [Far01] to denote definedness. In contrast to [Far01], $\downarrow$ is a predicate symbol and not an abbreviation ($\downarrow(e) \Leftrightarrow \exists\, x \bullet x = e$). We do so, because we want to store atomic formulae of the shape $\downarrow(e)$ in consistency reports.

$$
\begin{array}{lll}
A \models_\eta {\downarrow}(e) & :\Leftrightarrow & e^A \text{ is defined} \\
& \text{where } e^A = & s^A \quad \text{if } e \equiv s \\
& & \mathcal{V}_A[\![e]\!]\eta \quad \text{otherwise} \\[1em]
A \models_\eta p(e_1, \ldots, e_n) & :\Leftrightarrow & \text{all } e_i^A \text{ are defined and } (e_1^A, \ldots, e_n^A) \in p^A \\
& \text{where } e_i^A = & s_i^A \quad \text{if } e_i \equiv s_i \\
& & \mathcal{V}_A[\![e_i]\!]\eta \quad \text{otherwise} \\[1em]
A \models_\eta \neg\phi & :\Leftrightarrow & A \models_\eta \phi \text{ does not hold} \\[1em]
A \models_\eta \phi \wedge \psi & :\Leftrightarrow & A \models_\eta \phi \text{ and } A \models_\eta \psi \\[1em]
A \models_\eta \phi \vee \psi & :\Leftrightarrow & A \models_\eta \phi \text{ or } A \models_\eta \psi \\[1em]
A \models_\eta \phi \Rightarrow \psi & :\Leftrightarrow & A \models_\eta \neg\phi \vee \psi \\[1em]
A \models_\eta \forall\, x \in e \bullet \phi & :\Leftrightarrow & \mathcal{V}_A[\![e]\!]\eta \text{ is defined and} \\
& & A \models_{\eta \cup \{x \mapsto v\}} \phi \text{ for every } v \in \mathcal{V}_A[\![e]\!]\eta \\[1em]
A \models_\eta \exists\, x \in e \bullet \phi & :\Leftrightarrow & \mathcal{V}_A[\![e]\!]\eta \text{ is not defined, or} \\
& & \left(\begin{array}{l} \mathcal{V}_A[\![e]\!]\eta \text{ is defined and} \\ A \models_{\eta \cup \{x \mapsto v\}} \phi \text{ for at least one } v \in \mathcal{V}_A[\![e]\!]\eta \end{array}\right)
\end{array}
$$

Figure 5.3: Classic truth value semantics

For a universally quantified formula $\forall\, x \in e \bullet \phi$, we require validity of the subformula ϕ, where the variable x is bound to every value in the evaluated quantifier sphere e.[5] If its quantifier sphere is not defined, then a universally quantified formula is not valid. For an existentially quantified formula, it is sufficient that the subformula is valid for at least one value in the quantifier sphere. If its quantifier sphere is not defined, then an existentially quantified formula is valid, which may appear surprising at first sight. We see the soundness of our definition, if we translate an existentially quantified formula to classic predicate logic:

$$\exists\, x \in e \bullet \phi \;\Leftrightarrow\; \exists\, x \bullet \; x \in e \;\Rightarrow\; \phi$$

If the term e is not defined, then $x \in e$ is not valid, which in turn causes validity of the implication (independent from validity of ϕ). Hence, the whole formula is valid. Due to our definition, negation distributes over quantifiers as in classic predicate logic:

$$A \models_\eta \exists\, x \in e \bullet \phi \quad \text{holds iff} \quad A \models_\eta \neg\, \forall\, x \in e \bullet \neg\, \phi \quad \text{holds}$$

By the above classic truth semantics we can easily determine whether a repository is consistent w.r.t. a rule. Classic truth semantics is, however, insufficient for describing inconsistencies. Therefore, we calculate consistency reports.

[5]For convenience, we overload the symbol $\in$ to also operate on lists: $v \in xs$ means that the value v is a member of the list xs.

5.3 Generating Consistency Reports

In this section, we define a basic consistency checking algorithm. Fig. 5.4 shows the denotational semantics of our report generator $\mathcal{R}$. The function $\mathcal{R}_A[\![\phi]\!]\eta$ is defined by structural induction on a formula ϕ (like in classic boolean semantics as shown in the previous section) and creates a report for every subformula. As in the previous section, A denotes a first-order structure; η stands for the current variable assignment, which binds variables to values. Initially, $\mathcal{R}$ is applied to an empty variable assignment; i.e., $\mathcal{R}_A[\![\phi]\!]\emptyset$ returns the consistency report of the consistency rule ϕ. Next, we describe how $\mathcal{R}$ generates reports (we delay the description of auxiliary functions until Sect. 5.5).

For an atomic formula, we create a new report with only one diagnosis. The truth value of an atomic formula is computed as usual in partial predicate logic. Depending on its truth value the atomic formula itself is pushed into one of the sets $\mathcal{F}_t$ or $\mathcal{F}_f$. We store the complete atomic formula, in order to identify predicate symbols that occur more than once in a consistency rule. If any of the argument terms e_i is not defined, we also store the atomic formula $\downarrow(e_i)$ in the set $\mathcal{F}_f$, in order to indicate that this exception caused the inconsistency of the atomic formula.

For a conjunction $\phi \wedge \psi$, we retain the reports of subformulae that are responsible for the final truth value — this shortens our reports. If ϕ and ψ have the same truth value, we compute the cartesian product of their reports with help of the auxiliary function $\otimes$. Clearly, in this case both subformulae are responsible for the final truth value. Otherwise, we retain the report of the violated subformula only, because it is already sufficient for the violation of the conjunction. We have to generate a report also if a conjunction is fulfilled, because it may appear in a negated context. For disjunctions, we use a similar approach but join reports via $\oplus$, if both subformulae have the same truth value.

For a universally quantified formula $\forall\, x \in e \bullet \phi$, we first evaluate the sphere term e, giving a list of values. For each value v in this list, we compute ϕ's report w.r.t. the assignment extension $\eta \cup \{x \mapsto v\}$. If ϕ is violated for an assignment extension, then the boolean value of the corresponding report is False. In this case, we push the current variable binding $x \mapsto v$ into the variable assignment of each diagnosis in ϕ's report. Finally, $\overline{\oplus}$ joins the resulting reports in F. If F is empty, then ϕ is satisfied for each assignment extension. The set T contains reports of satisfied ϕ. These reports *lack* the new variable binding $x \mapsto v$, which is superfluous here, because ϕ is fulfilled for *every* $\eta \cup \{x \mapsto v\}$. In a consistency report, we understand omitted variables as universally quantified. We return the *minimized* report set T, in order to shorten reports. Minimizing the report set T means to generate a report containing those diagnoses that include the minimal number of atomic formulae. We retain the set T, because a fulfilled formula can become inconsistent by negation. Consider the formula $\neg\, \forall\, x \in e \bullet \phi$. Here, the universal quantifier behaves like an existential quantifier. If $\forall\, x \in e \bullet \phi$ is fulfilled, then its negation is violated. But we do not know which of the values in the evaluated sphere e is responsible for the violation. In our approach, we blame those values that caused the "least violations;" i.e.,

$$
\begin{array}{lcll}
\mathcal{R}_A[\![p(e_1,\ldots,e_n)]\!]\eta & = & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{p(e_1,\ldots,e_n)\}, \emptyset)\}) & \\
 & & \quad \text{all } e_i^A \text{ are defined and } (e_1^A,\ldots,e_n^A) \in p^A & \\
 & & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{p(e_1,\ldots,e_n)\} \cup \mathit{undefs})\}) & \\
 & & \quad \text{otherwise} & \\
\quad \text{where } e_i^A & = & s_i^A \quad \text{if } e_i \equiv s_i & \\
 & & \mathcal{V}_A[\![e_i]\!]\eta \quad \text{otherwise} & \\
\quad \mathit{undefs} & = & \{\downarrow(e_i) \mid e_i^A \text{ is not defined}\} &
\end{array}
$$

$$
\mathcal{R}_A[\![\neg\phi]\!]\eta = \overline{\mathsf{flip}}(\mathcal{R}_A[\![\phi]\!]\eta)
$$

$$
\begin{array}{lcll}
\mathcal{R}_A[\![\phi \wedge \psi]\!]\eta & = & r_\phi \otimes r_\psi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{fst}(r_\psi) \\
 & & r_\phi & \text{else if } \mathsf{fst}(r_\phi) = \mathsf{False} \\
 & & r_\psi & \text{else if } \mathsf{fst}(r_\psi) = \mathsf{False} \\
\quad \text{where } r_\phi & = & \mathcal{R}_A[\![\phi]\!]\eta & \\
\quad r_\psi & = & \mathcal{R}_A[\![\psi]\!]\eta &
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{R}_A[\![\phi \vee \psi]\!]\eta & = & r_\phi \oplus r_\psi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{fst}(r_\psi) \\
 & & r_\phi & \text{else if } \mathsf{fst}(r_\phi) = \mathsf{True} \\
 & & r_\psi & \text{else if } \mathsf{fst}(r_\psi) = \mathsf{True} \\
\quad \text{where } r_\phi & = & \mathcal{R}_A[\![\phi]\!]\eta & \\
\quad r_\psi & = & \mathcal{R}_A[\![\psi]\!]\eta &
\end{array}
$$

$$
\mathcal{R}_A[\![\phi \Rightarrow \psi]\!]\eta = \mathcal{R}_A[\![\neg\phi \vee \psi]\!]\eta
$$

$$
\begin{array}{lcll}
\mathcal{R}_A[\![\forall\, x \in e \bullet \phi]\!]\eta & = & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{\downarrow(e)\})\}) & \text{if } \mathcal{V}_A[\![e]\!]\eta \text{ is not defined} \\
 & & \overline{\oplus}(F) & \text{else if } F \neq \emptyset \\
 & & \overline{\mathsf{min}}(T) & \text{else if } T \neq \emptyset \\
 & & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{\mathtt{null}(e)\}, \emptyset)\}) & \text{otherwise} \\
\quad \text{where } rs & = & \{(v, \mathcal{R}_A[\![\phi]\!](\eta \cup \{x \mapsto v\})) \mid v \in \mathcal{V}_A[\![e]\!]\eta\} & \\
\quad F & = & \{\mathsf{push}(x \mapsto v, r) \mid (v, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\} & \\
\quad T & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} &
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{R}_A[\![\exists\, x \in e \bullet \phi]\!]\eta & = & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \emptyset, \{\downarrow(e)\})\}) & \text{if } \mathcal{V}_A[\![e]\!]\eta \text{ is not defined} \\
 & & \overline{\oplus}(T) & \text{else if } T \neq \emptyset \\
 & & \overline{\mathsf{min}}(F) & \text{else if } F \neq \emptyset \\
 & & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \{\mathtt{null}(e)\}, \emptyset)\}) & \text{otherwise} \\
\quad \text{where } rs & = & \{(v, \mathcal{R}_A[\![\phi]\!](\eta \cup \{x \mapsto v\})) \mid v \in \mathcal{V}_A[\![e]\!]\eta\} & \\
\quad T & = & \{\mathsf{push}(x \mapsto v, r) \mid (v, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} & \\
\quad F & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\} &
\end{array}
$$

Figure 5.4: A basic report generation algorithm (for auxiliary functions see Fig. 5.10 (pg. 63))

their diagnoses contain less atomic formulae than other diagnoses. Alternatively, we could retain all diagnoses of the reports in T (by employing $\overline{\oplus}$). This would, however, lead to large and incomprehensible reports. If the sphere of a universally quantified formula is empty, we generate a new diagnosis indicating that emptiness of the sphere e is responsible for fulfilling the formula. The atomic formula $\texttt{null}(e)$ means that the sphere e is empty, where the predicate symbol $\texttt{null} : [\alpha] \to \mathsf{Bool}$ is defined in the language Prelude. If the sphere term e is not defined, we immediately return a report including one diagnosis. This diagnosis indicates that it is undefinedness of e, which caused an inconsistency.

For existentially quantified formulae, we use a similar approach. The rôles of T and F are, however, "reversed," because an existentially quantified formula is satisfied already, if its subformula is fulfilled for *one* assignment extension. If an existentially quantified formula is violated, we do not know which values are responsible for its violation; hence, we minimize the report set F. Notice that an existentially quantified formula is fulfilled, if its sphere term e is not defined. In this case, the returned report indicates that undefinedness of the sphere e is responsible for this fact.

Notice that, by definition, negation distributes over quantifiers:

$$\mathcal{R}_A[\![\exists\, x \in e \bullet \phi]\!]\eta \quad = \quad \mathcal{R}_A[\![\neg\, \forall\, x \in e \bullet \neg\, \phi]\!]\eta$$

Of course, report generation is sound, i.e., a generated consistency report satisfies the following properties:

- A False consistency report contains diagnoses indicating inconsistencies only. A True consistency report contains diagnoses indicating consistencies only.
- In each diagnosis, all predicates in the set $\mathcal{F}_t$ are fulfilled and all predicates in the set $\mathcal{F}_f$ are violated.
- A consistency report indicates inconsistencies for a rule ϕ, if and only if ϕ is not valid.

The following theorems formalize the above properties. We prove them in App. C.

Theorem 5.1 (Generated consistency reports are sound) Let ϕ be a formula, A a first-order structure, η a variable assignment, and ds a set of diagnoses. Then we have:

$$\begin{array}{l}(\mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{False}, ds) \;\Rightarrow\; ds \neq \emptyset \;\wedge\; d = (\mathsf{IC}, _, _, _) \text{ for every } d \in ds) \quad \wedge \\ (\mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{True}, ds) \;\Rightarrow\; ds \neq \emptyset \;\wedge\; d = (\mathsf{C}, _, _, _) \text{ for every } d \in ds)\end{array}$$

Proof: The proof proceeds by straightforward induction on the structure of the formula ϕ; see App. C, Proof C.1. □

Theorem 5.2 (Reasons for inconsistencies are sound) Let ϕ be a formula, A a first-order structure, η a variable assignment, ds a diagnoses set, and ps_t and ps_f sets of atomic formulae. Then we have:

$$\begin{array}{l} \mathcal{R}'_A[\![\phi]\!]\eta = (_, ds) \wedge \\ (_, \eta, ps_t, ps_f) \in ds \end{array} \quad \Longrightarrow \quad \begin{array}{l} A \models_\eta \phi' \text{ for every } \phi' \text{ in } ps_t \wedge \\ A \models_\eta \neg\phi' \text{ for every } \phi' \text{ in } ps_f \end{array}$$

The report generation function $\mathcal{R}'$ deviates from $\mathcal{R}$ as follows: For an atomic formula, we push the complete variable assignment into the resulting diagnosis. For quantified formulae, we do not push bindings into the diagnoses. This is necessary, in order to ensure that η contains all free variables of the predicate sets ps_t and ps_f.
Proof: The proof proceeds by straightforward induction on the structure of the formula ϕ; see App. C, Proof C.2. □

Theorem 5.3 (Consistency reports indicate real inconsistencies) Let ϕ be a formula, A a first-order structure, and η a variable assignment. Then we have:

$$\text{not } A \models_\eta \phi \quad \Longleftrightarrow \quad \exists\, ds \in \wp(\mathbb{D}) \;\bullet\; \mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{False}, ds)$$

Proof: The proof proceeds by straightforward induction on the structure of the formula ϕ; see App. C, Proof C.3. □

5.4 Examples

In this section, we illustrate report generation for rule ϕ_1 at state 4 of our example repository. The final report can be found in Fig. 5.2 (pg. 47).

Initially, our report generator $\mathcal{R}$ is applied to ϕ_1 and an empty variable assignment. The quantifier sphere of t evaluates to the states $[1, 2, 3, 4]$. For each state $v \in [1, 2, 3, 4]$, we extend the variable assignment and compute the report of the subformula:

$$\mathcal{R}_A \left[\!\!\left[\begin{array}{l} \forall\, x \in \mathtt{repDs}(t) \bullet \forall\, k \in \mathtt{refs}(x) \bullet \\ \exists\, d \in \mathtt{concatMap}(\mathtt{kDefs}, \mathtt{repResDs}(t)) \bullet \exists\, m \in \mathtt{repManDs}(t) \bullet \\ k = \mathtt{key}(d) \;\wedge\; \mathtt{dId}(m) = \mathtt{kId}(d) \;\wedge\; \mathtt{kind}(m) = \mathtt{kKind}(d) \end{array} \right]\!\!\right] \{t \mapsto v\}$$

We obtain the following reports:

$$\begin{array}{ll}
\{t \mapsto 1\} & (\mathsf{True}, \emptyset) \\
\{t \mapsto 2\} & \left(\mathsf{False}, \left\{ \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}\}, \\ \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix} \right\}\right) \\
\{t \mapsto 3\} & \left(\mathsf{False}, \left\{ \begin{array}{l} \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}\}, \\ \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix}, \\ \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA2}, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\}, \\ \emptyset, \{k = \mathtt{key}(d), \mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix} \end{array} \right\}\right) \\
\{t \mapsto 4\} & \left(\mathsf{False}, \left\{ \begin{array}{l} \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}\}, \\ \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix}, \\ \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA2}, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\}, \\ \emptyset, \{k = \mathtt{key}(d), \mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix} \end{array} \right\}\right)
\end{array}$$

$$\begin{aligned}
\mathcal{V}_A[\![\texttt{repDs}(t)]\!]\{t \mapsto 4\} &= \begin{bmatrix} \{\texttt{dId} = \mathsf{doc1.txt}, \texttt{dState} = 1\}, \\ \{\texttt{dId} = \mathsf{keys.xml}, \texttt{dState} = 1\}, \\ \{\texttt{dId} = \mathsf{doc2.txt}, \texttt{dState} = 3\}, \\ \{\texttt{dId} = \mathsf{man1.xml}, \texttt{dState} = 2\}, \\ \{\texttt{dId} = \mathsf{man2.xml}, \texttt{dState} = 4\}, \\ \{\texttt{dId} = \mathsf{man3.xml}, \texttt{dState} = 4\}, \\ \{\texttt{dId} = \mathsf{man4.xml}, \texttt{dState} = 4\} \end{bmatrix} \\
\mathcal{V}_A[\![\texttt{refs}(x)]\!]\{x \mapsto \{\texttt{dId} = \mathsf{doc1.txt}, \texttt{dState} = 1\}\} & \\
&= [\mathsf{kaA3}] \\
\mathcal{V}_A[\![\texttt{refs}(x)]\!]\{x \mapsto \{\texttt{dId} = \mathsf{doc2.txt}, \texttt{dState} = 3\}\} & \\
&= [\mathsf{kaA2}] \\
\mathcal{V}_A[\![\texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t))]\!]\{t \mapsto 4\} & \\
&= \left[\begin{Bmatrix} \texttt{key} = \mathsf{kaA3}, \texttt{kId} = \mathsf{man1.xml}, \\ \texttt{kKind} = \mathsf{technical\ M.} \end{Bmatrix} \right] \\
\mathcal{V}_A[\![\texttt{repManDs}(t)]\!]\{t \mapsto 4\} &= \begin{bmatrix} \{\texttt{dId} = \mathsf{man1.xml}, \texttt{dState} = 2, \texttt{kind} = \mathsf{field\ M.}\}, \\ \{\texttt{dId} = \mathsf{man2.xml}, \texttt{dState} = 4, \texttt{kind} = \mathsf{field\ M.}\}, \\ \{\texttt{dId} = \mathsf{man3.xml}, \texttt{dState} = 4, \texttt{kind} = \mathsf{field\ M.}\}, \\ \{\texttt{dId} = \mathsf{man4.xml}, \texttt{dState} = 4, \texttt{kind} = \mathsf{field\ M.}\} \end{bmatrix}
\end{aligned}$$

Figure 5.5: Example quantifier spheres

For each False report above, we push the corresponding binding for t into the report's diagnoses and join the resulting reports by $\overline{\oplus}$. That way we obtain the final report shown in Fig. 5.2.

Next, we detail how the report for $\{t \mapsto 4\}$ is generated. Generation of reports for the other bindings to t proceeds similarly; in fact, these bindings are easier to process, because up to state 3 there exists only one manual in the repository. Fig. 5.5 shows some quantifier spheres, where we consider non-empty spheres only. For convenience, we restrict variable assignments to the sphere term's free variables. In the following, we concentrate on the existential quantifier for m for the following variable assignments:

$$\begin{aligned}
\eta_{\mathsf{doc1}} &= \begin{Bmatrix} t \mapsto 4, x \mapsto \{\texttt{dId} = \mathsf{doc1.txt}, \texttt{dState} = 1\}, k \mapsto \mathsf{kaA3}, \\ d \mapsto \{\texttt{key} = \mathsf{kaA3}, \texttt{kId} = \mathsf{man1.xml}, \texttt{kKind} = \mathsf{technical\ M.}\} \end{Bmatrix} \\
\eta_{\mathsf{doc2}} &= \begin{Bmatrix} t \mapsto 4, x \mapsto \{\texttt{dId} = \mathsf{doc2.txt}, \texttt{dState} = 3\}, k \mapsto \mathsf{kaA2}, \\ d \mapsto \{\texttt{key} = \mathsf{kaA3}, \texttt{kId} = \mathsf{man1.xml}, \texttt{kKind} = \mathsf{technical\ M.}\} \end{Bmatrix}
\end{aligned}$$

First, we consider report generation for η_{doc1}. The existential quantifier extends η_{doc1} consecutively by bindings to one of the manuals man1.xml through man4.xml. We generate four reports (the conjunction $k = \texttt{key}(d) \wedge \texttt{dId}(m) = \texttt{kId}(d) \wedge \texttt{kind}(m) = \texttt{kKind}(d)$ is abbreviated by $\phi_\wedge$):

$$\begin{aligned}
&\mathcal{R}_A[\![\phi_\wedge]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\texttt{dId} = \mathsf{man1.xml}, \texttt{dState} = 2, \texttt{kind} = \mathsf{field\ M.}\}\} \\
&\mathcal{R}_A[\![\phi_\wedge]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\texttt{dId} = \mathsf{man2.xml}, \texttt{dState} = 4, \texttt{kind} = \mathsf{field\ M.}\}\} \\
&\mathcal{R}_A[\![\phi_\wedge]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\texttt{dId} = \mathsf{man3.xml}, \texttt{dState} = 4, \texttt{kind} = \mathsf{field\ M.}\}\} \\
&\mathcal{R}_A[\![\phi_\wedge]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\texttt{dId} = \mathsf{man4.xml}, \texttt{dState} = 4, \texttt{kind} = \mathsf{field\ M.}\}\}
\end{aligned}$$

For the assignment extension $\eta_{\mathsf{doc1}} \cup \{m \mapsto \{\texttt{dId} = \mathsf{man1.xml}, \ldots\}\}$, only the subformula $\texttt{kind}(m) = \texttt{kKind}(d)$ is inconsistent. Thus, we obtain:

$$\begin{aligned}
&\mathcal{R}_A[\![\phi_\wedge]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\texttt{dId} = \mathsf{man1.xml}, \ldots\}\} \\
&\quad = \big(\mathsf{False}, \big\{\ \big(\mathsf{IC}, \emptyset, \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\}\big)\ \big\}\big)
\end{aligned}$$

In contrast, for the assignment extension $\eta_{\mathsf{doc1}} \cup \{m \mapsto \{\mathtt{dId} = \mathsf{man2.xml}, \ldots\}\}$ two subformulae are inconsistent:

$$\begin{aligned}&\mathcal{R}_A[\![\mathtt{dId}(m) = \mathtt{kId}(d)]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\mathtt{dId} = \mathsf{man2.xml}, \ldots\}\} \\ &\quad = \big(\mathsf{False}, \{\ (\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{dId}(m) = \mathtt{kId}(d)\})\ \}\big)\end{aligned}$$

$$\begin{aligned}&\mathcal{R}_A[\![\mathtt{kind}(m) = \mathtt{kKind}(d)]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\mathtt{dId} = \mathsf{man2.xml}, \ldots\}\} \\ &\quad = \big(\mathsf{False}, \{\ (\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\})\ \}\big)\end{aligned}$$

We compute the cartesian product of the above reports and obtain:

$$\begin{aligned}&\mathcal{R}_A[\![\phi_\wedge]\!] \quad \eta_{\mathsf{doc1}} \cup \{m \mapsto \{\mathtt{dId} = \mathsf{man2.xml}, \ldots\}\} \\ &\quad = \big(\mathsf{False}, \{\ (\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d), \mathtt{dId}(m) = \mathtt{kId}(d)\})\ \}\big)\end{aligned}$$

Report generation for man3.xml and man4.xml proceeds similarly and also results in the report above.

We return to the existential quantifier for m. For each assignment extension, we obtain a report indicating inconsistencies, thus:

$$F = \left\{ \begin{array}{l} \big(\mathsf{False}, \{\ (\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\})\ \}\big)\,, \\ \big(\mathsf{False}, \{\ (\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d), \mathtt{dId}(m) = \mathtt{kId}(d)\})\ \}\big) \end{array} \right\}$$

Clearly, the diagnosis of the first report contains less atomic formulae than the second report's diagnosis. Thus, minimizing F drops the second report:

$$\mathcal{R}_A[\![\exists\, m \in \mathtt{repManDs}(t) \bullet \phi_\wedge]\!]\eta_{\mathsf{doc1}} \;=\; (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\})\})$$

Generating the report for the existential quantifier for m w.r.t. the variable assignment η_{doc2} proceeds similarly to the above generation. Here, however, also the atomic formula $k = \mathtt{key}(d)$ is violated for all bindings to m:

$$\begin{aligned}&\mathcal{R}_A[\![\exists\, m \in \mathtt{repManDs}(t) \bullet \phi_\wedge]\!]\eta_{\mathsf{doc2}} \\ &\quad = \big(\mathsf{False}, \{\ (\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d), k = \mathtt{key}(d)\})\ \}\big)\end{aligned}$$

The existential quantifier for d does not change the above reports. The universal quantifier for k pushes the binding $k \mapsto \mathsf{kaA3}$ into the report for η_{doc1} and the binding $k \mapsto \mathsf{kaA2}$ into the report for η_{doc2}.

We return to the universal quantifier for x, which is violated for the assignments $\{t \mapsto 4, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \ldots\}\}$ and $\{t \mapsto 4, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \ldots\}\}$ (for other assignments the formula is fulfilled), thus:

$$F = \left\{ \begin{array}{l} \left(\mathsf{False}, \left\{ \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}\}, \\ \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix} \right\}\right), \\ \left(\mathsf{False}, \left\{ \begin{pmatrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA2}, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\}, \\ \emptyset, \{k = \mathtt{key}(d), \mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix} \right\}\right) \end{array} \right\}$$

We join the above reports and obtain the report for the assignment $\{t \mapsto 4\}$.

Next, we introduce some formal structures and show how our logic is related to classic predicate logic. Moreover, we define the calculation of values. Since these technical details are not essential for the further understanding of this thesis, readers might want to skip the following sections and to continue with the chapter summary (Sect. 5.7, pg. 64).

5.5 Formal Structures for Report Generation

In this section, we define the formal structures needed for report generation and value computation (defined in Sect. 5.6), namely:

- A *first-order structure*, which interprets types, function symbols, and predicate symbols. In addition, we show laws for function symbols in the base signature (see Fig. 4.5, pg. 31).
- A *value* as our basic semantic object.
- A *variable assignment*, which binds variables to values.
- A *consistency report*, which pinpoints inconsistencies.

Finally in this section, we define auxiliary functions needed for report generation. We define our formal structures in a way that retains compatibility with already existing logic tools, e.g., theorem provers. In particular, we simulate parametric polymorphism and subtyping.

We interpret formulae in the *current* first-order structure, which we call Σ-algebra. The Σ-algebra changes over time by the check-ins to the repository, see below. As in many-sorted predicate logic, a Σ-algebra is defined relative to the signature Σ. A Σ-*algebra* A contains a universe $\mathbf{U}^A$ (which includes a domain τ^A for each type τ that can be constructed), a set $\mathbf{P}^A$ (which contains interpretations of predicate symbols), and a set $\mathbf{T}^A$ (which contains interpretations of function symbols). For simplicity, we model domains by sets.

Function symbols are interpreted by *partial* functions over appropriate domains; predicate symbols are interpreted by relations over appropriate domains. Recall that relations are total by definition. We interpret a function symbol $f : \tau'$ by a *set* of functions. Basically, this is, because in many-sorted logic we have simple sorts only; hence, we must simulate parametric polymorphism. If f is polymorphic, we have to provide a function for every possible monomorphic instantiation $\theta\tau'$ of f's type τ'. Due to subtyping, we have to take every possible subtype τ of $\theta\tau'$ into account. Predicate symbols are interpreted similarly by relations. We make the following coherence assumptions: A function symbol $f : \tau$ is not "re-defined" at subtypes of τ. Similarly, a predicate symbol $p : \tau_p$ is not "re-defined" at subtypes of τ_p.

Below, we use abbreviated syntax: τ_p^A stands for the cartesian product $\tau_1^A \times \ldots \times \tau_n^A$ interpreting the predicate type $\tau_p = \tau_1 \times \ldots \times \tau_n \to \mathsf{Bool}$; τ^A stands for the set of all partial functions from the cartesian product $\tau_1^A \times \ldots \times \tau_n^A$ to the domain τ_g^A interpreting the function type $\tau = \tau_1 \times \ldots \times \tau_n \to \tau_g$.

Definition 5.4 (Σ-algebra) Given a signature $\Sigma = (\mathbf{T}, \mathbf{S}, \prod, \mathbf{K}, \sum, \mathbf{P}, \mathbf{F})$, we define a Σ-*algebra* as a triple $A = (\mathbf{U}^A, \mathbf{P}^A, \mathbf{F}^A)$, where:

- The universe $\mathbf{U}^A$ contains a domain τ^A for each monomorphic type $\tau \in \Omega(\mathbf{T})$, i.e., $\mathsf{tv}(\tau) = \emptyset$.
  - The domain of an atomic type is a set of discrete atomic values.

- The domain of a record type is the cartesian product of the domains of its label result types.
- The domain of a variant type is the union of the domains of its constructor argument types.
- The domain of a function type is the set of all partial functions from the argument type domains to the result type domain.
- The domain of a predicate type is the set of all relations over the argument type domains.

- The set $\mathbf{P}^A = \{\tilde{p}^A \mid p \in \mathbf{P}\}$ contains interpretations of predicate symbols. Each $\tilde{p}^A$ is a non-empty set of relations $p^A_{\tau_p^A}$ over the cartesian product τ_p^A iff $p : \tau'_p$, $\tau_p \leq \theta\tau'_p$, and $\mathsf{tv}(\tau_p) = \emptyset$.
- The set $\mathbf{F}^A = \{\tilde{f}^A \mid f \in \mathbf{F}\}$ contains interpretations of function symbols. Each $\tilde{f}^A$ is a non-empty set of partial functions $f^A_{\tau^A} : \tau^A$ iff $f : \tau'$, $\tau \leq \theta\tau'$, and $\mathsf{tv}(\tau) = \emptyset$.

We let $\mathbb{A}$ denote the set of all Σ-algebras. □

Although a Σ-algebra consists of infinite sets, we can represent these sets finitely. Relations and functions can be represented by computations in a programming language supporting polymorphism and subtyping. Subtyping can be simulated by explicit coercion functions, which convert a value of type τ to a value of type τ' if $\tau \leq \tau'$. Coercion functions can be generated, which guarantees our coherence assumptions above. Indeed, this is actually done in our implementation, because Haskell lacks subtyping (see App. B). Finally, the universe $\mathbf{U}^A$ can be left implicit.

We interpret formulae w.r.t. a Σ-algebra A that evolves over time. Formally, an abstract state machine (ASM) [Gur00] models the changes to the partial algebra A. These changes are induced by the check-ins to the repository. Our ASM is defined by the initial algebra A_0 and a transition function $\delta_{\mathbb{A}} : \mathbb{A} \to \mathbb{A}$, which computes a new algebra A_{i+1} from a previous algebra A_i.[6] In each algebra, we find functions and domains as illustrated in Fig. 5.6.

In order to model repository behavior, we extend the base signature from Fig. 4.5 (pg. 31) by two functions add_i (modelling documents added by a check-in) and del_i (modelling documents deleted by a check-in). Formally, we assume the following domains: String denotes all possible strings, State denotes all possible states (i.e., natural numbers $\mathbb{N}$),[7] Doc $=$ String $\times$ State denotes the name and the check-in state of a document, and $[\alpha]$ denotes the set of all sequences of the domain α. We borrow the symbol $\nrightarrow$ from the specification language Z [Spi89] to denote partial functions. An algebra defines at least the following functions:

[6] Gurevich uses the symbol τ_A for the algebra transition function. We use $\delta_{\mathbb{A}}$, in order to avoid confusions with our notation of types τ.

[7] We use an *infinite* domain, in order to facilitate constant domains, which keeps our ASM simple.

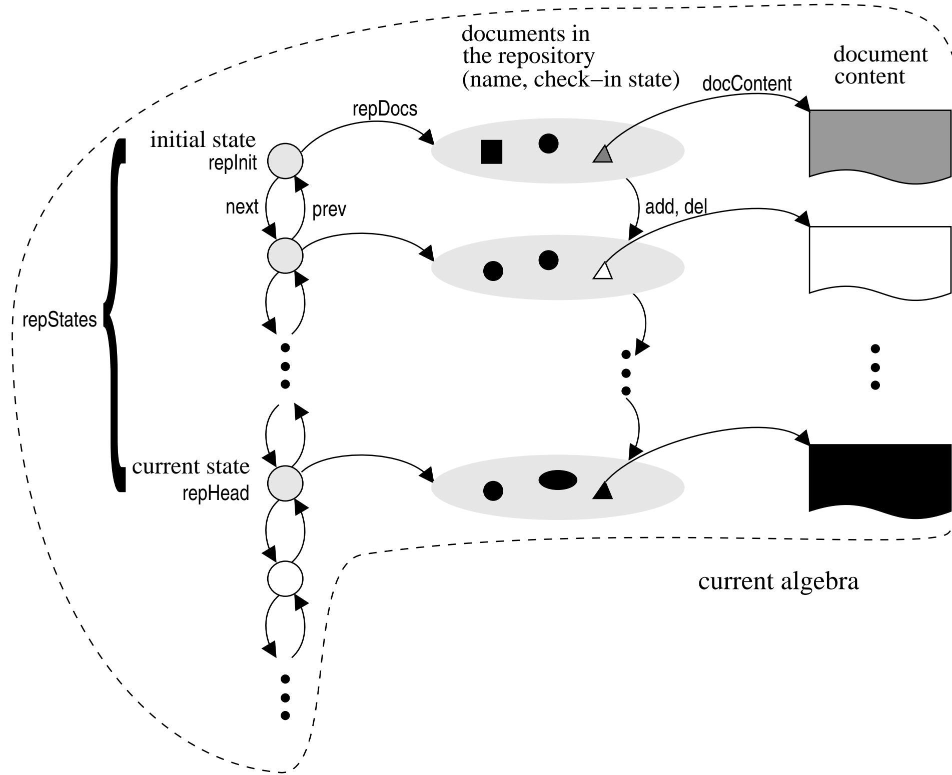


Figure 5.6: Temporally evolving algebra

`repStates`	:	[State]
`repHead`	:	State
`repInit`	:	State
`next`	:	State $\to$ State
`prev`	:	State$\nrightarrow$ State
`repDocs`	:	State$\nrightarrow$ [Doc]
`docContent`	:	Doc$\nrightarrow$ `String`
add_i	:	[`String` $\times$ `String`]
del_i	:	[`String`]

We assume the natural $\leq$ ordering for natural numbers, i.e., we facilitate linear discrete time. The predicate symbol $=$ is interpreted by structural equality. By `next` we move one state forward; by `prev` we move one state backward. The initial state is given by `repInit`, the current state is given by `repHead`. Notice that `prev` is not defined for the initial repository state. The function `repStates` returns all states between the initial state and the current state, i.e., those states for which there exists a repository. For each state t up to the current state, the function `repDocs`(t) returns the names and check-in states of the documents in the repository. For states larger than the current state, the function `repDocs` is undefined, i.e., there does not yet exist a corresponding check-in.[8] For each document d in the repository, the function `docContent`(d)

[8]Alternatively, we could evaluate `repDocs` to an empty list. Then we could, however, not distinguish an empty repository from a non-existent repository.

returns the document content. Recall that we identify documents by their name and check-in state.

A check-in to the repository at state i is modelled by the functions add_{i+1} (returning the document names and contents added by some author) and del_{i+1} (returning the names of documents deleted by some author). In other words, add_{i+1} and del_{i+1} "build" the algebra A_{i+1} from A_i. For simplicity, we model document changes by add and del. Each check-in enhances the current algebra A_i as follows:

- The repository changes as determined by add_{i+1} and del_{i+1}.
- The function `repHead` is updated to return the new current repository state `next(repHead)`.
- The function `repStates` now also incorporates the new repository state.
- For the new state, the function `repDocs` returns the names and check-in states of all documents in the repository.
- For added documents, the function `docContent` returns their content.

The new algebra A_{i+1} includes the old algebra A_i; A_{i+1} deviates from A_i in the current state, `repStates`, and `repHead` only. This important property guarantees *referential transparency* of all functions and predicates that do not use `repStates` or `repHead` directly or indirectly.[9] Referential transparency is a prerequisite for efficient consistency checking as described in Chapter 6.

We now define an ASM that specifies how our algebra changes. The functions `repInit`, `next`, and `prev` do not change their behavior over time — they are static. In contrast, the functions `repStates`, `repHead`, `repDocs`, and `docContent` change their behavior over time — they are dynamic. In the initial algebra A_0, all functions are undefined. We regard the functions add_{i+1} and del_{i+1} as inputs to the transition function $\delta_{\mathbb{A}}$ — we do not control them. They are updated by the environment, i.e., authors who check in documents to the repository. In addition, we need the domain Mode $=$ {InitialCheckin, Checkin, Wait}, where InitialCheckin stands for the initial check-in to the repository, Checkin stands for another check-in to the repository, and Wait means that the consistency checker waits for a repository check-in. The function `currentMode` : Mode models interaction with the repository. We assume that the repository sets the function `currentMode` to InitialCheckin, if the author performs the first check-in to a newly created repository; the repository sets `currentMode` to Checkin, if the author performs another check-in. At each of these modes, the ASM program $\delta_{\mathbb{A}}$ reads the inputs add_{i+1} and del_{i+1} and updates the algebra A_i appropriately (see Fig. 5.7, for brevity we omit the superscript A for functions and relations).

By our ASM we interpret function symbols and predicate symbols always w.r.t. the current algebra A_i computed by repeatedly applying the algebra transition function $\delta_{\mathbb{A}}$ to the initial algebra A_0, i.e., $A_i = \delta_{\mathbb{A}}^i(A_0)$.

[9] Roughly speaking, referential transparency means that the result of a function or relation depends on its parameters only.

```
par
 if currentMode = InitialCheckin then
  null                 :=  {[]}
  next(t)              :=  min{t' ∈ State | t ≤ t'  ∧  t ≠ t'}
  prev(t)              :=  max{t' ∈ State | t' ≤ t  ∧  t ≠ t'}
  repInit              :=  1
  repHead              :=  1
  repStates            :=  [repHead]
  repDocs(repHead)     :=  [(n, repHead) | (n, _) ∈ add_{i+1}]
  if (n, c) ∈ add_{i+1} then
   docContent(n, repHead) := c
  endif
  currentMode          :=  Wait
 endif
 if currentMode = Checkin then
  repHead              :=  next(repHead)
  repStates            :=  repStates ++ [repHead]
  repDocs(repHead)     :=  [(n, t) | (n, t) ∈ repDocs(prev(repHead)) ∧
                                     n ∉ del_{i+1} ∧ (n, _) ∉ add_{i+1}] ++
                           [(n, repHead) | (n, _) ∈ add_{i+1}]
  if (n, c) ∈ add_{i+1} then
   docContent(n, repHead) := c
  endif
  currentMode          :=  Wait
 endif
```

Figure 5.7: Algebra transition $\delta_{\mathbb{A}}$ (++ denotes list concatenation)

Obviously, our ASM preserves the invariants shown in Fig. 5.8. The functions `repHead` and `repStates` are not referentially transparent. Then each symbol is referentially transparent, if it is not defined in terms of `repHead` or `repStates`. Consequently, two algebras A_i and $\delta_{\mathbb{A}}(A_i)$ coincide in all terms that are defined in A_i and contain referentially transparent symbols only.

Values are the basic semantic objects in the domains of a universe. A value can be either an atomic value, a record value, or a variant value. For an easier computation of values, they carry their type constructors. In addition, a variant value includes its variant constructor. A record value carries bindings of record labels to values. A variant value carries argument values of its variant constructor. For brevity, we omit type constructors and record constructors, if they are clear from the context.

Definition 5.5 (Value) Given a signature $\Sigma = (\mathbf{T}, \mathbf{S}, \prod, \mathbf{K}, \sum, \mathbf{P}, \mathbf{F})$, we define a *value* v^t (where $t \in \mathbf{T}$ is a type constructor) inductively as follows:

- If $A \in \mathbf{T}$ is an atomic type constructor, then v^A is an atomic value; v^{Top} and v^{State} are atomic values, too.
- If $V \in \mathbf{T}$ is a variant type constructor, $\widehat{V} = \{\overline{k_j}\}$ is its constructor closure, and $v_1^{t_1}, \ldots, v_{n_j}^{t_{n_j}}$ are values of appropriate types (constructed by type constructors t_1 through t_{n_j}), then $\mathsf{Var}\ k_j\ (v_1, \ldots, v_{n_j})^V$ is a variant value.

next and prev are injective inverse functions, next is total
$\texttt{next} \,.\, \texttt{prev} \subset \texttt{id}$
$\texttt{repInit} \neq t \Rightarrow \texttt{prev}(t)$ is defined
$\texttt{repInit} \neq t \Rightarrow \texttt{next}(\texttt{prev}(t)) = t$
$\texttt{prev} \,.\, \texttt{next} = \texttt{id}$
repStates returns all states between repInit and repHead
$\texttt{repInit} \leq t \;\wedge\; t \leq \texttt{repHead} \Rightarrow t \in \texttt{repStates}$
$\texttt{repInit} \neq t \;\wedge\; t \in \texttt{repStates} \Rightarrow \texttt{prev}(t) \in \texttt{repStates}$
$\texttt{repHead} \neq t \;\wedge\; t \in \texttt{repStates} \Rightarrow \texttt{next}(t) \in \texttt{repStates}$
for each state in repStates we find the current documents and their contents
$t \in \texttt{repStates} \Rightarrow \texttt{repDocs}(t)$ is defined
$d \in \texttt{repDocs}(t) \Rightarrow \texttt{docContent}(d)$ is defined
repDocs and docContent respect the check-ins to the repository
$\mathsf{add}_i = (n, c) \Rightarrow (n, i) \in \texttt{repDocs}(i)$
$\mathsf{del}_i = n \Rightarrow (n, i) \notin \texttt{repDocs}(i)$
$\mathsf{add}_i = (n, c) \Rightarrow \texttt{docContent}(n, i) = c$
$\mathsf{del}_i = n \Rightarrow \texttt{docContent}(n, i)$ is not defined
if defined, repDocs and docContent are referentially transparent over time
$\texttt{repDocs}^{A_t}(t) = ds \;\wedge\; t \leq t' \Rightarrow \texttt{repDocs}^{A_{t'}}(t) = ds$
$\texttt{docContent}^{A_t}(d) = c \;\wedge\; t \leq t' \Rightarrow \texttt{docContent}^{A_{t'}}(d) = c$

Figure 5.8: Repository invariants (free variables are universally quantified: $t, i \in \mathsf{State}$ denote repository states, $d = (n, i) \in \mathtt{Doc}$ denotes a document (name and check-in state), $c \in \mathtt{String}$ denotes document content, function composition is denoted by ".")

η	$::=$	$\{\overline{b_i}\}$	variable assignment
b	$::=$	$x \mapsto v$	binding $\mathcal{X} \times \mathbb{V}$ (x variable, v value)

Figure 5.9: Variable assignment $\mathbb{E}$

- If $R \in \mathbf{T}$ is a record type constructor, K_R is its record constructor, $\widehat{R} = \{\overline{l_j}\}$ is its label closure, and all $v_j^{t_j}$ are values of appropriate types (constructed by type constructors t_j), then $\mathsf{Rec}^{K_R}\left\{\overline{l_j = v_j^{t_j}}\right\}^R$ is a record value.

We let $\mathbb{V}$ denote the set of all values. □

A *variable assignment* binds variables to values. We define a variable assignment as a set of bindings (see Fig. 5.9). We denote the set of all variable assignments by $\mathbb{E}$; the set of all bindings is denoted $\mathcal{X} \times \mathbb{V}$. A variable assignment η induces a partial function $\eta' : \mathcal{X} \nrightarrow \mathbb{V}$ that maps a variable x to its current value v (this is more in the sense of classic predicate logic). The assignment extension $\eta \cup \{x \mapsto v\}$ can be seen as an update to the function η'. During report generation, a variable assignment is extended by quantifiers. For well-typed formulae, η is a *total* function.

A *consistency report* contains a boolean value and a set of diagnoses:

Definition 5.6 (Consistency Report) A *consistency report* (b, ds) contains a boolean value b (representing boolean truth semantics) and a set of diagnoses

$ds \subseteq \mathbb{D}$, where $\mathbb{D} = \{\mathsf{C}, \mathsf{IC}\} \times \mathbb{E} \times \wp(\mathcal{F}_{\text{at}}) \times \wp(\mathcal{F}_{\text{at}})$. A diagnosis $(c, \eta, \mathcal{F}_t, \mathcal{F}_f) \in \mathbb{D}$ contains a consistency flag c, a variable assignment η, and sets of atomic formulae $\mathcal{F}_t$ and $\mathcal{F}_f$. Therefore, $\mathbb{B} \times \wp(\mathbb{D})$ denotes the set of all consistency reports. □

The report generation function $\mathcal{R}$ has the type $\mathcal{R}_{.}\,[\![.]\!]_{.} : \mathbb{A} \times \mathcal{F} \times \mathbb{E} \to \mathbb{B} \times \wp(\mathbb{D})$. Notice that $\mathcal{R}$ is total, as is the validity relation $\models$.

Fig. 5.10 includes the definitions of auxiliary functions, needed for report generation. $\overline{\mathsf{flip}}$ inverts a report by inverting the boolean result and applying flip to each diagnosis, turning consistent diagnoses to inconsistent diagnoses and vice versa. Two reports can be combined by $\otimes$. The resulting report contains the conjunction of the boolean values and the cartesian product of the diagnoses sets, computed by applying join to each pair of diagnoses. Alternatively, we can join reports via $\oplus$. $\otimes$ and $\oplus$ $\mathsf{condense}$ the resulting diagnoses, such that diagnoses with equal variable assignments are joined. $\overline{\oplus}$ folds $\oplus$ over a non-empty report set. fold is an instance of (non-deterministically) folding a function over a list; we use fold with commutative associative function arguments only, which makes our applications well defined. From a set of diagnoses, the function min retains those diagnoses that contain the minimal sets of atomic formulae: min groups a set of diagnoses by their assignment and then returns for each group its greatest lower bound w.r.t. the partial diagnoses ordering $\sqsubseteq$. push extends the variable assignment of each diagnosis in a report by a variable binding. In addition, fst and snd decompose pairs as usual.

5.6 Computing Values

In this section, we define how values are computed, which is similar to value computation in partial predicate logic [Far01]. The evaluation function $\mathcal{V}_A[\![e]\!]\eta$ returns the value of a term e w.r.t. a Σ-algebra A and a variable assignment η. Due to support for partial function symbols, $\mathcal{V}$ has the type $\mathcal{V}_{.}\,[\![.]\!]_{.} : \mathbb{A} \times \mathcal{T} \times \mathbb{E} \nrightarrow \mathbb{V}$, i.e., value computation may fail, if a function is undefined for some arguments.

Fig. 5.11 shows the denotational semantics of $\mathcal{V}$. To simplify notation, we require that, in order to evaluate a term e, the values of all subterms of e must be defined; otherwise, the value of e is not defined (we adopt this behavior from [Far01]). We interpret a variable x by the current variable assignment η. Since rules are well typed, η contains a binding for x. In Fig. 5.11, we have annotated function symbols with their type τ. Since our type checker assigns *monomorphic* types, the interpretation of function symbols is uniquely determined. We distinguish the arguments of a symbol application syntactically: Symbol arguments are interpreted by the corresponding symbols, other arguments are further evaluated by $\mathcal{V}$. Record construction and deconstruction are straightforward, where record deconstruction determines the built-in semantics of record labels. Variant constructors build variant values carrying their constructor. Record labels and variant constructors are total functions by construction. Since variant values carry their constructors, case analysis becomes quite simple. From the evaluation of the case scrutinee e, we obtain

Invert a consistency report

$$\overline{\mathsf{flip}} \quad : \quad (\mathbb{B} \times \wp(\mathbb{D})) \rightarrow \mathbb{B} \times \wp(\mathbb{D})$$

$$\overline{\mathsf{flip}}\,((b, ds)) \quad = \quad (\neg b, \{\mathsf{flip}(d) \mid d \in ds\})$$

Invert a diagnosis

$$\mathsf{flip} \quad : \quad \mathbb{D} \rightarrow \mathbb{D}$$

$$\mathsf{flip}\left((\mathsf{C}, as, ps_t, ps_f)\right) \quad = \quad (\mathsf{IC}, as, ps_t, ps_f)$$

$$\mathsf{flip}\left((\mathsf{IC}, as, ps_t, ps_f)\right) \quad = \quad (\mathsf{C}, as, ps_t, ps_f)$$

Cartesian product of reports

$$\otimes \quad : \quad (\mathbb{B} \times \wp(\mathbb{D})) \times (\mathbb{B} \times \wp(\mathbb{D})) \rightarrow \mathbb{B} \times \wp(\mathbb{D})$$

$$(b, ds) \otimes (b', ds') \quad = \quad \left(b \wedge b', \mathsf{condense}\left\{\mathsf{join}(d, d') \mid d \in ds, d' \in ds'\right\}\right)$$

Join diagnoses

$$\mathsf{join} \quad : \quad \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$$

$$\mathsf{join}\left((c, as, ps_t, ps_f), (c', as', ps'_t, ps'_f)\right) \quad = \quad (c, as \cup as', ps_t \cup ps'_t, ps_f \cup ps'_f)$$

Condense diagnoses

$$\mathsf{condense} \quad : \quad \wp(\mathbb{D}) \rightarrow \wp(\mathbb{D})$$

$$\mathsf{condense}\left((c, \eta, ps_t, ps_f) \uplus ds\right) \quad = \quad \mathsf{fold}(\mathsf{join}, (c, \eta, ps_t, ps_f), ds_\eta) \cup \mathsf{condense}(ds \setminus ds_\eta)$$

where $ds_\eta = \{(c', \eta', ps'_t, ps'_f) \mid (c', \eta', ps'_t, ps'_f) \in ds \text{ and } \eta = \eta'\}$

Join reports

$$\oplus \quad : \quad (\mathbb{B} \times \wp(\mathbb{D})) \times (\mathbb{B} \times \wp(\mathbb{D})) \rightarrow \mathbb{B} \times \wp(\mathbb{D})$$

$$(b, ds) \oplus (b', ds') \quad = \quad \left(b \vee b', \mathsf{condense}\left(ds \cup ds'\right)\right)$$

Join a report set by folding ($\mathbf{S} \neq \emptyset$)

$$\overline{\oplus} \quad : \quad \wp(\mathbb{B} \times \wp(\mathbb{D})) \rightarrow \mathbb{B} \times \wp(\mathbb{D})$$

$$\overline{\oplus}(\mathbf{S}) \quad = \quad \mathsf{fold}(\oplus, s, \mathbf{S}'), \text{ where } \mathbf{S}' = \mathbf{S} \uplus \{s\}$$

Fold a set (f symmetric and associative)

$$\mathsf{fold} \quad : \quad \forall \alpha.(\alpha \times \alpha \rightarrow \alpha) \times \alpha \times \wp(\alpha) \rightarrow \alpha$$

$$\mathsf{fold}(f, x, \emptyset) \quad = \quad x$$

$$\mathsf{fold}(f, x, (\{x'\} \uplus xs)) \quad = \quad \mathsf{fold}(f, f(x, x'), xs)$$

Minimize reports

$$\overline{\mathsf{min}} \quad : \quad \wp(\mathbb{B} \times \wp(\mathbb{D})) \rightarrow \mathbb{B} \times \wp(\mathbb{D})$$

$$\overline{\mathsf{min}}\,((b, ds) \uplus rs) \quad = \quad (b, \mathsf{min}(ds \ \cup \ \bigcup\{ds \mid (_, ds) \in rs\}$$

Minimize diagnoses

$$\mathsf{min} \quad : \quad \wp(\mathbb{D}) \rightarrow \wp(\mathbb{D})$$

$$\mathsf{min}\left((c, \eta, ps_t, ps_f) \uplus ds\right) \quad = \quad \textstyle\bigsqcap_{\sqsubseteq} ds_\eta \ \cup \ \mathsf{min}(ds \setminus ds_\eta)$$

where $ds_\eta = \{(c', \eta', ps'_t, ps'_f) \mid (c', \eta', ps'_t, ps'_f) \in ds \text{ and } \eta = \eta'\}$

Partial order for diagnoses

$$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$$

$$(c, \eta, ps_t, ps_f) \sqsubseteq (c', \eta', ps'_t, ps'_f) \quad \Leftrightarrow \quad c = c' \wedge \eta = \eta' \wedge ps_t \subseteq ps'_t \wedge ps_f \subseteq ps'_f$$

Push an assignment into a report

$$\mathsf{push} \quad : \quad (\mathcal{X} \times \mathbb{V}) \times (\mathbb{B} \times \wp(\mathbb{D})) \rightarrow \mathbb{B} \times \wp(\mathbb{D})$$

$$\mathsf{push}\,(x \mapsto v, (b, ds)) \quad = \quad \left(b, \left\{\begin{array}{l} (c, \{x \mapsto v\} \cup as, ps_t, ps_f) \\ \mid (c, as, ps_t, ps_f) \in ds \end{array}\right\}\right)$$

Figure 5.10: Auxiliary functions for report generation

$$
\begin{array}{lcl}
\mathcal{V}_A[\![x]\!]\eta & = & \eta(x) \\
\mathcal{V}_A\left[\!\left[l_j(K_R\{\overline{l_i = e_i}\})\right]\!\right]\eta & = & \mathcal{V}_A[\![e_j]\!]\eta \\
\mathcal{V}_A[\![k(e_1,\ldots,e_n)]\!]\eta & = & \mathsf{Var}\ k\ (\mathcal{V}_A[\![e_1]\!]\eta,\ldots,\mathcal{V}_A[\![e_n]\!]\eta) \\
\mathcal{V}_A[\![s^\tau(e_1,\ldots,e_n)]\!]\eta & = & s^A_{\tau^A}\ (e^A_1,\ldots,e^A_n) \\
 & & \text{where}\quad e^A_i = \begin{array}{ll} s^A_{\tau^A} & \text{if } e_i \equiv s^\tau \\ \mathcal{V}_A[\![e_i]\!]\eta & \text{otherwise} \end{array} \\
\mathcal{V}_A\left[\!\left[K_R\{\overline{l_i = e_i}\}\right]\!\right]\eta & = & \mathsf{Rec}^{K_R}\{\overline{l_i = \mathcal{V}_A[\![e_i]\!]\eta}\} \\
\mathcal{V}_A\left[\!\left[\mathsf{case}(e,\{\overline{k_i \to s_i^{\tau_i}}\})\right]\!\right]\eta & = & s^A_{j\ \tau^A_j}\ (v_1,\ldots,v_n) \\
 & & \text{where}\quad \mathsf{Var}\ k_j\ (v_1,\ldots,v_n) = \mathcal{V}_A[\![e]\!]\eta \\
\mathcal{V}_A\left[\!\left[\mathsf{case}(e,V,\{\overline{k_i \to s_i}\})\right]\!\right]\eta & = & \mathcal{V}_A\left[\!\left[\mathsf{case}(e,\{\overline{k_i \to s_i}\})\right]\!\right]\eta \\
\mathcal{V}_A[\![e :: \tau]\!]\eta & = & \mathcal{V}_A[\![e]\!]\eta
\end{array}
$$

Figure 5.11: Evaluating terms

the variant constructor k_j. Type constructor annotation in a case statement and type annotation, in general, are ignored during the evaluation of terms.

Due to type checking, applied functions result in a value, not in a function. Formally, the value of a term is a member of a domain τ^A in the universe of the Σ-algebra A. This guarantees that a quantifier never iterates over functions, such that on the logic level first-order properties are preserved.

5.7 Summary

In this chapter, we show how our new tolerant semantics points out inconsistent document parts. Instead of a simple boolean value, for each consistency rule, we generate a *consistency report.* Report generation is quite similar to classic evaluation of predicate logic formulae. Our approach deviates mostly in the treatment of quantifiers: We capture the bindings to values that cause inconsistencies. These bindings indicate *when* and *where* inconsistencies occur. In addition, we retain the boolean results of atomic formulae, which give reasons for rule violation.

Our formal structures for rule evaluation correspond to the formal structures in partial predicate logic, because we facilitate partial functions. This compatibility causes, however, some effort, because we also support parametric polymorphism, higher-order functions and predicates, and subtyping. Our type checker guarantees that applied function symbols always evaluate to first-order values (not to functions). This preserves the first-order properties of our logic, i.e., we do not quantify over functions. But, as pointed out earlier, these issues are theoretical matters; they do not restrict the practical applicability of our approach: Language designers simply use Haskell for the definition of symbol semantics. In the background, our tools simulate Nordlander-style subtyping, which has as yet not been implemented in a Haskell compiler [Maz03] (see also App. B).

What really impacts the practical applicability of our approach so far is the computational expense of report generation. Evaluating our example rules

Check-in	Repository Changes	Number of Inconsistencies	CPU Time (Sec.)
1	txt: 1n, key: 1n, man: 9n	0	5.23
2	txt: 1c, 4n	4	13.48
3	man: 2c	14	18.95
4	man: 2c	19	26.38
5	txt: 1c	21	33.40
6	txt: 1c	26	41.15
7	key: 1n	31	47.11
8	man: 1n	35	57.34
9	key: 1c	36	67.14

Table 5.2: Brute force consistency checking performance (tests were performed against a DARCS repository [Rou04] on a Dell X200 laptop; 800 MHz PIII CPU)

in a brute force manner results in an extremely poor performance. Tab. 5.2 shows the development of a “toy” repository, which we shall use throughout for performance measurements. The second column shows how many documents were added (n) or changed (c) by a check-in. We use `man` for manuals, `key` for key resolvers, and `txt` for plain text documents. Inconsistencies found when checking the rules ϕ_1 and ϕ_2 are summarized in the third column. The final column shows the CPU time needed for consistency checking. To a great extent, CPU time depends on the repository state, because brute force checking evaluates *every* repository state.

The results in Tab. 5.2 are unacceptable but they are not surprising. Clearly, as in classic predicate logic, every quantifier sphere contributes a polynomial factor; the nesting of quantifiers results in an exponential behavior. In order to make our approach viable, it is absolutely necessary to develop methods that reduce evaluation time. In particular, evaluation time should not depend on the repository state. We discuss efficient report generation in the next chapter.

Chapter 6

Speeding up Consistency Checking

In Sect. 5.7, we have seen that we desperately need methods that reduce the time needed for consistency checking. In fact, *speed* is a key to user acceptance. After a check-in to the repository, authors want to know almost immediately whether this check-in is accepted and how it meets the consistency rules. In this chapter, we address the issue of *efficiently* evaluating first-order linear temporal consistency rules against a heterogeneous repository. In contrast to many other approaches, we retain our tolerant semantics. We improve efficiency by two measures: First, we lower checking complexity by static analysis, which is performed prior to consistency checking. Second, we modify our consistency checking algorithm. Static rule analysis attempts to reduce the number of rules to be re-evaluated at a check-in by filtering, and to lower the static computational complexity of a rule by rewriting. Our modified consistency checking algorithm dynamically reduces quantifier spheres. Since we tolerate inconsistencies in previous repository states, we employ an incremental algorithm that makes heavy use of previous consistency reports. In document management, incremental consistency checking is a real challenge: In contrast to database approaches, we cannot benefit from a formal document model, a formal update model, and full consistency prior to a check-in. Consequently, we cannot achieve the high performance of constraint checking approaches for databases. We argue, however, that DMSs do not have to handle the high loads databases have to handle. Therefore, our methods suffice.

In this chapter, we proceed as follows: In Sect. 6.1, we explore the effect of our methods by our running example. We detail static rule analysis in Sect. 6.2. Sect. 6.3 is concerned with incremental evaluation. In Sect. 6.4, we give advice to rule designers and language designers about how to formalize "fast" consistency rules. We summarize this chapter in Sect. 6.5. Fig. 6.1 illustrates the context of this chapter.

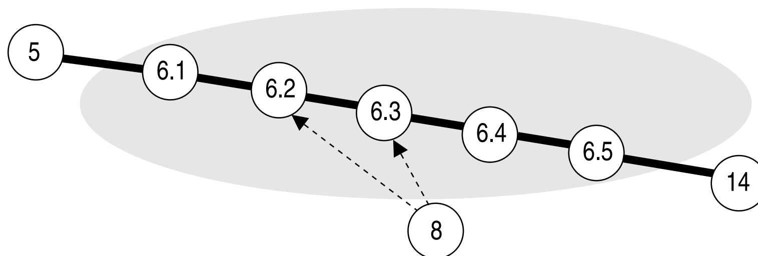


Figure 6.1: Chapter 6 in context

Check-in	Document	Modifications	
		Kind	Details
1	doc1.txt	add	...as shown in manual kaA3 ...
	keys.xml	add	<kDef key="kaA3" kId="man1.xml" kKind="technical M."/>
	man1.xml	add	<man kind="technical M."> ...
2	man1.xml	change	<man kind="field M."> ...
3	doc2.txt	add	...as shown in manual kaA2 ...

Table 6.1: Example repository up to state 3

$$\begin{aligned}\phi_1 = \; & \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\ & \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \exists\, m \in \texttt{repManDs}(t) \bullet \\ & k = \texttt{key}(d) \;\wedge\; \texttt{dId}(m) = \texttt{kId}(d) \;\wedge\; \texttt{kind}(m) = \texttt{kKind}(d)\end{aligned}$$

$$\begin{aligned}\phi_2 = \; & \forall\, t_1 \in \texttt{repStates} \bullet \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \forall\, t_2 \in \texttt{repStates} \bullet \\ & t_1 < t_2 \;\Rightarrow\; \begin{pmatrix}\exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{dId}(m_1) = \texttt{dId}(m_2) \;\wedge\; \texttt{kind}(m_1) = \texttt{kind}(m_2)\end{pmatrix}\end{aligned}$$

Figure 6.2: Formal example rules

6.1 Informal Overview

Static analysis tries to localize and simplify consistency rules *before* they are evaluated against a repository. It is performed exclusively on a syntactical level. *Localizing* a rule means to associate it with the set of documents affected by this rule. This reduces the number of rules to be re-evaluated at a check-in. A rule is *simplified* by minimizing its quantifier nesting and thus lowering the static evaluation time complexity. Our static methods are adaptations to database techniques [dC86, GSUW94, Pac97] that are already known.

When the DMS signals a check-in to the repository, we want to re-evaluate only those consistency rules that might be affected by this check-in. With each rule, we associate a set of affected documents, because we also target revision control systems, such as CVS [C+02], that lack a formal document model.[1] For example, we associate rule ϕ_1 (see Fig. 6.2) with all plain text documents and all XML documents {`*.txt,*.xml`}. Rule ϕ_2 is associated with all manuals {`man*.xml`}. At the third check-in (see Tab. 6.1), we do not need to re-evaluate ϕ_2. Instead, we lift its report to the new repository state.

The computational cost to evaluate a consistency rule depends on its deepest quantifier nesting, which is minimal if the scope of each quantifier is minimal. Such formulae are called *miniscope* [dC86]. Consider the following variant of rule ϕ_1:

[1] In more sophisticated DMSs, we could also define document classes (so-called stereotypes) and associate these classes with consistency rules.

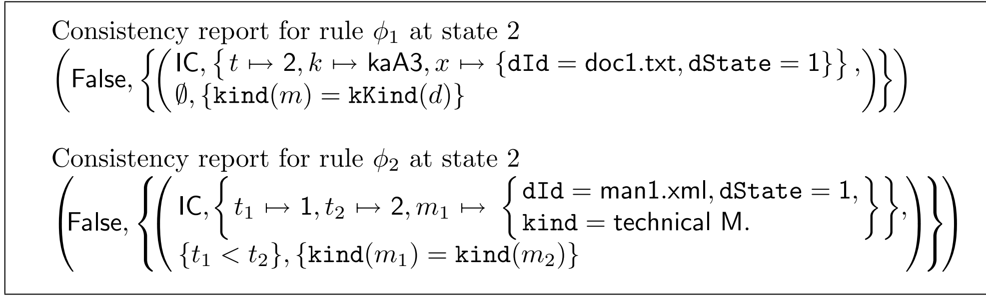


Figure 6.3: Example consistency reports at state 2

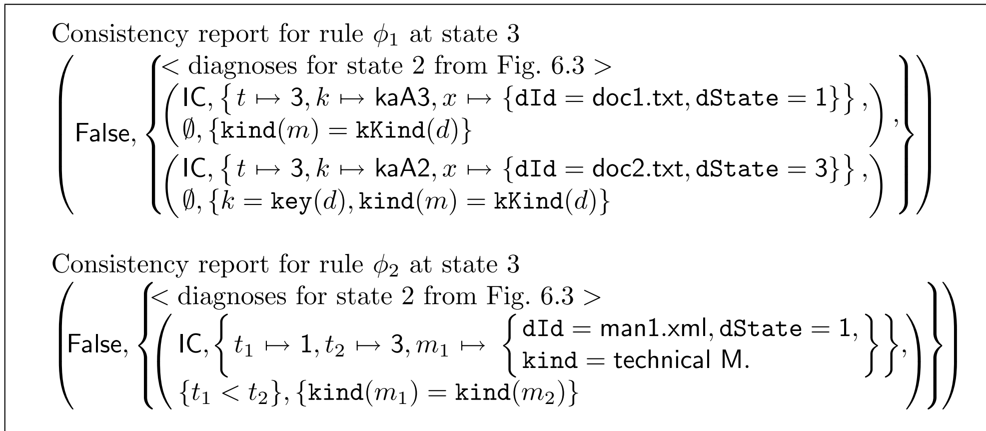


Figure 6.4: Example consistency reports at state 3

$$\phi_1' = \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet k = \texttt{key}(d) \;\wedge\; \begin{pmatrix} \exists\, m \in \texttt{repManDs}(t) \bullet \\ \texttt{dId}(m) = \texttt{kId}(d) \;\wedge\; \texttt{kind}(m) = \texttt{kKind}(d) \end{pmatrix}$$

Here, the existential quantifier for m was moved into the conjunction. Since in ϕ_1' the existential quantifier has a smaller scope, ϕ_1' can be evaluated faster than ϕ_1. We adapt the techniques in [dC86] to convert our rules to miniscope.

We shall see that static analysis improves performance between 15% and 32% for our running example. These results are, however, unsatisfactory: Still rule evaluation time depends on the repository state. A major reason is that we access documents at previous repository states. Usually, a DMS rebuilds these documents step by step using state transition descriptions, e.g., diffs or patches. Our *incremental consistency checking algorithm* attempts to avoid accessing past document versions, which improves performance significantly.

In our example, the third check-in changes the consistency report for ϕ_1 marginally (see Fig. 6.3 and Fig. 6.4). In fact, the report for state 3 contains all diagnoses from the previous report (at state 2). Naturally, the report for state 3 deviates from the previous report, due to the modifications by the third check-in.

Symbol	Type	dAcc	sRes	accD	accI
repDs	State → [Doc]	{*.txt,*.xml}	∅	∅	∅
repManDs	State → [ManD]	{man*.xml}	∅	∅	∅
repResDs	State → [ResD]	{keys*.xml}	∅	∅	∅
refs	Doc → [String]	∅	{*}	{0}	∅
concatMap	$(\alpha \to [\beta])\times[\alpha] \to [\beta]$	∅	∅	∅	$\{0, 1\}$
repStates	[State]	∅	∅	∅	∅
$=$	$\alpha\times\alpha \to$ Bool	∅	∅	∅	∅
$\leq$	$\alpha\times\alpha \to$ Bool	∅	∅	∅	∅
$<$	$\alpha\times\alpha \to$ Bool	∅	∅	∅	∅

Table 6.2: Example symbol metadata

For static rules like ϕ_1, which do not relate different repository states, a simple strategy would suffice: Just evaluate the rule for the new repository state and add the result to the accumulated previous report. Alas this breaks down for temporal rules like ϕ_2, which relate different repository states. Therefore, we have developed a general technique that applies to temporal rules, too. As a result, consistency checking time mostly depends on the check-in and the number of inconsistencies; the repository state and the size of the repository influence evaluation time marginally. As usual, it is most challenging to balance the achieved speedup against the space needed for storing auxiliary information. Our approach needs little additional information only (see Sect. 6.3.4). It exploits previous consistency reports, in order to re-evaluate rules w.r.t. a part of the documents they affect: If possible, it accesses modified (parts of) documents only.

In the next section, we detail our methods for static analysis; Sect. 6.3 is devoted to incremental consistency checking.

6.2 Static Analysis

In this section, we concentrate on static methods that improve the performance of our consistency checker. Firstly, we show our methods to *filter* consistency rules. Secondly, we *rewrite* rules, in order to lower their static evaluation time complexity. Finally, we summarize the improvements achieved by static analysis.

6.2.1 Filtering Rules

At a check-in to the repository, we want to re-evaluate only those rules that might be affected by this check-in. This requires to associate with each rule a set of documents, the rule depends on. If a consistency rule does not need to be re-evaluated, we *lift* its previous report to the current repository state. Alas not all rules are subject to rule filtering; we discuss sufficient conditions at the end of this section.

Rule filtering depends on appropriate metadata for function and predicate symbols. These metadata are added by the language designer with the help of static Haskell code analysis of the implementations. Notice, however, that

static analysis of function and predicate implementations only can determine a superset of the documents a symbol accesses. Tab. 6.2 summarizes the metadata for symbols we use for our running example. We distinguish between the documents a symbol might access (`dAcc`), the Strings a symbol might result in (`sRes`), the formal arguments responsible for *direct* document access (`accD`), and the formal arguments responsible for *indirect* document access (`accI`). Indirect document access is important for higher-order symbols. For example, the function symbol `concatMap` does not access documents directly. The function provided as first argument may, however, access documents the names of which are given in the second argument. Hence, the first and second argument are responsible for indirect document access.[2] We let regular expressions denote document sets, because even in a simple revision control system we can identify documents by name and check-in state.

From these metadata, we determine the documents a formula depends on. For brevity, we omit a formal definition of our straightforward algorithm that collects the metadata of used symbols by traversing formulae and terms. For our example rules, we obtain that ϕ_1 depends on the documents {`*.txt, *.xml, man*.xml, keys*.xml`}; ϕ_2 depends on the documents {`man*.xml`}. These document sets do not exactly correspond to our rule dependencies, because our algorithm collects regular expressions only and does not join them. This approach suffices, because, at a check-in, we match the names of modified documents against the document set of a rule. If any of the regular expressions matches, then the rule has to be re-evaluated.

In our example, the third check-in causes re-evaluation of ϕ_1, whereas ϕ_2 is not re-evaluated. We lift the previous report for ϕ_2 to the new repository state, because inconsistencies at state 2 are still present at state 3. Basically, lifting the previous report means to duplicate each diagnosis whose assignment contains a binding to the previous state. This binding is replaced by a binding to the new repository state. For ϕ_2, the report at state 2 contains one diagnosis whose assignment includes the binding $t_2 \mapsto \mathsf{2}$. We duplicate this diagnosis and replace the binding $t_2 \mapsto \mathsf{2}$ by $t_2 \mapsto \mathsf{3}$. We obtain the report shown in Fig. 6.4. For brevity, we omit our straightforward report lifting algorithm.

The fundamental prerequisite for rule filtering is that the report change depends on the modified documents only. Alas we cannot guarantee this assumption for arbitrary consistency rules. Consider the following variant of ϕ_2:

$$\begin{aligned}\phi_{2,init} = \;& \forall\, t_1 \in \texttt{init(repStates)} \bullet \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \\ & \forall\, t_2 \in \texttt{init(repStates)} \bullet t_1 < t_2 \Rightarrow \\ & \quad \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \texttt{dId}(m_1) = \texttt{dId}(m_2) \;\wedge\; \texttt{kind}(m_1) = \texttt{kind}(m_2)\end{aligned}$$

The above rule iterates over all repository states except the current state (the function `init` removes the last element of a list, `repStates` returns states in ascending order). The report for $\phi_{2,init}$ at state 2 is $(\mathsf{True}, \emptyset)$; its report at state 3 equals the report for ϕ_2 at state 2. Clearly, lifting the previous report fails here. What prohibits rule filtering is *calculation over repository states.*

[2] Arguments are numbered beginning by zero.

Therefore, we restrict rule filtering to those rules that avoid time calculations. For this, each symbol has to be annotated whether it involves calculation over repository states. Similar to the other symbol metadata, static Haskell code analysis can help the language designer to add these metadata. Since we support polymorphic function symbols like $\texttt{init} : \forall \alpha . [\alpha] \to [\alpha]$, we also prohibit rule filtering for those rules that contain a polymorphic function symbol whose result type is instantiated to a state type.[3] In the above example, the type of init is instantiated to $[\mathsf{State}] \to [\mathsf{State}]$.

6.2.2 Rewriting Rules

The computational expense to evaluate a consistency rule depends on its deepest quantifier nesting, which is minimal if the scope of each quantifier is minimal. Such formulae are called miniscope [dC86]. Consider the following variant of rule ϕ_1:

$$\begin{aligned}\phi_1' = \; & \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\ & \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \\ & k = \texttt{key}(d) \quad \wedge \quad \begin{pmatrix} \exists\, m \in \texttt{repManDs}(t) \bullet \\ \texttt{dId}(m) = \texttt{kId}(d) \;\wedge\; \texttt{kind}(m) = \texttt{kKind}(d) \end{pmatrix}\end{aligned}$$

Here, the existential quantifier for m was moved into the conjunction, which is sound, because $k = \texttt{key}(d)$ does not contain the variable m. Since in ϕ_1' the existential quantifier has a smaller scope, ϕ_1' can be evaluated faster than ϕ_1. For example, let each quantifier sphere have a cardinality of n and each atomic formula evaluate in constant time c, then evaluating ϕ_1 costs $n^5 \cdot 3c$, whereas evaluating ϕ_1' costs $n^4 \cdot c + n^5 \cdot 2c$ only.[4]

We adapt the techniques in [dC86] to convert our rules to miniscope. Miniscoping also removes implications, pushes negations into formulae, and "flattens" nested conjunctions and disjunctions. Our incremental consistency checking algorithm benefits from these simplifications. For our purposes, we use terminology from De Champeaux [dC86] and Wang [Wan60]. According to [Wan60] a predicate logic formula is in *miniscope* if:

W1. The only logical symbols are $\neg$ (negation), $\vee$ (disjunction), and $\wedge$ (conjunction), where $\neg$ can occur in front of atomic formulae only.

W2. A subformula of a conjunction (disjunction) is no conjunction (disjunction), i.e., formulae are "flattened."

W3. The body of a universal (existential) formula is not a conjunction (disjunction).

[3] Formally, a state type is built from State and any non-atomic type constructor in the type structure $\Omega(\mathbf{T})$ of the used signature (see Def. 4.1).

[4] Due to our simple example, the speedup is not very impressive. For more complex rules, we have experienced greater speedups.

W4. If the body of a quantified formula is a conjunction (disjunction), then each subformula of the conjunction (disjunction) contains the quantifier variable.

W5. No permutation of a sequence of universal (existential) quantifiers invalidates W4.

De Champeaux extends the above miniscope property to compressed miniscope. A predicate logic formula is in *compressed miniscope* if:

C1. The formula is closed and each quantifier introduces a unique variable.

C2. No two subformulae of a conjunction (disjunction) are in an *Instance* relationship.

C3. No two subformulae of a conjunction (disjunction) are in a *half-negated Instance* relationship.

Put roughly, a formula ϕ is an Instance of a formula ψ iff ϕ is subsumed by ψ, i.e., $\psi \Rightarrow \phi$. A formula ϕ is a half-negated Instance of a formula ψ iff $\neg\phi$ is an Instance of ψ, where the negation is pushed inward ϕ.

At first sight, it appears that we should convert consistency rules to compressed miniscope formulae. We have, however, to adapt the above definitions for our purposes, which is mainly due to the explicit sphere terms for quantifiers and that these sphere terms may evaluate to empty spheres. In addition, property W3 above is impractical for our purposes, because it implies to distribute universal quantifiers over conjunctions and existential quantifiers over disjunctions. We drop W3, because it would require re-computation of sphere terms and introduction of new variables (every quantifier introduces a different variable). In addition, we change the definition of the Instance relationship, which originally includes quantifier subsumption: For example, let a be a constant, p a unary predicate, and x a variable. Then $p(a)$ is an Instance of $\forall x \bullet p(x)$. In our setting, we have an explicit sphere *dom* for x, say $\forall\, x \in dom \bullet p(x)$. Without evaluating consistency rules, we cannot know with certainty whether $a \in dom$, which is required for making $p(a)$ an instance of $\forall\, x \in dom \bullet p(x)$. In our setting, two formulae are in an Instance relationship, if they are equal up to renaming of variables and permutation of conjunctions and disjunctions.

Formally, we define *weak compressed miniscope* formulae as follows:

Definition 6.1 (Weak compressed miniscope formula) A formula in *weak compressed miniscope* form fulfills the properties W1, W2, W4, W5, C1, C2, and C3 above, where two weak compressed miniscoped formulae are in an Instance relationship, if they are equal up to renaming of variables and permutation of conjunctions and disjunctions. □

We use the following six-step algorithm, which converts a consistency rule to weak compressed miniscope form:

1. Replace implications $\phi \Rightarrow \psi$ by disjunctions $\neg\phi \vee \psi$.

2. Push negations into the formula, such that only atomic formulae may appear in a negated context. As in classic predicate logic, negation distributes over quantifiers.[5]

3. Replace cascading conjunctions $(\phi_1 \land \phi_2) \land \phi_3$ by flat conjunctions $\phi_1 \land \phi_2 \land \phi_3$. Proceed similarly for disjunctions.

4. If below a conjunction or disjunction a subformula ϕ is in Instance relationship with another subformula, then remove ϕ. If below a conjunction a subformula is in half negated Instance relationship with another subformula, then replace the conjunction by the constant False. If below a disjunction a subformula is in half negated Instance relationship with another subformula, then replace the disjunction by the constant True. In conjunctions, remove all True subformulae. In disjunctions, remove all False subformulae. If a conjunction contains the constant False, then replace the conjunction by False. If a disjunction contains the constant True, then replace the disjunction by True.

5. Push quantifiers into the formula (see below).

6. Replace cascading conjunctions $(\phi_1 \land \phi_2) \land \phi_3$ by flat conjunctions $\phi_1 \land \phi_2 \land \phi_3$. This is necessary, because the above step might introduce cascading conjunctions.

Fig. 6.5 shows our algorithm that pushes quantifiers into a consistency rule (we adapt the algorithm from [dC86]). The general idea is as follows: Consider a quantifier over a flat conjunction $\phi_1 \land \ldots \land \phi_n$. This quantifier only affects those subformulae that contain the quantifier variable. Thus, we push the quantifier into the conjunction, such that its scope is restricted to these subformulae only. For disjunctions, we use a similar approach.

Our algorithm deviates from [dC86] mostly in its treatment of existential quantifiers. An existential quantifier distributes over a disjunction, only if its sphere contains values. Therefore, we add the formula $\neg\, \texttt{null}(e)$, which requires that the sphere e is not empty (the predicate symbol $\texttt{null} : \forall \alpha.[\alpha] \to \textsf{Bool}$ indicates whether a list is empty). Consider a quantifier Q over a formula ϕ that is neither a conjunction nor a disjunction. We first process the subformula ϕ, resulting in ϕ'. If ϕ' is quantified by the *same* quantifier Q, then the outer quantification $Q\ x \in e\ \bullet$ may be exchanged with the inner quantification $Q\ x' \in e'\ \bullet$ provided that the quantifications are independent from each other (i.e., x does not occur in e'). That way the outer quantifier can be pushed further into the subformula. Notice that different quantifiers are not exchanged.

In rule ϕ_2, miniscoping replaces the implication by a disjunction. Also, the universal quantifier for m_1 is pushed into the right hand side of this disjunction. This is sound, because the universal quantifications $\forall\ m_1 \in \texttt{repManDs}(t_1)\ \bullet$ and $\forall\ t_2 \in \texttt{repStates}\ \bullet$ are independent and can, therefore, be exchanged.

[5] Here, the case for a negated universal quantifier (for existential quantifiers proceed similarly): $\neg\forall\ x \in e \bullet \phi \iff \neg\forall x \bullet x \in e \Rightarrow \phi \iff \exists x \bullet \neg(\neg(x \in e) \lor \phi) \iff \exists x \bullet x \in e \land \neg\phi \iff \exists\ x \in e \bullet \neg\phi$.

$$\mathsf{pushQ} \quad : \ \mathcal{F} \rightarrow \mathcal{F}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{pushQ}(\exists\, x \in e \bullet \bigvee\{\phi_1, \ldots, \phi_n\})} \\
 & = \neg\, \texttt{null}(e) \ \wedge\ (\bigvee\ not_x) & \text{if } has_x = \emptyset \\
 & \quad \exists\, x \in e \bullet \bigvee\ \{\phi'_1, \ldots, \phi'_n\} & \text{else if } not_x = \emptyset \\
 & \quad \neg\, \texttt{null}(e)\ \wedge\ \bigvee\ (\{quant\} \cup not_x) & \text{otherwise}
\end{array}$$

$$\begin{array}{llll}
\text{where} & \phi'_i & = & \mathsf{pushQ}(\phi_i) \\
 & not_x & = & \{\phi \mid \phi \in \{\phi'_1, \ldots, \phi'_n\} \text{ and } x \notin \mathsf{fv}(\phi)\} \\
 & has_x & = & \{\phi \mid \phi \in \{\phi'_1, \ldots, \phi'_n\} \text{ and } x \in \mathsf{fv}(\phi)\} \\
 & quant & = & \mathsf{pushQ}(\exists\, x \in e \bullet \phi) \quad \text{if } has_x = \{\phi\} \\
 & & & \exists\, x \in e \bullet \bigvee\ has_x \quad \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{pushQ}(\exists\, x \in e \bullet \bigwedge\ \{\phi_1, \ldots, \phi_n\})} \\
 & = \bigwedge\ (\{\neg\, \texttt{null}(e)\} \cup not_x) & \text{if } has_x = \emptyset \\
 & \quad \bigwedge\ (\{quant\} \cup not_x) & \text{otherwise}
\end{array}$$

$$\begin{array}{llll}
\text{where} & \phi'_i & = & \mathsf{pushQ}(\phi_i) \\
 & not_x & = & \{\phi \mid \phi \in \{\phi'_1, \ldots, \phi'_n\} \text{ and } x \notin \mathsf{fv}(\phi)\} \\
 & has_x & = & \{\phi \mid \phi \in \{\phi'_1, \ldots, \phi'_n\} \text{ and } x \in \mathsf{fv}(\phi)\} \\
 & quant & = & \mathsf{pushQ}(\exists\, x \in e \bullet \phi) \quad \text{if } has_x = \{\phi\} \\
 & & & \exists\, x \in e \bullet \bigwedge\ has_x \quad \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{pushQ}(\forall\, x \in e \bullet X\ \{\phi_1, \ldots, \phi_n\})} \\
 & = X\ not_x & \text{if } has_x = \emptyset \\
 & \quad X\ (\{quant\} \cup not_x) & \text{otherwise}
\end{array}$$

$$\begin{array}{llll}
\text{where} & \phi'_i & = & \mathsf{pushQ}(\phi_i) \\
 & not_x & = & \{\phi \mid \phi \in \{\phi'_1, \ldots, \phi'_n\} \text{ and } x \notin \mathsf{fv}(\phi)\} \\
 & has_x & = & \{\phi \mid \phi \in \{\phi'_1, \ldots, \phi'_n\} \text{ and } x \in \mathsf{fv}(\phi)\} \\
 & quant & = & \mathsf{pushQ}(\forall\, x \in e \bullet \phi) \quad \text{if } has_x = \{\phi\} \\
 & & & \forall\, x \in e \bullet X\ has_x \quad \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
\mathsf{pushQ}(Q\ x \in e \bullet \phi) & = Q\ x' \in e' \bullet & \text{if } \phi' = Q\ x' \in e' \bullet \phi'' \\
 & \quad\quad \mathsf{pushQ}(Q\ x \in e \bullet \phi'') & \quad \text{and } x \notin \mathsf{fv}(e') \\
 & \quad \mathsf{pushQ}(Q\ x \in e \bullet \phi') & \text{if } \phi' = X\ \psi_1 \ldots \psi_n \\
 & \quad Q\ x \in e \bullet \phi' & \text{otherwise}
\end{array}$$

$$\text{where } \phi' = \mathsf{pushQ}(\phi)$$

$$\mathsf{pushQ}(X\ \{\phi_1, \ldots, \phi_n\}) \ = \ X\ \{\mathsf{pushQ}(\phi_1), \ldots, \mathsf{pushQ}(\phi_n)\}$$

$$\mathsf{pushQ}(\phi) \quad = \ \phi$$

Figure 6.5: Pushing quantifiers into formulae ($\bigvee$ and $\bigwedge$ denote disjunction and conjunction of a set of formulae, respectively; X stands for either $\bigvee$ or $\bigwedge$; Q denotes a quantifier).

Check-in	Repository Changes	Rules checked	CPU Time (Sec.) (brute Force)	CPU Time (Sec.) (Static Analysis)
1	`txt`: 1`n`, `key`: 1`n`, `man`: 9`n`	ϕ'_1, ϕ'_2	5.23	4.47
2	`txt`: 1`c`, 4`n`	ϕ'_1	13.48	9.55
3	`man`: 2`c`	ϕ'_1, ϕ'_2	18.95	15.25
4	`man`: 2`c`	ϕ'_1, ϕ'_2	26.38	20.40
5	`txt`: 1`c`	ϕ'_1	33.40	23.46
6	`txt`: 1`c`	ϕ'_1	41.15	27.98
7	`key`: 1`n`	ϕ'_1	47.11	32.50
8	`man`: 1`n`	ϕ'_1, ϕ'_2	57.34	44.66
9	`key`: 1`c`	ϕ'_1	67.14	45.34

Table 6.3: Performance improvements by static analysis (tests were performed against a DARCS repository [Rou04] on a Dell X200 laptop; 800 MHz PIII CPU)

$$\phi'_2 = \forall\, t_1 \in \texttt{repStates} \bullet \forall\, t_2 \in \texttt{repStates} \bullet \\ \neg(t_1 < t_2) \;\vee\; \begin{pmatrix} \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{dId}(m_1) = \texttt{dId}(m_2) \;\wedge \texttt{kind}(m_1) = \texttt{kind}(m_2) \end{pmatrix}$$

6.2.3 Achievements by Static Analysis

How does static analysis improve evaluation time? The last column in Tab. 6.3 shows benefits between 15% and 32%. Improvements achieved by miniscoping can be seen in states at which both rules are re-evaluated. But still, the results are unsatisfactory: Rule evaluation time depends on the repository state. A major reason is that we access documents at *previous* repository states. Usually, the DMS rebuilds these documents step by step using state transition descriptions, e.g., diffs or patches. Next, we introduce our incremental consistency checking algorithm, which attempts to avoid accessing past document versions. As we shall see, this improves performance significantly.

6.3 Incremental Consistency Checking

The major goals of incremental consistency checking are: (1) only access document versions at the current repository state and (2) whenever possible access modified documents only. For static rules like ϕ'_1, which do not relate different repository states, a very simple strategy would suffice: Just evaluate the rule for the new repository state and add the result to the accumulated previous reports. Alas this breaks down for temporal rules like ϕ'_2, which relate different repository states.

Therefore, we have developed a general technique that applies to temporal rules, too, and also performs better for static rules. As usual, it is most challenging to balance the achieved speedup against the space needed for storing auxiliary information. Our approach needs little additional information only (see Sect. 6.3.4). It exploits previous consistency reports, in order to re-evaluate rules only w.r.t. a part of the documents they affect. Our strategy is as follows:

1. Keep the old report from the previous consistency check.
2. If possible, re-evaluate a quantifier in a rule for modified sphere values only. For other sphere values, copy the relevant part from the old report.

A fundamental prerequisite to incremental evaluation is that the consistency report of a formula remains constant, if its quantifier spheres remain constant, i.e., the report depends on the free variables of a formula only. This requires that the result of each function and each predicate depends on its parameters only — a feature called *referential transparency* in functional programming. Since language designers use Haskell to define symbol semantics, the above strategy is sound.[6] Notable exceptions are the "unsafe" functions `repStates` and `repHead`, which are not referentially transparent.[7]

Since rule evaluation time depends mostly on the cardinality of quantifier spheres, we try to narrow them. We partition a quantifier sphere into four sets: *new* contains new values, *chg* contains changed values, *old* contains unmodified values, and *del* contains deleted values. In addition, we extend variable assignments, which bind variables to values, to also mark variables as new and old, respectively. We call these markers variable kinds.

The central idea behind incremental consistency checking is to *re-evaluate a subformula only, if it contains a variable that is marked as new* in the current assignment or if it contains an unsafe symbol. If a subformula contains old variables and referentially transparent symbols only, we copy part of the old report and abort re-evaluation of this subformula. Before defining our incremental consistency checking algorithm, we illustrate its effect on our running example.

6.3.1 Example

We consider re-evaluation of rule ϕ'_1 at the second and third check-in, respectively. Fig. 6.6 shows how our incremental consistency checking algorithm evaluates ϕ'_1. In the evaluation trees, vertices represent conjunctions, disjunctions, or quantifier spheres. Old values are printed in grey and new values in black. A path from the root to a leaf stands for a variable assignment. For convenience, we also show the result report of each copy action.

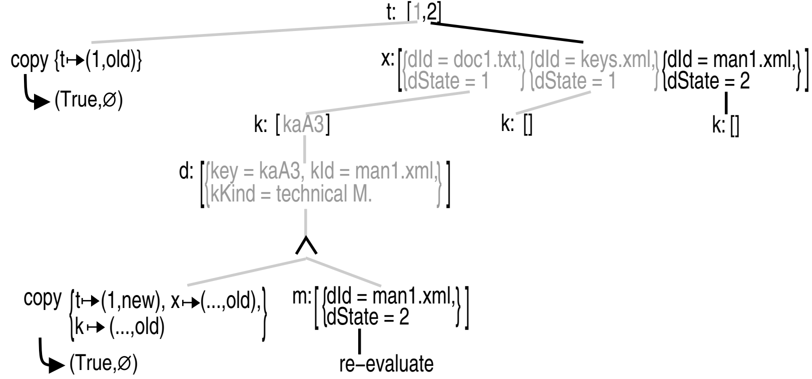
---

[6] In general, Haskell guarantees referential transparency. The use of Haskell is, however, not obligatory; any referentially transparent programming language (excerpt) suffices.

[7] We access the repository by some functions to get the repository states of already performed check-ins (`repStates`), get the state of the current check-in (`repHead`), get documents at a given state, and parse documents (see Sect. 5.5). The repository guarantees that, except for `repStates` and `repHead`, interface functions are referentially transparent although they involve IO. The interface functions `repStates` and `repHead` are not referentially transparent, because they read the states of already performed check-ins directly from the repository. Thus, their results change between (but not during) consistency checks. Static source code analysis can determine whether a function or predicate defined by the language designer is referentially transparent, i.e., it does not call `repStates` or `repHead` (directly or indirectly). Since this is done statically only, such analysis has to be very "pessimistic."

$$\phi_1' = \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet$$
$$\exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet$$
$$k = \texttt{key}(d) \quad \wedge \quad \left(\begin{array}{l} \exists\, m \in \texttt{repManDs}(t) \bullet \\ \texttt{dId}(m) = \texttt{kId}(d) \wedge \texttt{kind}(m) = \texttt{kKind}(d) \end{array} \right)$$

Evaluation tree at state 2



Evaluation tree at state 3

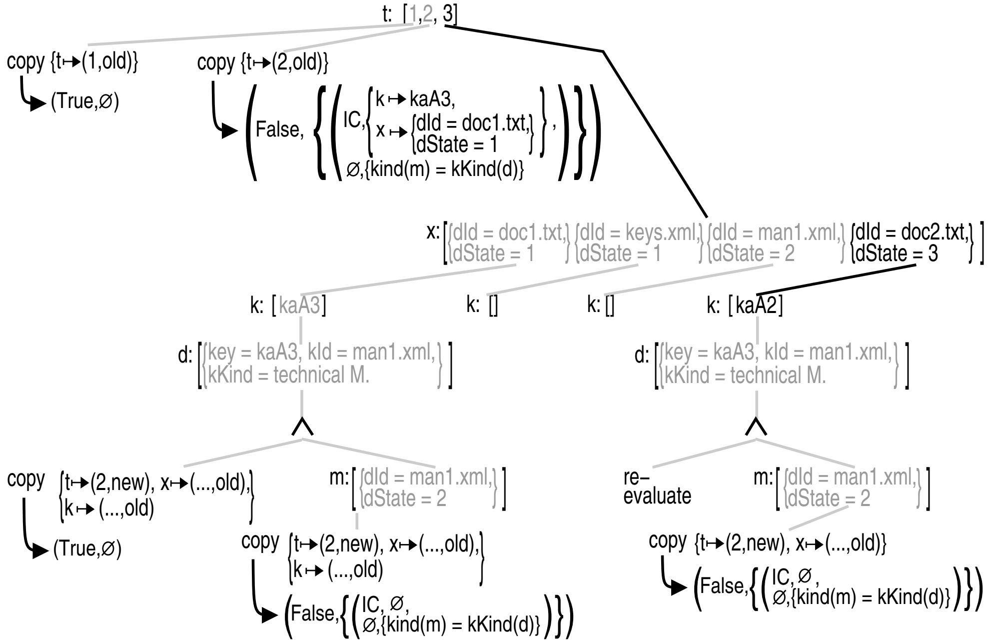


Figure 6.6: Incremental evaluation of rule ϕ_1' at state 2 and state 3

First, we consider the second check-in. At state 2, the sphere of the variable t is composed of the sets $new = \{2\}$, $chg = \emptyset$, $old = \{1\}$, and $del = \emptyset$. Since t's subformula only contains t freely and is referentially transparent, we can abort re-evaluation for values in *old*. Instead, we copy the relevant diagnoses from the old report for $t \mapsto (1, \mathsf{old})$, which results in the report $(\mathsf{True}, \emptyset)$. For $t \mapsto (2, \mathsf{new})$, we have to re-evaluate t's subformula. We review evaluation of the conjunction below the existential quantifier for d ($k = \mathtt{key}(d) \ \wedge \ \exists\, m \ldots$) for the variable assignment

$$\left\{ \begin{matrix} t \mapsto (2, \mathsf{new}), x \mapsto (\{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}, \mathsf{old}), k \mapsto (\mathsf{kaA3}, \mathsf{old}), \\ d \mapsto (\{\mathtt{key} = \mathsf{kaA3}, \mathtt{kId} = \mathsf{man1.xml}, \mathtt{kKind} = \mathsf{technical\ M.}\}, \mathsf{old}) \end{matrix} \right\}$$

For the left hand side of the conjunction, we can copy part of the old report, because all variables in the formula $k = \mathtt{key}(d)$ are marked as old in the current assignment. Notice that we adapt the binding $t \mapsto (2, \mathsf{new})$ to the previous repository state: $t \mapsto (1, \mathsf{new})$. Clearly, the old report cannot contain a binding of t to the new repository state. In addition, we neglect bindings of existentially quantified variables for copying, because assignments in diagnoses contain bindings to universally quantified variables only. The right hand side of the conjunction must be re-evaluated, because it contains the new variable t freely.

At the third check-in, we copy parts of the old report for the bindings $t \mapsto (1, \mathsf{old})$ and $t \mapsto (2, \mathsf{old})$, where the latter results in:

$$\left(\mathsf{False}, \left\{ \left(\begin{matrix} \mathsf{IC}, \{k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}\}, \\ \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{matrix} \right) \right\} \right)$$

The above report lacks a binding for t. The current binding for t is pushed in later by t's universal quantifier. For the new state 3, t's subformula must be re-evaluated. We review evaluation of the conjunction $k = \mathtt{key}(d) \ \wedge \ \exists\, m \ldots$ for the assignment

$$\left\{ \begin{matrix} t \mapsto (3, \mathsf{new}), x \mapsto (\{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}, \mathsf{old}), k \mapsto (\mathsf{kaA3}, \mathsf{old}), \\ d \mapsto (\{\mathtt{key} = \mathsf{kaA3}, \mathtt{kId} = \mathsf{man1.xml}, \mathtt{kKind} = \mathsf{technical\ M.}\}, \mathsf{old}) \end{matrix} \right\}$$

In Fig. 6.6, this conjunction is on the left hand side. Again, we copy the relevant part of the old report for the left hand side of the conjunction. This results in a true report although the old report contains a diagnosis with the assignment $\{t \mapsto 2, k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\}\}$. This diagnosis, however, does *not* contain the atomic formula $k = \mathtt{key}(d)$. Hence, the left hand side of the conjunction was not responsible for an inconsistency at the previous repository state. In contrast, below the existential quantifier for m (at the right hand side of the conjunction) copying results in the report

$$\left(\mathsf{False}, \left\{\left(\mathsf{IC}, \emptyset, \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\}\right)\right\}\right)$$

That is, because $\mathtt{kind}(m) = \mathtt{kKind}(d)$ *is* included in the diagnosis above. The above report does not contain any bindings; they are pushed in by universal

$$\phi_2' = \forall\, t_1 \in \texttt{repStates} \bullet \forall\, t_2 \in \texttt{repStates} \bullet$$
$$\neg(t_1 < t_2) \;\vee\; \begin{pmatrix} \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{dId}(m_1) = \texttt{dId}(m_2) \;\wedge\; \texttt{kind}(m_1) = \texttt{kind}(m_2) \end{pmatrix}$$

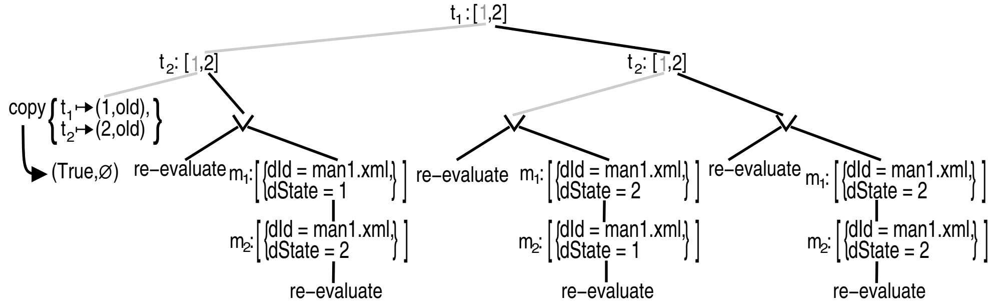


Figure 6.7: Incremental evaluation of rule ϕ_2' at state 2.

quantifiers, as done by brute force consistency checking. When we evaluate the conjunction $k = \texttt{key}(d) \;\wedge\; \exists\, m \ldots$ for the assignment

$$\begin{Bmatrix} t \mapsto (3, \mathsf{new}), x \mapsto (\{\texttt{dId} = \mathsf{doc2.txt}, \texttt{dState} = 3\}, \mathsf{new}), k \mapsto (\mathsf{kaA2}, \mathsf{new}), \\ d \mapsto (\{\texttt{key} = \mathsf{kaA3}, \texttt{kId} = \mathsf{man1.xml}, \texttt{kKind} = \mathsf{technical\ M.}\}, \mathsf{old}) \end{Bmatrix}$$

the left hand side must be re-evaluated, because now k is bound to a new value. Below the existential quantifier for m (at the right hand side of the conjunction) we can copy part of the old report, because the variables m and d are both bound to old values. Notice that for copying we neglect the binding of k, because it is bound to a new value. Clearly, in the old report we cannot find a binding of k to the new value kaA2.

Consistency checking for rule ϕ_2' proceeds similarly (see Fig. 6.7). Alas, copying parts from the old report is unsound below disjunctions (see the Sect. 6.3.2 for a detailed discussion). Therefore, we evaluate ϕ_2' non-incrementally below the disjunction.

We have seen that our incremental strategy also copies parts of the old report when a rule is evaluated for the new repository state. Clearly, this could not be achieved by the simple strategy from the beginning of Sect. 6.3. Our incremental algorithm is quite simple to understand but copying diagnoses from the old report may become rather complex. That is, because our consistency checker combines diagnoses below conjunctions.

6.3.2 An Incremental Consistency Checking Algorithm

In this section, we define our new evaluation algorithm — an incremental variant of brute force evaluation presented in Chapter 5. Recall that we want to improve the algorithm from Sect. 5.3 as follows:

- We copy those parts from previous reports that have not changed.

η	::=	$\{\overline{b_i}\}$	incremental variable assignment
b	::=	$x \mapsto (k, v)$	incremental binding (x variable, v value)
k	::=	$\mathsf{old} \mid \mathsf{new}$	variable kind $\mathbb{K}$

Figure 6.8: Incremental variable assignment $\mathbb{E}_{\text{inc}}$

- We partition quantifier spheres into four sets: *new*, *chg*, *old*, and *del*. Variable assignments mark variables as new and old, respectively. Our incremental algorithm deviates from brute force evaluation mostly in the treatment of quantified formulae.
- Due to miniscoping, only atomic formulae can appear in a negated context. In particular, existential quantifiers cannot "disguise" as universal quantifiers and vice versa. This considerably simplifies our algorithm.

Fig. 6.9 shows the denotational semantics of our incremental report generator, which has the formal type $\mathcal{IR}[\![.]\!] : \mathbb{A} \times \mathcal{F} \times \mathbb{E}_{\text{inc}} \to \mathbb{B} \times \wp(\mathbb{D})$. $\mathbb{E}_{\text{inc}}$ denotes the set of all incremental variable assignments (see Fig. 6.8). The function $\mathcal{IR}_A[\![\phi]\!]\eta$ is defined by structural induction on a formula ϕ (like brute force report generation in Sect. 5.3). Recall that A denotes a first-order structure and η stands for the current variable assignment. For readability, we introduce a global variable ®, which represents the old report. Superscripts denote the free variables of a formula; ϕ^{xs} means that the variables from the set xs are free in ϕ. For a formal definition of new auxiliary functions see Fig. 6.10 (pg. 83).

For every formula, $\mathsf{notEval}(xs, \eta)$ determines whether (1) its free variables xs are marked as old in the current assignment η and (2) it contains referentially transparent symbols only. In this case, we copy the relevant diagnoses from the old report ®. In order to identify relevant diagnoses, copy needs as additional arguments the current assignment η and the formula for which the report should be copied. We discuss details about copying below.

Miniscoping also simplifies the handling of conjunctions and disjunctions, because we have to generate a report only if the formula is violated. Also, we neglect implications, which are removed by miniscoping. If an atomic formula or a negated (atomic) formula needs to be re-evaluated, we employ non-incremental report generation. Recall that only atomic formulae can appear in a negated context; thus incremental report generation performs exactly as non-incremental report generation for negated formulae. We let $\eta_{\mathbb{E}}$ denote a non-incremental variable assignment that contains all bindings from the incremental assignment η without variable kinds.

The subformulae of a disjunction $\phi \vee \psi$ are evaluated non-incrementally, because it is unsound to copy parts of the old report. If the disjunction is fulfilled, one subformula, say ϕ, might be violated — consistency reports do not store these information. Consider the situation where a rule containing $\phi \vee \psi$ is fulfilled. At the next repository state, we might re-evaluate ψ only and copy part of the old report for ϕ, which results in $(\mathsf{True}, \emptyset)$. This is unsound, because ϕ was violated at the previous state. We could permit incremental evaluation below disjunctions, if we also stored diagnoses for violated subformulae of fulfilled

$$
\begin{array}{lcll}
\mathcal{IR}_A\llbracket p(e_1,\ldots,e_n)^{xs}\rrbracket\eta & = & \mathsf{copy}(p(e_1,\ldots,e_n),\circledR,\eta) & \text{if } \mathsf{notEval}(xs,\eta)\\
 & & \mathcal{R}_A\llbracket p(e_1,\ldots,e_n)\rrbracket\eta_{\mathbb{E}} & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{IR}_A\llbracket(\neg\phi)^{xs}\rrbracket\eta & = & \mathsf{copy}(\neg\phi,\circledR,\eta) & \text{if } \mathsf{notEval}(xs,\eta)\\
 & & \mathcal{R}_A\llbracket\neg\phi\rrbracket\eta_{\mathbb{E}} & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{IR}_A\llbracket\phi\wedge^{xs}\psi\rrbracket\eta & = & \mathsf{copy}(\phi\wedge\psi,\circledR,\eta) & \text{if } \mathsf{notEval}(xs,\eta)\\
 & & r_\phi\otimes r_\psi & \text{else if } \mathsf{fst}(r_\phi)=\mathsf{fst}(r_\psi)=\mathsf{False}\\
 & & r_\phi & \text{else if } \mathsf{fst}(r_\phi)=\mathsf{False}\\
 & & r_\psi & \text{else if } \mathsf{fst}(r_\psi)=\mathsf{False}\\
 & & (\mathsf{True},\emptyset) & \text{otherwise}\\
\text{where } r_\phi & = & \mathcal{IR}_A\llbracket\phi\rrbracket\eta & \\
r_\psi & = & \mathcal{IR}_A\llbracket\psi\rrbracket\eta &
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{IR}_A\llbracket\phi\vee^{xs}\psi\rrbracket\eta & = & \mathsf{copy}(\phi\vee\psi,\circledR,\eta) & \text{if } \mathsf{notEval}(xs,\eta)\\
 & & r_\phi\oplus r_\psi & \text{else if } \mathsf{fst}(r_\phi)=\mathsf{fst}(r_\psi)=\mathsf{False}\\
 & & (\mathsf{True},\emptyset) & \text{otherwise}\\
\text{where } r_\phi & = & \mathcal{R}_A\llbracket\phi\rrbracket\eta_{\mathbb{E}} & \\
r_\psi & = & \mathcal{R}_A\llbracket\psi\rrbracket\eta_{\mathbb{E}} &
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{IR}_A\llbracket\forall^{xs}\, x\in e\bullet\phi\rrbracket\eta & = & \mathsf{copy}(\forall\, x\in e\bullet\phi,\circledR,\eta) & \text{if } \mathsf{notEval}(xs,\eta)\\
 & & (\mathsf{False},\{(\mathsf{IC},\emptyset,\emptyset,\{\downarrow(e)\})\}) & \text{else if}\\
 & & & \mathcal{IV}_A\llbracket e\rrbracket\eta \text{ is not defined}\\
 & & \overline{\oplus}(F) & \text{else if } F\neq\emptyset\\
 & & (\mathsf{True},\emptyset) & \text{otherwise}\\
\text{where } (new,chg,old,del) & = & \mathcal{IV}_A\llbracket e\rrbracket\eta & \\
rs & = & \{(v,\mathcal{IR}_A\llbracket\phi\rrbracket(\eta\cup\{x\mapsto(v,\mathsf{old})\}))\mid v\in old\}\,\cup & \\
 & & \{(v,\mathcal{IR}_A\llbracket\phi\rrbracket(\eta\cup\{x\mapsto(v,\mathsf{new})\}))\mid v\in new\cup chg\} & \\
F & = & \{\mathsf{push}(x\mapsto v,r)\mid(v,r)\in rs \text{ and } \mathsf{fst}(r)=\mathsf{False}\} &
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{IR}_A\llbracket\exists^{xs}\, x\in e\bullet\phi\rrbracket\eta & = & \mathsf{copy}(\exists\, x\in e\bullet\phi,\circledR,\eta) & \text{if } \mathsf{notEval}(xs,\eta)\\
 & & (\mathsf{True},\{(\mathsf{C},\emptyset,\emptyset,\{\downarrow(e)\})\}) & \text{else if}\\
 & & & \mathcal{IV}_A\llbracket e\rrbracket\eta \text{ is not defined}\\
 & & (\mathsf{False},\{(\mathsf{IC},\emptyset,\{\mathtt{null}(e)\},\emptyset)\}) & \text{else if } T=F=\emptyset\\
 & & (\mathsf{True},\emptyset) & \text{else if } T\neq\emptyset\\
 & & \overline{\mathsf{min}}(F) & \text{otherwise}\\
\text{where } (new_0,chg,old_0,del) & = & \mathcal{IV}_A\llbracket e\rrbracket\eta & \\
(new,old) & = & (new_0\cup old_0\cup chg,\emptyset) \text{ if } del\cup chg\neq\emptyset; & \\
 & & (new_0,old_0) \quad \text{otherwise} & \\
rs & = & \{\mathcal{IR}_A\llbracket\phi\rrbracket(\eta\cup\{x\mapsto(v,\mathsf{old})\})\mid v\in old\}\,\cup & \\
 & & \{\mathcal{IR}_A\llbracket\phi\rrbracket(\eta\cup\{x\mapsto(v,\mathsf{new})\})\mid v\in new\} & \\
T & = & \{r\mid r\in rs \text{ and } \mathsf{fst}(r)=\mathsf{True}\} & \\
F & = & \{r\mid r\in rs \text{ and } \mathsf{fst}(r)=\mathsf{False}\} &
\end{array}
$$

Figure 6.9: An incremental report generation algorithm ($\circledR$ denotes the old report from the previous consistency check, $\eta_{\mathbb{E}}$ denotes a non-incremental assignment with bindings from η; for auxiliary functions see Fig. 5.10 (pg. 63) and Fig. 6.10)

disjunctions. This requires, however, to evaluate disjunctions strictly, whereas in our approach disjunctions can be evaluated non-strictly (i.e., we abort evaluation, if we find a fulfilled subformula). Our experiments have shown that strict evaluation of disjunctions leads to significantly longer times for consistency checking. Therefore, we have chosen to use non-incremental evaluation below disjunctions in favor of non-strict evaluation.

For a universally quantified formula $\forall\, x \in e \bullet \phi$, we first compute the sphere e, resulting in the four sets *new*, *chg*, *old*, and *del*. They are computed by the function $\mathcal{IV}_A[\![e]\!]\eta$, which we discuss in Sect. 6.3.4. Similar to $\mathcal{V}_A[\![e]\!]\eta$, incremental quantifier sphere calculation by $\mathcal{IV}_A[\![e]\!]\eta$ is partial. We mark the values in $new \cup chg$ as new and the values in *old* as old. Then the subformula ϕ is evaluated for possible assignment extensions to these values: $\eta \cup \{x \mapsto (v, \mathsf{new})\}$ and $\eta \cup \{x \mapsto (v, \mathsf{old})\}$, respectively. If ϕ is violated for an assignment extension, we push the current variable binding $x \mapsto v$ into the variable assignment of each diagnosis in ϕ's report. Finally, $\overline{\oplus}$ joins the resulting reports in F.

Evaluation of an existentially quantified formula is slightly more complicated. Here, the above procedure is sound, only if no values were changed or deleted in the sphere, i.e., $chg \cup del = \emptyset$. Naturally, an existentially quantified formula, which was satisfied in the previous repository state, can become violated if values in its sphere are deleted or changed. We do not know which values in the sphere were responsible for fulfilling the formula (this is quite similar to disjunctions). Adding new values to the sphere cannot falsify an existentially quantified formula. Consequently, if $chg \cup del$ contains values we must mark *every* sphere value as new in the assignment extensions. So the worst cases for incremental evaluation are changed or deleted values in spheres of existential quantifiers.

We could weaken the above restrictions by adding explicit information to the reports stating which values in the sphere of an existential quantifier are responsible for fulfilling its subformula. We would really like to do so but we found that reports become extremely large, if we store these additional values. Moreover, this strategy would require strict evaluation of existential quantifiers. Therefore, our tradeoff is to avoid storing these additional information even though evaluation of existential quantifiers can be slower than evaluation of universal quantifiers. Universal quantifiers do not suffer from changed or deleted sphere values, because consistency reports store value bindings of universally quantified variables — they pinpoint inconsistencies. Also, universal quantifiers must be evaluated strictly, similar to conjunctions.

For our incremental report generation algorithm, we introduce new auxiliary functions (see Fig. 6.10). To large part, incremental consistency checking depends on copying diagnoses from the previously generated report. The function copy determines the relevant diagnoses from the old report (b, ds) for a formula ϕ w.r.t. the current variable assignment η. From η we extract all bindings to universally quantified variables that are either marked as old, or where the bound value represents the current repository state. Diagnoses in the old

Copy relevant part of a report

$$
\begin{array}{lcl}
\mathsf{copy} & : & \mathcal{F} \times (\mathbb{B} \times \wp(\mathbb{D})) \times \mathbb{E}_{\text{inc}} \rightarrow \mathbb{B} \times \wp(\mathbb{D}) \\
\mathsf{copy}(\phi, (b, ds), \eta) & = & \begin{cases} (\mathsf{True}, \emptyset) & \text{if } ds' = \emptyset \\ (\mathsf{False}, ds') & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\text{where } ds' = \{(c, as', ps'_t, ps'_f) \mid (c, as, ps_t, ps_f) \in ds \text{ and } \\
\qquad\qquad \eta'_\forall \sqsubseteq as \text{ and } ps'_t \cup ps'_f \neq \emptyset\} \\
\qquad \text{where } as' = \{x \mapsto (v, m) \mid x \mapsto (v, m) \in as \text{ and } x \mapsto _ \notin \eta\} \\
\qquad\qquad ps'_t = ps_t \cap \mathsf{preds}(\phi) \\
\qquad\qquad ps'_f = ps_f \cap \mathsf{preds}(\phi) \\
\eta' = \{x \mapsto (v', m) \mid x \mapsto (v, m) \in \eta \ \wedge \ (m = \mathsf{old} \vee v = \mathsf{current\ state})\} \\
\qquad \text{where } v' = \begin{cases} \mathsf{previous\ check\ state} & \text{if } v = \mathsf{current\ state} \\ v & \text{otherwise} \end{cases}
\end{array}
$$

First assignment subsumes second assignment (preorder)

$$
\begin{array}{lcl}
\sqsubseteq & \subseteq & \mathbb{E}_{\text{inc}} \times \mathbb{E}_{\text{inc}} \\
as \sqsubseteq as' & \Leftrightarrow & x \mapsto (v, _) \in as \Rightarrow x \mapsto (v, _) \in as'
\end{array}
$$

Collect the atomic subformulae of a formula

$$
\begin{array}{lcl}
\mathsf{preds} & : & \mathcal{F} \rightarrow \wp(\mathcal{F}_{\text{at}}) \\
\mathsf{preds}(\forall\, x \in e \bullet \phi) & = & \{\downarrow(e)\} \cup \mathsf{preds}(\phi) \\
\mathsf{preds}(\exists\, x \in e \bullet \phi) & = & \{\mathtt{null}(e)\} \cup \mathsf{preds}(\phi) \\
\mathsf{preds}(\phi \cdot \psi) & = & \mathsf{preds}(\phi) \cup \mathsf{preds}(\psi) \\
\mathsf{preds}(\neg\phi) & = & \{\phi\} \\
\mathsf{preds}(\phi) & = & \{\phi\}
\end{array}
$$

Determine whether to re-evaluate a subformula

$$
\begin{array}{lcl}
\mathsf{notEval} & : & \wp(\mathcal{X}) \times \mathbb{E}_{\text{inc}} \rightarrow \mathbb{B} \\
\mathsf{notEval}(xs, \eta) & = & xs_{new} = \emptyset \text{ and } xs \neq \{*\} \\
\quad \text{where } xs_{new} & = & \{x \mid x \in xs \text{ and } x \mapsto (_, \mathsf{new}) \in \eta\}
\end{array}
$$

Figure 6.10: Auxiliary functions for incremental rule evaluation ($\{*\}$ denotes that a formula contains an unsafe symbol)

report can contain bindings to universally quantified old variables only. We replace bindings to the current repository state with bindings to the state of the previous consistency check. Clearly, this is the "current" state of the old consistency report. We call this modified assignment $\eta'_\forall$. Then a diagnosis in the old report is copied if (1) its assignment is subsumed by $\eta'_\forall$ (the diagnosis' assignment may contain more bindings than $\eta'_\forall$) and (2) some of its atomic formulae are subformulae of ϕ. Thus, atomic formulae also identify diagnoses. Finally, we adapt the assignments of relevant diagnoses to the new repository state and retain atomic formulae that are subformulae of ϕ. The function $\mathsf{notEval}$ determines whether a formula should be re-evaluated. We re-evaluate a formula if (1) all free variables in the set xs are marked as old in the current assignment η and (2) it contains referentially transparent symbols only (otherwise, the free variables are denoted by $\{*\}$).

Check-in	Repository Changes	Rules checked	CPU Time (Sec.) (Static Analysis)	CPU Time (Sec.) (Static Analysis & incr. Evaluation)
1	`txt`: 1n, `key`: 1n, `man`: 9n	ϕ'_1, ϕ'_2	4.44	4.47
2	`txt`: 1c, 4n	ϕ'_1	9.55	2.33
3	`man`: 2c	ϕ'_1, ϕ'_2	15.25	6.45
4	`man`: 2c	ϕ'_1, ϕ'_2	20.40	6.58
5	`txt`: 1c	ϕ'_1	23.46	2.68
6	`txt`: 1c	ϕ'_1	27.98	2.53
7	`key`: 1n	ϕ'_1	32.50	3.09
8	`man`: 1n	ϕ'_1, ϕ'_2	44.66	4.01
9	`key`: 1c	ϕ'_1	45.34	3.61

Table 6.4: Performance improvements of incremental evaluation over static analysis

6.3.3 Achievements by Incremental Evaluation

We return to our example repository. What does incremental evaluation buy? The last column in Tab. 6.4 shows the performance of our consistency checker using both static analysis and incremental evaluation. Now, evaluation time depends on the changed content rather than on the repository state. Notice that the primitive strategy described at the beginning of Sect. 6.3 cannot achieve the following: Except for states 3 and 4 every consistency check is faster than the initial consistency check *although* the repository grows. Also, if we add up the last column, we get an overall incremental checking time of 35.75 seconds. The performance of classic evaluation at state 9 is worse, because it re-evaluates rule ϕ'_1 for *all* documents at each repository state. Clearly, incrementalization imposes the overhead of reading previous consistency reports. In this example, however, this overhead is negligible, compared to the performance improvements of our incremental algorithm. Our case study in Part III confirms satisfactory performance of our prototype consistency checker; see Sect. 12.5 (pg. 211) for a performance summary.

6.3.4 Computing Quantifier Spheres Incrementally

In this section, we discuss computation of quantifier spheres. Recall that, in contrast to brute force evaluation, a sphere consists of four sets: *new* contains new values, *chg* contains changed values, *old* contains values that remained constant, and *del* contains deleted values.

A simple approach to incremental evaluation would require to store quantifier spheres from the previous evaluation. This is, however, infeasible, because quantifier spheres can become unexpectedly large since we do not control their vocabulary. For example, a quantifier might iterate over the complete document content. Instead of storing spheres, we *memoize* certain functions that occur in quantifier sphere terms [ABH03]. This reduces the space needed for storing intermediate data and maximizes the benefits for incremental evaluation. We distinguish functions by their result type. Storing repository states is cheap. We also consider documents by exploiting a natural property of a DMS:

A document can be identified by its name and check-in state. Since we only need to know whether a document has been modified by a check-in we store the "`Doc`-results" of each function the result type of which is a subtype of [`Doc`]. We neglect additional record labels, such as `kind` in the record type `ManD`. In summary, we memoize each function that occurs in a quantifier sphere term and has either [State] or a subtype of [`Doc`] as result type. In our example, these are: `repStates`, `repDs`, `repResDs`, and `repManDs`. This rather complicated approach is necessary because DMSs do not provide information about how the document structure has been changed by a check-in.

The function $\mathcal{IV}$ computes the sphere of a quantifier, which results in four sets. Fig. 6.11 shows the denotational semantics, where we concentrate on symbol application only. As in Sect. 5.6, we require that, in order to evaluate a term e, the values of all subterms of e must be defined; otherwise, the value of e is not defined. Evaluations of record constructions and case statements are straightforward. For the actual computation of a sphere, we use the helper function $\mathcal{IV}'$, which returns either a 4-tuple of values (*new*, *chg*, *old*, *del*) or a pair containing a value and a variable kind, i.e., new or old. Below, we refer to such a 4-tuple as an incremental value and to the pair as a non-incremental value. An incremental value results from evaluating a memoized applied function symbol, whereas non-incremental values result from evaluating other terms. If $\mathcal{IV}'$ produces an incremental value, then we simply convert this value to the four sets as required by our incremental consistency checker. Otherwise, we push the returned value into the old set or the new set depending on its kind.

The helper function $\mathcal{IV}'$ evaluates variables and symbols as usual. We mark referentially transparent symbols as old, because their evaluation does not change. In contrast, "unsafe" function symbols are marked as new, because their evaluation might have changed since the previous consistency check. For an applied function symbol s, returning a result of either type [State] or a subtype of [`Doc`], diff calculates four lists *new*, *chg*, *old*, and *del* from the current result and the stored result storedRes(s) from the previous evaluation. These lists are propagated up the term structure. Therefore, some restrictions apply to quantifier sphere terms. We only permit symbols that treat each list member separately, e.g., `concatMap`. Otherwise, if a symbol s is not memoized then we apply the implementation of s like in classic evaluation but also return the incremental kind of the result, which is old if each argument value of s is marked as old and s is referentially transparent.

We apply $\mathcal{IV}$ also to sphere terms that lack memoized functions, e.g., `refs`(x) in ϕ_1. If a free variable of a sphere term is marked as new in the current assignment or the term is unsafe, the lists *chg*, *old*, and *del* are empty and *new* contains the complete sphere. Otherwise, if all free variables of a sphere term are marked as old in the current assignment and the term is referentially transparent, $\mathcal{IV}$ returns all sphere values in *old* leaving *new*, *chg*, and *del* empty. This is sound, because, due to referential transparency, the sphere cannot have changed.

$$
\begin{array}{lll}
\mathcal{IV}[\![.]\!] & : & \mathbb{A} \times \mathcal{T} \times \mathbb{E}_{\text{inc}} \nrightarrow \wp(\mathbb{V}) \times \wp(\mathbb{V}) \times \wp(\mathbb{V}) \times \wp(\mathbb{V}) \\
\mathcal{IV}_A[\![e]\!]\eta & = & \mathsf{toSet}(vs_{incr}) \quad \text{if } \mathcal{IV}'_A[\![e]\!]\eta = vs_{incr} \\
 & & (\emptyset, \emptyset, \mathsf{toSet}(vs), \emptyset) \quad \text{if } \mathcal{IV}'_A[\![e]\!]\eta = (vs, \mathsf{old}) \\
 & & (\mathsf{toSet}(vs), \emptyset, \emptyset, \emptyset) \quad \text{if } \mathcal{IV}'_A[\![e]\!]\eta = (vs, \mathsf{new})
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{IV}'[\![.]\!] & : & \mathbb{A} \times \mathcal{T} \times \mathbb{E}_{\text{inc}} \nrightarrow (\mathbb{V} \times \mathbb{V} \times \mathbb{V} \times \mathbb{V}) \cup (\mathbb{V} \times \mathbb{K}) \\
\mathcal{IV}'_A[\![x]\!]\eta & = & \eta(x) \\
\mathcal{IV}'_A[\![s^\tau(e_1, \ldots, e_n)]\!]\eta & & \\
 & = & \mathsf{apply}(A, s, [\mathsf{fst}(v_1), \ldots, \mathsf{fst}(v_n)], resKind) \quad \text{if no } v_i \text{ incremental} \\
 & & \mathsf{applyI}(A, s, [\mathsf{toIncr}(v_1), \ldots, \mathsf{toIncr}(v_n)]) \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{where } v_i & = & (s_i^A, \mathsf{old}) \quad \text{if } e_i \equiv s_i \text{ and } s_i \text{ ref. transparent} \\
 & & (s_i^A, \mathsf{new}) \quad \text{if } e_i \equiv s_i \text{ and } s_i \text{ not ref. transparent} \\
 & & \mathcal{IV}'_A[\![e_i]\!]\eta \quad \text{otherwise} \\
resKind & = & \mathsf{new} \quad \text{if any } v_i \text{ is } \mathsf{new}, \text{ or } s \text{ is unsafe} \\
 & & \mathsf{old} \quad \text{otherwise}
\end{array}
$$

Apply a function to non-incremental values

$$
\begin{array}{lll}
\mathsf{apply} & : & \mathbb{A} \times \mathcal{S} \times [\mathbb{V}] \times \mathbb{K} \nrightarrow (\mathbb{V} \times \mathbb{V} \times \mathbb{V} \times \mathbb{V}) \cup (\mathbb{V} \times \mathbb{K}) \\
\mathsf{apply}(A, s, [v_1, \ldots, v_n], resKind) & & \\
 & = & \mathsf{diff}(\mathsf{storedRes}(s), s^A_{\tau^A}(v_1, \ldots, v_n)) \quad \text{if } s \text{ memoized} \\
 & & (s^A_{\tau^A}(v_1, \ldots, v_n), resKind) \quad \text{otherwise}
\end{array}
$$

Apply a function to incremental values

$$
\begin{array}{lll}
\mathsf{applyI} & : & \mathbb{A} \times \mathcal{S} \times [\mathbb{V} \times \mathbb{V} \times \mathbb{V} \times \mathbb{V}] \nrightarrow \mathbb{V} \times \mathbb{V} \times \mathbb{V} \times \mathbb{V} \\
\multicolumn{3}{l}{\mathsf{applyI}(A, s, [(v_{1,\mathsf{new}}, v_{1,\mathsf{chg}}, v_{1,\mathsf{old}}, v_{1,\mathsf{del}}), \ldots, (v_{n,\mathsf{new}}, v_{n,\mathsf{chg}}, v_{n,\mathsf{old}}, v_{n,\mathsf{del}})])} \\
 & = & \begin{pmatrix} new_{\mathsf{new}} + chg_{\mathsf{new}} + old_{\mathsf{new}} + del_{\mathsf{new}}, \\ new_{\mathsf{chg}} + chg_{\mathsf{chg}} + old_{\mathsf{chg}} + del_{\mathsf{chg}}, \\ new_{\mathsf{old}} + chg_{\mathsf{old}} + old_{\mathsf{old}} + del_{\mathsf{old}}, \\ new_{\mathsf{del}} + chg_{\mathsf{del}} + old_{\mathsf{del}} + del_{\mathsf{del}} \end{pmatrix} \quad \text{if } new \text{ incremental} \\
 & & (\mathsf{fst}(new), \mathsf{fst}(chg), \mathsf{fst}(old), \mathsf{fst}(del)) \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{where } new & = & \mathsf{apply}(A, s, [v_{1,\mathsf{new}}, \ldots, v_{n,\mathsf{new}}], \mathsf{new}) \\
old & = & \mathsf{apply}(A, s, [v_{1,\mathsf{old}}, \ldots, v_{n,\mathsf{old}}], \mathsf{old}) \\
chg & = & \mathsf{apply}(A, s, [v_{1,\mathsf{chg}}, \ldots, v_{n,\mathsf{chg}}], \mathsf{new}) \\
del & = & \mathsf{apply}(A, s, [v_{1,\mathsf{del}}, \ldots, v_{n,\mathsf{del}}], \mathsf{old})
\end{array}
$$

Convert a value to an incremental value

$$
\begin{array}{lll}
\mathsf{toIncr} & : & (\mathbb{V} \times \mathbb{V} \times \mathbb{V} \times \mathbb{V}) \cup (\mathbb{V} \times \mathbb{K}) \nrightarrow \mathbb{V} \times \mathbb{V} \times \mathbb{V} \times \mathbb{V} \\
\mathsf{toIncr}(v, k) & = & (v, v, v, v) \\
\mathsf{toIncr}(v_{\mathsf{new}}, v_{\mathsf{chg}}, v_{\mathsf{del}}, v_{\mathsf{old}}) & & \\
 & = & (v_{\mathsf{new}}, v_{\mathsf{chg}}, v_{\mathsf{del}}, v_{\mathsf{old}})
\end{array}
$$

Figure 6.11: Incremental quantifier sphere evaluation (toSet converts a list to a set (distributes over tuples), diff determines differences between two lists, $\mathsf{storedRes}(s)$ denotes the memoized result of the symbol s, $+\!\!+$ concatenates lists, $[\mathbb{V}]$ denotes the set of all lists with elements in the set $\mathbb{V}$)

6.4 Formalizing Efficient Consistency Rules

We have seen that quite some effort is required to speed up consistency checking. In this section, we shortly review the impact of our techniques to the work of the language designer and the rule designer, respectively. Generally, rules should be "local." Obviously, our techniques described in this chapter perform better on rules that affect a few documents only.

The *language designer* should associate with each symbol a narrow document set that the symbol potentially accesses. Static analysis of symbol implementation can determine a superset of these documents only. This also means to modularize document access functions. Our static analysis methods can then associate tighter document sets to rules.

The *rule designer* should formalize local consistency rules. A rule should concentrate on one requirement only; independent requirements should be formalized in different rules. This may also lead to more comprehensible rules. The rule designer should restrict himself to using referentially transparent symbols. In particular, unsafe symbols below disjunctions and implications prohibit incremental evaluation.

6.5 Summary

In this chapter, we present methods that speed up consistency checking significantly. The techniques shown make our tolerant approach viable for general document management, which is a rather informal application area (as compared to databases).

By exploiting domain knowledge supplied by the language designer, we analyze consistency rules statically: (1) we associate with each rule a document set the rule depends on; (2) we miniscope rules, in order to lower their static evaluation time complexity. At run-time, we re-evaluate a rule only on modified documents if possible. Haskell's referential transparency is a fundamental prerequisite to the soundness of our techniques. Our performance measurements prove that static analysis combined with incremental evaluation results in a significant speedup compared to brute force evaluation. We conjecture that our incremental evaluation algorithm could be of value in other research areas as well.

So far we have concentrated on showing inconsistencies to authors. In the next part of this thesis, we extend this approach to also suggesting *repairs* that can resolve inconsistencies. We will see that, in order to generate repairs efficiently, we need to modify our incremental algorithm slightly. Our methods for static analysis carry over without any adaptations.

Part II

Consistency Maintenance: Repairing Inconsistencies

The second part of this thesis is devoted to *consistency maintenance*. We extend our consistency checking approach by methods that give advice to authors about how inconsistencies can be resolved. In Chapter 7, we discuss basic design decisions and show how we integrate consistency maintenance into the every-day work with a DMS. We shall see that simple repair enumeration is infeasible due to the large number of possible repair actions. Therefore, we develop a new two-step approach towards efficient repair generation. In the first step, shown in Chapter 8, we model inconsistencies and possible repairs with the help of directed acyclic graphs (DAGs). We shall see that we can re-use many ideas from Chapter 6 to speed up the generation of these DAGs. In the second step, shown in Chapter 9, we derive a repair collection from DAGs.

Chapter 7

From Consistency Awareness towards Consistency Maintenance

Up to now we have concentrated on finding inconsistencies. Consistency reports leave it, however, unclear how inconsistencies can be best resolved. In this chapter, we explore the design space of *generating repairs* that resolve inconsistencies. In Sect. 7.1, we determine basic design decisions for document repair generation. Sect. 7.2 illustrates why repair enumeration is infeasible in document management. This motivates our new two-step repair generation approach, which we outline in Sect. 7.3. Finally in Sect. 7.4, we integrate this two-step approach into our model of consistency-aware DMSs shown in Chapter 3. Fig. 7.1 illustrates the context of this chapter.

7.1 How Can We React to Inconsistencies?

In this section, we determine basic design decisions that outline the road we shall follow in the next chapters. In general, there are various ways possible to react to inconsistencies.

1. Ignore inconsistencies or forbid inconsistencies at all, i.e., reject check-ins that cause inconsistencies.

2. Permit violation of weak consistency rules and send their consistency reports to the authors involved. Prohibit violation of strict rules, i.e., reject check-ins that raise inconsistencies.

3. Permit violation of weak rules but forbid violation of strict rules. In either case derive repairs and send them to the authors involved.

4. Prohibit violation of rules, derive repairs for inconsistencies, and apply these repairs iteratively until the repository is consistent. This strategy is widely used in active databases.

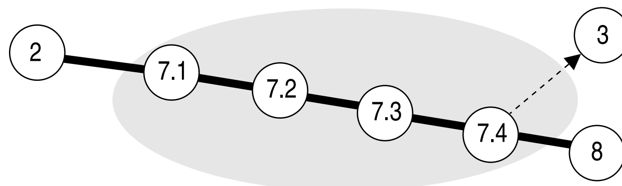


Figure 7.1: Chapter 7 in context

The first alternative represents the way current DMSs or revision control systems "react" to semantic inconsistencies. Since it does not contribute to consistency maintenance, we drop this alternative.

The second alternative makes use of our consistency reports, which are generated as shown in Part I. The rule designer could add metadata to each rule indicating who should receive which part of the consistency report. In addition, we could attach the documents that caused inconsistencies.

The third alternative goes beyond consistency reports, which pinpoint inconsistencies but fail to indicate how these inconsistencies can be resolved. We shall further explore this idea in this part of this thesis. Similar to the preceding chapters, we borrow ideas from the database community and try to apply them to document management. It will, however, turn out that, due to less formal documents, a very careful treatment of database consistency maintenance approaches is necessary.

The fourth alternative appears attractive. This iterative approach is, however, not guaranteed to terminate, because rule designers can formalize contradicting rules. Clearly, this is due to our expressive rule language, in which it is undecidable whether for a rule system there exists a repository that fulfills all rules. In addition, less formal document semantics prohibits the automatic application of computer generated repairs (see below).

In general, we consider three repair types:

- *Delete* content from a document or delete a document.
- *Change* content in a document.
- *Add* content to a document or add a document. Here, we suffer from lack of information, because, in general, we do not know *where* to add new document content. In contrast to databases, the *order* of the content in a document may be significant.

These repairs can be either sent to the authors of the affected documents or we can try to apply repairs automatically. Whereas the former involves again the author to resolve inconsistencies, the latter approach does not require author interaction. There are, however, several issues to be dealt with: First, it is known from database research that on the syntax level we cannot anticipate how a repair for one rule impacts consistency of other rules. Therefore, databases apply repairs iteratively, i.e., repairs are applied and then the database is checked for consistency again, which can raise new inconsistencies that must be repaired, too. Using iterative repair strategies requires sophisticated algorithms that handle loops and potential deadlocks.

In contrast to constraint maintenance in active databases, we do not aim at automatic repair. Instead, we only derive some of the best (i.e., least cost) repairs, from which authors can choose. In document management, automatic repair is not a feasible option, because (1) document structures are less formal than database schemata and (2) the order of the content in a document may be important for its semantics (this is in sharp contrast to relational databases,

where the order of tuples in a table is negligible). In particular, adding new content suffers from lack of formality, because we do not know where to add the content.

Of course, repairs must be "correct," i.e., satisfy the following requirements:

- A repair must resolve an inconsistency. This will be guaranteed by construction.
- A repair must not violate the document structure, e.g., the DTD or Schema of an XML document. For the majority of repairs, this is guaranteed by static type checking.
- A repair must not introduce new inconsistencies. We can determine such negative impacts only after applying a repair to a document in the repository. Since repairs are applied manually, we cannot guarantee this requirement automatically.

Intuitively, a set of repairs for some inconsistencies is considered *complete*, if it contains all possible repairs that can resolve inconsistencies. It is obvious that for a violated rule, formalized in first-order logic, we can find infinitely many repairs. Therefore, completeness is not a desirable option. Instead, we try to derive some of the best repairs only. This is impossible for our rules as such, because they employ *arbitrary* functions and predicates. For example, if an atomic formula $p(x, y)$ is violated, our system cannot anticipate how to change x and y, in order to fulfill $p(x, y)$.

Therefore, rule designers may annotate an atomic formula by *hints*. These hints guide repair generation by providing domain knowledge. Also, hints support high-level repairs, which can propose different actions for the same inconsistency depending on some other condition, e.g., the development state of a document engineering project. Static type checking guarantees that hints respect the document structure.

Our design decisions are as follows:

- We derive repairs that can resolve inconsistencies.
- Repairs for a violated rule must resolve inconsistencies for this rule.
- Hints from the rule designer guide repair generation.
- We are not striving for a complete set of all repairs for violated rules. Instead, we try to find a small set of repairs that proposes small changes to the repository.

Next, we illustrate an enumeration approach towards repair generation. Although simple repair enumeration is infeasible in our setting, the next section motivates our new two-step approach towards efficient repair generation, which we outline in Sect. 7.3.

7.2 Generating Repairs — A First Account

From the hints of the rule designer, we can derive suggestions for violated atomic formulae. One might be tempted to enumerate repairs as follows: Suggestions for an atomic formula form a basic repair collection.[1] Each repair set of the collection is an alternative. Within a repair set, all repairs have to be applied simultaneously, in order to resolve inconsistencies. One could join the repair collections generated for the subformulae of a disjunction or the subformula of an existential quantifier. For conjunctions and universal quantifiers, one could compute the cartesian product of the repair collections generated for their subformulae (and map binary union over this product). By this algorithm one would compute the repair collection for each rule. Since a rule system can be seen as a conjunction of many rules, one could combine these collections as if the rules were subformulae of a conjunction. In sum, one would arrive at a repair collection, from which authors could choose an alternative repair set.

The above approach is, however, infeasible due to the large number of repairs in the collections, which lets cartesian products "explode" and causes exponential running time. For example, assume a conjunction $\phi \wedge \psi$, for which we already have computed the following repair collections:

$$\begin{array}{ll} \text{repair collection for } \phi: & \{\ \{r_1, r_2, r_3\}, \{r_4, r_5\}, \{r_6, r_7\}\ \} \\ \text{repair collection for } \psi: & \{\ \{r_8\}, \{r_9\}, \{r_{10}\}\ \} \end{array}$$

Then, by the above approach, we would compute the following repair collection:

$$\left\{ \begin{array}{l} \{r_1, r_2, r_3, r_8\}, \{r_4, r_5, r_8\}, \{r_6, r_7, r_8\}, \\ \{r_1, r_2, r_3, r_9\}, \{r_4, r_5, r_9\}, \{r_6, r_7, r_9\}, \\ \{r_1, r_2, r_3, r_{10}\}, \{r_4, r_5, r_{10}\}, \{r_6, r_7, r_{10}\} \end{array} \right\}.$$

From two collections, each containing 3 sets, we would build a new collection containing $3 \cdot 3 = 9$ sets. This procedure is applied to *every* conjunction and *every* universal quantifier. In sum, inefficient cartesian product calculation would be applied to very large inputs, if we followed this approach.

Recall that performance of consistency checking is crucial, because the repository must be locked during consistency checking. In addition, interactions between repairs must be considered: Repairs within a repair set must be applicable simultaneously, e.g., they must not suggest to delete some content and at the same time to change this content.

7.3 Feasible Document Repair Generation: A new Two-Step Approach

In order to handle the large amount of possible repairs, we do not enumerate them. Instead, in the first step, for each rule we *describe* repair actions by a directed acyclic graph (DAG) only. We call these DAGs *suggestion DAGs* (short: S-DAGs). S-DAGs are optimized for efficient repair generation and

[1] Throughout, we use the term "repair collection" to mean a set of sets of repairs.

also provide a convenient way to visualize repairs. From an S-DAG, authors can choose repair actions interactively. S-DAGs provide a computationally tractable approach to generating useful repair actions; but they suffer from an inherent weakness: Authors choose actions for each rule *separately*, independent of their effect on other rules. If, however, many rules are violated, interactions between repair actions and potential impacts regarding overall consistency gain importance. Therefore, in the second step, we derive a single repair collection from the S-DAGs of all rules, similarly to the above approach. The repair collection contains alternative repair sets, each of which contains repairs that are necessary to resolve all inconsistencies in the repository. Our approach guarantees the following:

- Repairs really resolve inconsistencies.
- Each repair set contains compatible repairs only, i.e., repairs do not contradict each other.
- Repair sets provide mutually independent alternatives.

The collection can be sorted w.r.t. user-defined metrics that are based on repair ratings. We rate a repair according to its individual cost, the inconsistencies it resolves, and the rules that may be violated by applying it. Notice that the application of repairs might introduce new inconsistencies. From the repair collection, authors can choose a set of a high ranking and then manually apply its repairs to the documents in the repository. Since often time and cost restrictions prohibit the resolution of all inconsistencies, we also support *partial inconsistency resolution*. Instead of applying a complete repair set, authors can choose those repairs that resolve the most troubling inconsistencies at small costs.

In contrast to many other approaches (most notably [NEF03]), we derive repairs from S-DAGs only. Consequently, no further inspection of the repository is necessary, i.e., the repository is not locked during repair derivation. This is essential, because deriving repairs is computationally expensive as compared to generating S-DAGs. We consider the separation of S-DAG generation from repair derivation a major feature of our approach. In addition, S-DAGs are reduced during their generation w.r.t. the criterion of minimal change. Thus, derived repairs propose small changes to the repository only. Of course, repair collections are invalidated by further check-ins to the repository, which cause new consistency checks generating new S-DAGs. This is, however, no issue, because we derive repair collections on demand only.

We borrow some techniques from the database community. In the database context, however, active constraint maintenance aims to arrive at a consistent database state after each update. Therefore, active databases automatically apply repairs, if an inconsistency occurs. These repairs range from additional updates to the database towards a complete rollback of the update. Notice that at least part of the database is locked during constraint maintenance. The major differences between our approach and constraint maintenance approaches for databases are:

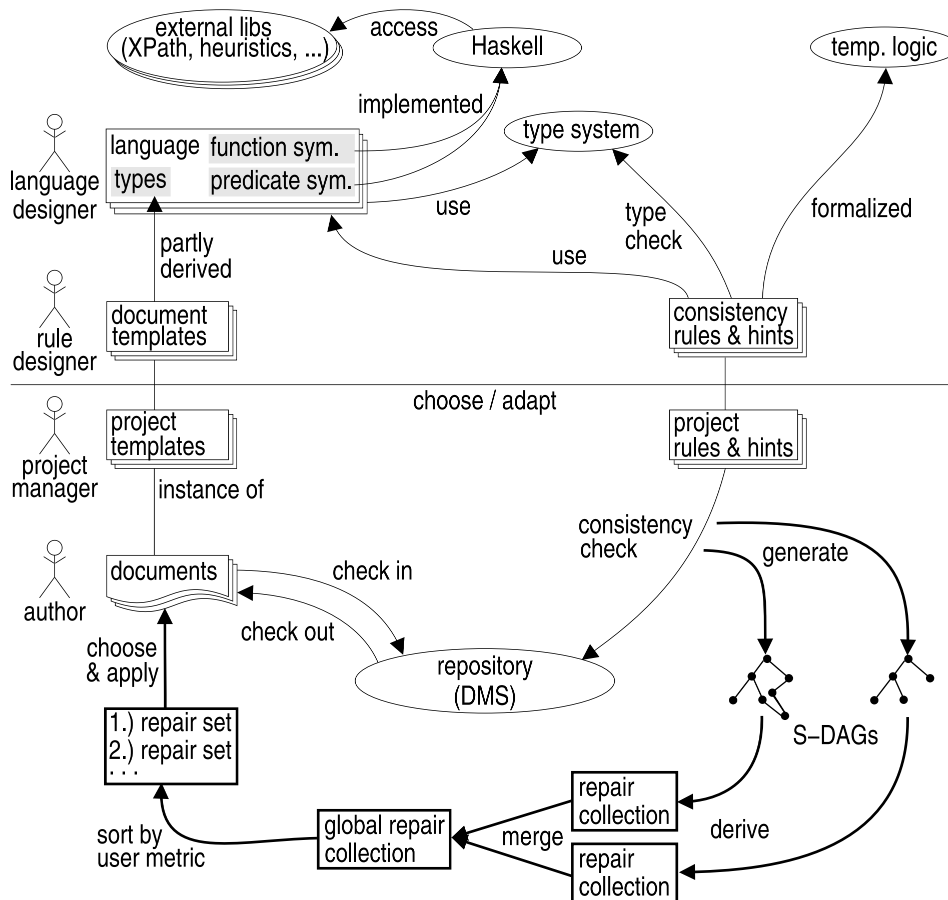


Figure 7.2: Overview of a consistency maintaining DMS (ovals mark fixed components; rectangles mark customizable components)

- We derive repairs from S-DAGs and not directly from the documents in the repository.
- Hints guide repair generation and facilitate flexible response to inconsistencies.
- We do not aim at automatic repair but provide authors with a few repair alternatives, which are prioritized w.r.t. user-defined preference metrics.
- We permit inconsistencies and facilitate partial inconsistency resolution.

Next, we show how we integrate our new two-step repair generation approach into our model of consistency-aware DMSs shown in Chapter 3.

7.4 Consistency Maintaining DMSs

We extend our DMS model from Fig. 3.2 (pg. 22) as shown in Fig. 7.2. The rule designer annotates consistency rules by hints that guide S-DAG generation.

For each rule, our consistency checker generates an S-DAG. During consistency checking, the repository is locked. S-DAGs already contain explicit information about inconsistencies that have occurred and how individual inconsistencies can be resolved. From each S-DAG, we derive a repair collection. This and all following steps do not access the repository and are, therefore, independent from consistency checking. We merge the repair collections, in order to determine a *single* repair collection for *all* rules. Finally, we sort this collection according to user-defined preference metrics.

We detail S-DAG generation in Chapter 8. In Chapter 9, we show how the repair collection can be derived from S-DAGs.

Chapter 8

S-DAGs: Towards Efficient Document Repair Generation

In this chapter, we present the first step of our consistency maintenance approach. Instead of just finding inconsistencies, we generate useful repair actions that can resolve inconsistencies. Recall that we tolerate inconsistencies; but we advise how they can be resolved. The decision to resolve inconsistencies still lays in the hands of authors. Following our approach in Sect. 7.3, this chapter is concerned with *S-DAGs*, which provide a comfortable means to show inconsistencies and possible repair actions to authors. Also, S-DAGs can be generated incrementally, which is a key to achieve scalability of our approach.

We start this chapter by giving an informal overview in Sect. 8.1. In Sect. 8.2, we show how hints are formalized and how they guide S-DAG generation. In Sect. 8.3, we explain S-DAGs in detail and explore how they can be reduced. We define an (incremental) algorithm for S-DAG generation in Sect. 8.4. In Sect. 8.5, we show how S-DAGs can be used for interactive repair. Sect. 8.6 summarizes this chapter. Fig. 8.1 illustrates the context of this chapter.

8.1 Informal Overview

In order to generate good repair actions, rule designers should annotate consistency rules by *hints*; Fig. 8.2 shows the annotated rule ϕ_1.[1] Basically, a hint indicates how the truth value of an atomic formula can be changed. Hints form a collection.[2] In a hint collection, each hint set is an alternative; within each

[1] We use the miniscoped variant of ϕ_1.

[2] Throughout, we use the term "hint collection" to mean a set of sets of hints.

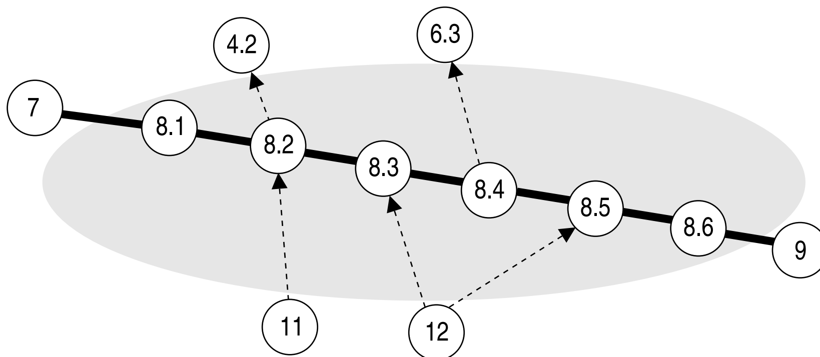


Figure 8.1: Chapter 8 in context

$$
\begin{array}{lll}
\phi_1 = & \forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet & \\
& \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet & \\
& k = \texttt{key}(d) \quad \left\{ \begin{array}{l} \{k \rightsquigarrow \texttt{key}(d)\ \mathsf{False}\ \ 1\}, \\ \{d.\texttt{key} \rightsquigarrow k\ \mathsf{False}\ \ 5\} \end{array} \right\} & \wedge \\
& \exists\, m \in \texttt{repManDs}(t) \bullet & \\
& \texttt{dId}(m) = \texttt{kId}(d) \quad \{\{m.\texttt{dId} \rightsquigarrow \texttt{kId}(d)\ \mathsf{False}\ \ 3\}\} & \wedge \\
& \texttt{kind}(m) = \texttt{kKind}(d) \quad \{\{m.\texttt{kind} \rightsquigarrow \texttt{kKind}(d)\ \mathsf{False}\ 2\}\} &
\end{array}
$$

Figure 8.2: Example rule ϕ_1 (miniscoped) with hints

set, all hints are evaluated simultaneously. For example, the atomic formula $k = \texttt{key}(d)$ is annotated by

$$
\left\{ \begin{array}{l} \{\ k \rightsquigarrow \texttt{key}(d)\ \mathsf{False}\ \ 1\ \}, \\ \{\ d.\texttt{key} \rightsquigarrow k\ \mathsf{False}\ \ 5\ \} \end{array} \right\}.
$$

The above hint collection proposes changes to one of the variables k or d, in order to invert the truth value of $k = \texttt{key}(d)$, if its boolean result is False. For this, we could either set the current value of k to the value of $\texttt{key}(d)$ or set the component `key` of d's value to the value of k. Clearly, then k and $\texttt{key}(d)$ are equal. We assign a cost of 1 to the first hint and a cost of 5 to the second hint. Changing key definitions is more expensive, because this may cause further inconsistencies. We annotate the temporal variable t by KEEP, in order to prevent our algorithm to suggest changing or deleting a repository state (clearly, in a DMS this hardly makes sense). The hint for the atomic formula $\texttt{dId}(m) = \texttt{kId}(d)$ sets the component `dId` of m's value only; we do not want to change the key definition d if $\texttt{dId}(m) = \texttt{kId}(d)$ is violated.

For each rule, our system generates an S-DAG, which visualizes inconsistencies and possible repair actions. The structure of an S-DAG resembles that of a consistency rule. Nodes represent logical connectives or atomic formulae; edges target the subformulae of a connective. Edges below quantifier nodes carry bindings of variables to values. A predicate node occurs as a leaf only. It contains an atomic formula ϕ responsible for an inconsistency, the truth value of ϕ, and the predicate suggestion collection[3] resulting from evaluating the hint collection for ϕ.

From the user perspective, a universal node Ⓥ represents inconsistencies resulting from *dubious* document content, where each edge blames a value for inconsistencies represented by the edge's target S-DAG. An existential node Ⓔ represents an inconsistency resulting from *missing* document content. This content could be either really missing or it could be regarded as missing, because one of the edges carries defective content. In Sect. 8.5.1, we present heuristics that recognize which of these possibilities applies. Below conjunction nodes $\textcircled{\wedge}$ and universal nodes Ⓥ each S-DAG must be repaired. In contrast, it is sufficient to repair only one S-DAG below a disjunction node $\textcircled{\vee}$ and existential node Ⓔ, respectively.

[3]Throughout, we use the term "predicate suggestion collection" to mean a set of sets of predicate suggestions.

Check-in	Document	Modifications	
		Kind	Details
1	doc1.txt	add	`...as shown in manual kaA3 ...`
	keys.xml	add	`<kDef key="kaA3" kId="man1.xml" kKind="technical M."/>`
	man1.xml	add	`<man kind="technical M."> ...`
2	man1.xml	change	`<man kind="field M."> ...`
3	doc2.txt	add	`...as shown in manual kaA2 ...`
4	man2.xml	add	`<man kind="field M."> ...`
	man3.xml	add	`<man kind="field M."> ...`
	man4.xml	add	`<man kind="field M."> ...`

Table 8.1: Example repository up to state 4

Figure 8.3: S-DAG for rule ϕ_1 at state 4

At state 4, we generate the S-DAG shown in Fig. 8.3 (for the development of the repository see Tab. 8.1). In the leaves, we find atomic formulae responsible for inconsistencies. For example, the right hand leaf indicates that the atomic formula $\texttt{kind}(m) = \texttt{kKind}(d)$ is violated, if m is bound to the manual man1.xml and d is bound to the key definition for kaA3. We can grasp these bindings by following the paths from the S-DAG root to this leaf.

The S-DAG in Fig. 8.3 lacks concrete repair actions, i.e., to add, change, or delete document content. In order to support interactive repair, we augment S-DAGs by actions after consistency checking. Fig. 8.4 shows the augmented S-DAG for rule ϕ_1 at state 4 (where we consider inconsistencies at state 4 only). Quantifier edges carry actions marked grey. An action proposes to either add a value to (Add), or change a value within (Chg), or delete a value from the sphere of this quantifier (Del). An action KEEP indicates that the value must

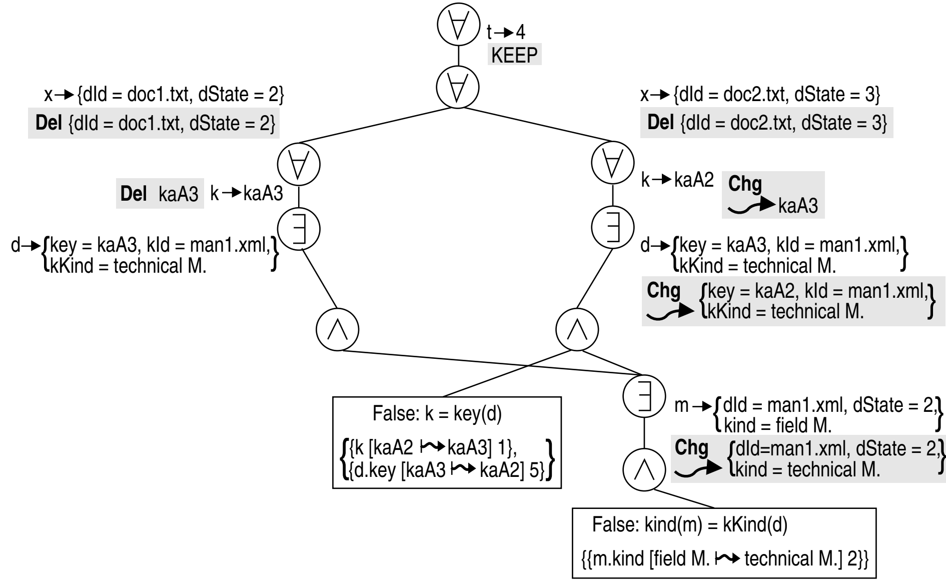


Figure 8.4: Augmented S-DAG for rule ϕ_1 at state 4

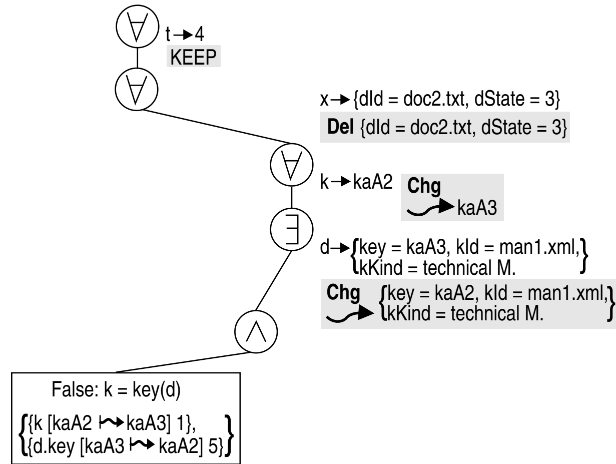


Figure 8.5: Augmented S-DAG resulting from changing the kind for the manual man1.xml towards technical M.

not be changed. For example, we propose to change the key kaA2 to kaA3 or to delete the document doc1.txt. From an augmented S-DAG, authors can choose actions interactively. Notice that, still, it is unclear which actions in an S-DAG must be applied together, in order to resolve the inconsistencies. For example, an author may choose to change the kind of manual man1.xml towards technical M. Applying this action to the S-DAG in Fig. 8.4 results in the S-DAG shown in Fig. 8.5. The remaining inconsistency can be resolved by choosing another action, e.g., by changing the key reference kaA2 to kaA3.

Notice that the existential quantifier for m does *not* contain edges for the manuals man2.xml through man4.xml. We regard changes to these manuals too expensive; instead, we propose to change the kind of the manual man1.xml only. Recall that we do not aim to generate the complete set of all repairs but to present a few repairs to authors. We consider this as a major benefit of our approach, as compared to alternative approaches towards interactive repair, e.g., [NEF03].

Compare our S-DAGs with the diagnoses from Part I. From a diagnosis

$$\begin{pmatrix} \mathsf{IC}, \left\{ t \mapsto 4, k \mapsto \mathsf{kaA3}, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 1\} \right\}, \\ \emptyset, \{\mathtt{kind}(m) = \mathtt{kKind}(d)\} \end{pmatrix}$$

authors learn about an inconsistent key reference. But they have to find out resolution actions by themselves, which requires good knowledge about document structures and the consistency rule. With an augmented S-DAG at hand, authors can *choose* the right action for repair. On the other hand, rule designers must annotate rules by hints. We shift some of the responsibilities for resolving inconsistencies from authors to rule designers. We argue that this step is natural, because rule designers know the rules better than authors do. Also, hints are defined only once together with the rules.

In the next section, we detail how rule designers formalize hints. Sect. 8.3 is dedicated to S-DAGs. We define our (incremental) S-DAG generation algorithm in Sect. 8.4. Interactive repair is discussed in Sect. 8.5.

8.2 Hints for Better Repair Actions

Actions are derived from predicate suggestions, which in turn are generated from truth values of atomic formulae. A predicate suggestion indicates how the truth value of an atomic formula should be changed. For example, the key reference to kaA2 causes violation of the atomic formula $k = \mathtt{key}(d)$ for the variable assignment $\{k \mapsto \mathsf{kaA2}, d \mapsto \{\mathtt{key} = \mathsf{kaA3}, \mathtt{kId} = \mathsf{man1.xml}, \mathtt{kKind} = \mathsf{technical\ M.}\}\}$. Then we can generate two predicate suggestions:

Change k from kaA2 to kaA3. or
In d change the field `key` from kaA3 to kaA2.

Clearly, both alternatives are plausible, because they invert the truth value of $k = \mathtt{key}(d)$ under the given assignment. In some situations, we know that inconsistencies of the above kind can result purely from a wrong key reference. Then we would like to generate the first alternative only. For arbitrary predicate symbols, however, predicate suggestions are not that obvious.

8.2.1 Formalizing Hints

A hint is formalized through a term and has one of the two forms shown in Fig. 8.6. In a hint $x \leadsto e\ b\ k\ c$, x is a quantified variable, e is a term that calculates the new value for x, and b is the truth value of the atomic formula,

Annotated formulae $\mathcal{F}^{\mathcal{H}}$	
$\phi, \psi ::= p(e_1, \ldots, e_n) \left\{\overline{\{\overline{h}\}}\right\}$	atomic formula with hints
$\mid \ \phi \cdot \psi \mid \neg\phi$	junction ($\cdot \in \{\vee, \wedge, \Rightarrow\}$); negation
$\mid \ Q\ x^{[a]} \in e \bullet \phi$	quantified formula ($Q \in \{\forall, \exists\}$), annotation a
Hints $\mathcal{H}$	
$h \quad ::= x \rightsquigarrow e\ b\ [k]\ [c]$	set x to e and assign the cost c if the result of the atomic formula is b and the kind of x is k
$\mid \ x.l \rightsquigarrow e\ b\ [k]\ [c]$	set the label l in x to e and assign the cost c if the formula's result is b and the kind of x is k
Quantifier annotation	
$a \quad ::= \mathsf{KEEP} \mid \mathsf{CHG}$	do not repair; force change

Figure 8.6: Abstract syntax of annotated formulae $\mathcal{F}^{\mathcal{H}}$ and hints $\mathcal{H}$

the truth value of which should be inverted to the opposite of b. The optional parameter k determines whether this hint applies to new and old variables, respectively. The optional parameter c assigns a cost to a hint; our simple cost model comprises natural numbers only. Costs are useful for reducing S-DAGs (see Sect. 8.4.5) and rating repairs (see Chapter 9). By costs, the rule designer can gauge potential negative impacts of a repair. If the type of x is a record type, we support a reduced hint $x.l \rightsquigarrow e\ b\ k\ c$, which applies to a record label l of x. In the term e, the rule designer can use any quantified variable and any function symbol (defined by the language designer). A reference to a variable means the variable's value prior to evaluating the hint. Thus, we can set the new value of a variable depending on its current value, which is responsible for the violation. Hints form a collection. Each hint set is an alternative. Within a hint set, all hints are evaluated simultaneously. We denote the set of all hints by $\mathcal{H}$ and the set of all annotated formulae by $\mathcal{F}^{\mathcal{H}}$.

Quantifier annotations provide further guidance by limiting the kinds of repair actions derived for quantifier variables. This is important for S-DAG augmentation (see Sect. 8.5.1) and repair derivation (see Chapter 9). There, a quantifier variable annotated by KEEP is ignored. For a quantifier variable, annotated by CHG, we only derive repairs to change the variable.

Incremental consistency checking distinguishes between new (or changed) and old variables. Rule designers can exploit these information in hints. Thus, they can "blame" new (or changed) documents for inconsistencies. Consider the following variant of ϕ_1:

$$\begin{array}{lll}
\phi_1 = & \forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet & \\
& \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet & \\
& \quad k = \texttt{key}(d) \quad \left\{ \begin{array}{l} \{k \rightsquigarrow \texttt{key}(d)\ \mathsf{False}\ \mathsf{new}\ 1\}, \\ \{d.\texttt{key} \rightsquigarrow k\ \mathsf{False}\ \mathsf{new}\ 5\} \end{array} \right\} & \wedge \\
& \exists\, m \in \texttt{repManDs}(t) \bullet & \\
& \quad \texttt{dId}(m) = \texttt{kId}(d) \quad \{\{m.\texttt{dId} \rightsquigarrow \texttt{kId}(d)\ \mathsf{False}\ 3\}\} & \wedge \\
& \quad \texttt{kind}(m) = \texttt{kKind}(d) \quad \{\{m.\texttt{kind} \rightsquigarrow \texttt{kKind}(d)\ \mathsf{False}\ 2\}\} &
\end{array}$$

If either of the variables k and d is modified, then we evaluate *one* hint only. If both variables k and d are modified, then we evaluate both hints. The case

that both variables k and d are marked as old cannot occur (during incremental consistency checking) since then the atomic formula is not evaluated (instead, part of the old S-DAG is copied).

Hints are useful for flexible inconsistency handling strategies. For example, a hint can depend on the current repository state (at which the inconsistency occurred) or the content of a specific document. That way we can react differently to an inconsistency depending on the development state of a document engineering project.

Our static type checker ensures well-typedness of hints, such that predicate suggestions are compatible with the document structure. Of course, this is guaranteed only if the document type resembles the document structure correctly. Our consistency checker evaluates hints and generates the corresponding predicate suggestions. If no hint is given, we generate a predicate suggestion that proposes to invert the truth value of the atomic formula. Predicate suggestions do not indicate whether authors should delete, change, or add document content. These indications are added during S-DAG augmentation; see Sect. 8.5.

8.2.2 Type Checking Hints

Of course, hints must be type checked. Since they are formalized with the help of terms, type checking is straightforward. For a hint $x \rightsquigarrow e\ b\ k\ c$, we require that x is in scope, e is a well typed term, and x has the same type as e (type checking ignores the boolean value b, the variable kind k, and the cost c). For a hint $x.l \rightsquigarrow e\ b\ k\ c$, we require that x is in scope, e is well typed, l is a valid label in the record type of x (i.e., l is a member of the record label environment of x's type), and e's type equals the result type of l. Fig. 8.7 shows the new well-typedness rules and the new inference rules for our type checking algorithm. The judgements follow the shape of judgements for formulae (see Sect. 4.3.2 and 4.3.3). In the type inference rule ChkHint, we expect the instantiated type $\theta_x \nu$ of x when inferring the type of e. Similarly, in the type inference rule ChkHintField we expect the instantiated result type $\theta_x \tau_{g_l}$ of l when inferring the type of e.

We consider type checking of hints a crucial aspect, because generated repair actions must be compatible with the document structure, i.e., applying an action to a document does not invalidate its structure (e.g., a DTD or a XML Schema). Static type checking is the first step towards this property. We can guarantee that repairs, proposing to change content, are compatible with the document structure. For a more detailed discussion see Sect. 8.5.3.

In the next section, we show how S-DAGs can be used to store predicate suggestions space efficiently and to describe repair actions. A major feature of our approach is to *reduce* S-DAGs, in order to abandon redundant actions.

Well typedness rules for hints

$$\frac{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ x : \tau_g \qquad C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \tau_g}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ x\ \rightsquigarrow\ e\ b\ k\ c}\quad \text{TypHint}$$

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ x : \tau_g \qquad \tau_g = R\ \tau_{g_1} \ldots \tau_{g_n}\\ l : \tau_g \rightarrow \tau_{g_l} \in \Delta \qquad l \in \widehat{R}\\ C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e : \tau_{g_l}\end{array}}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ x.l\ \rightsquigarrow\ e\ b\ k\ c}\quad \text{TypHintField}$$

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ p : \tau_1 \times \ldots \times \tau_n \rightarrow \mathsf{Bool}\\ C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ e_i : \tau_i \qquad C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ h_{i,j}\end{array}}{C,\ \Gamma,\ \Delta\ \vdash_{\mathbf{S}}\ p(e_1, \ldots, e_n)\ \{\{\overline{\overline{h_{i,j}}}\}\}}\quad \text{TypPredApp}$$

Type inference rules for hints

$$\frac{C_x,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ x : \theta_x(\nu) \qquad C_e,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ e : \theta_e(\theta_x \nu)}{\theta_e C_x \cup C_e,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ x\ \rightsquigarrow\ e\ b\ k\ c, \theta_e}\quad \text{ChkHint}$$

$$\frac{\begin{array}{c}C_x,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ x : \theta_x(\nu) \qquad \theta_x \nu = R\ \tau_{g_1} \ldots \tau_{g_n}\\ l : \tau_g \rightarrow \tau_{g_l} \in \Delta \qquad l \in \widehat{R}\\ C_e,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ e : \theta_e(\theta_x \tau_{g_l})\end{array}}{\theta_e C_x \cup C_e,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ x.l\ \rightsquigarrow\ e\ b\ k\ c, \theta_e}\quad \text{ChkHintField}$$

$$\frac{\begin{array}{c}C_i,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ e_i : \theta_i(\nu_i)\\ C,\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ p : \theta(\theta_1 \nu_1 \times \ldots \times \theta_n \nu_n \rightarrow \mathsf{Bool})\\ C_{i,j},\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\ h_{i,j}, \theta_{i,j}\end{array}}{\begin{array}{l}\theta_{i,j} C \cup \bigcup \theta \theta_{i,j} C_i \cup \theta \bigcup C_{i,j},\ \Gamma,\ \Delta\ \Vdash_{\mathbf{S}}\\ \qquad p(e_1, \ldots, e_n)\ \{\{\overline{\overline{h_{i,j}}}\}\},\ \theta \circ \theta_{i,j}\end{array}}\quad \text{ChkPredApp}$$

Figure 8.7: Typing hints

8.3 Describing Repair Actions by S-DAGs

Describing repair actions by S-DAGs has three major advantages:

- Generating S-DAGs is faster than enumerating repair collections. S-DAGs can be reduced during their generation, in order to abandon redundant actions. Also, S-DAGs are well suited to incremental generation.
- S-DAGs visualize inconsistencies and possible repair actions in a convenient way to authors. Sharing of nodes avoids redundancies and also helps authors to anticipate the effect of an action.
- From S-DAGs, we can derive a repair collection on demand, which is done *after* the actual consistency check; see Chapter 9.

First, we show how S-DAGs can be used to store predicate suggestions, thereby describing repair actions. In Sect. 8.3.2, we describe the rationale behind S-DAG reduction. A formal definition of S-DAGs can be found in Sect. 8.4.1.

Figure 8.8: Complete S-DAG for rule ϕ_1

8.3.1 What is an S-DAG?

The structure of an S-DAG resembles that of a consistency rule. Nodes represent logical connectives or atomic formulae; edges target the subformulae of a connective. We distinguish six kinds of nodes.

Universal nodes (∀) and existential nodes (∃) represent universal and existential quantification, respectively. An outgoing edge carries a value binding to the quantified variable and actions. A value represents a repository state, a document, or document content. Actions are added by S-DAG augmentation; they indicate how inconsistencies can be resolved. Conjunction nodes (∧) and disjunction nodes (∨) stand for conjunctions and disjunctions, respectively. A predicate node occurs as a leaf only; it contains (1) an atomic formula ϕ responsible for an inconsistency, (2) ϕ's truth value, and (3) a predicate suggestion collection, which indicates how the truth value of ϕ can be inverted. A leaf Abandoned stands for an S-DAG that has been reduced, because it contains repairs that are redundant or too expensive. Notice that by reducing S-DAGs we eliminate repairs that are not useful; S-DAGs contain a subset of the complete set of all repairs. S-DAGs do not include implication nodes, because formulae are miniscoped prior to S-DAG generation; negation nodes are omitted.

Fig. 8.8 shows the complete S-DAG for rule ϕ_1 at state 4. In the leaves, we find predicate suggestions for atomic formulae. For example, the bottom

leaf indicates that the atomic formula $\texttt{kind}(m) = \texttt{kKind}(d)$ is violated, if m is bound to one of the manuals man1.xml, man2.xml, man3.xml, or man4.xml, and d is bound to the key definition for kaA3. The predicate suggestion $m.\texttt{kind}$ [field M. $\rightsquigarrow$ technical M.] 2 proposes to change the kind of the manual m to technical M. It results from evaluating the hint $m.\texttt{kind} \rightsquigarrow \texttt{kKind}(d)$ False 2 under the following assignments:

$$\{d \mapsto \{\texttt{kKind} = \text{technical M.}, \ldots\}, m \mapsto \{\texttt{dId} = \text{man1.xml}, \texttt{kind} = \text{field M.}\}, \ldots\}$$
$$\{d \mapsto \{\texttt{kKind} = \text{technical M.}, \ldots\}, m \mapsto \{\texttt{dId} = \text{man2.xml}, \texttt{kind} = \text{field M.}\}, \ldots\}$$
$$\{d \mapsto \{\texttt{kKind} = \text{technical M.}, \ldots\}, m \mapsto \{\texttt{dId} = \text{man3.xml}, \texttt{kind} = \text{field M.}\}, \ldots\}$$
$$\{d \mapsto \{\texttt{kKind} = \text{technical M.}, \ldots\}, m \mapsto \{\texttt{dId} = \text{man4.xml}, \texttt{kind} = \text{field M.}\}, \ldots\}$$

At the current stage, S-DAGs lack information about concrete repair actions, i.e., adding, changing, or deleting values. S-DAGs will be augmented with actions after the consistency check (see Sect. 8.5.1). Separating the generation of S-DAGs from their augmentation speeds up consistency checking. This is vital, because the repository must be locked during consistency checks.

The S-DAG in Fig. 8.8 is considerably larger than the S-DAG in Fig. 8.3 (pg. 99). In an S-DAG, we have to store every possible binding below an existential node, because we do not know which edge should be repaired. Consequently, S-DAGs may grow really large.

8.3.2 Reducing S-DAGs

We can, however, *reduce* S-DAGs by removing redundant parts. Recall that our major design decision is to generate repairs that require *small* changes to the repository. Currently, this decision cannot be overridden by any annotations of the rule designer.

In our example, one would resolve the inconsistency introduced by the wrong kind of man1.xml rather by changing the kind of man1.xml than by changing the name *and* the kind of one of the other manuals. Intuitively, the S-DAG for man1.xml requires less changes to the repository than the S-DAGs for the other manuals. Thus, in Fig. 8.8, we replace the target S-DAGs of the edges for man2.xml, man3.xml, and man4.xml by Abandoned (see Fig. 8.9). In order to save space, we replace all edges targeting Abandoned by one edge carrying a binding to a dummy value $*$, which intuitively means “all other values in the quantifier sphere.” Fig. 8.9 shows the benefits of this edge reduction. Finally, we arrive at the S-DAG shown in Fig. 8.10. The S-DAG differs from the S-DAG in Fig. 8.3, because we cannot share the S-DAGs below the edges labeled $t \mapsto 3$ and $t \mapsto 4$. The (additional) dummy S-DAG Abandoned is necessary for incremental S-DAG generation.

Formally, we have defined an ordering relation $\prec$ for S-DAGs. By this partial strict order, an S-DAG d_1 is smaller than an S-DAG d_2 iff all leaves in d_1 are also contained by d_2 and d_1 represents less inconsistencies d_2.[4] Then, for existential nodes and disjunction nodes we retain the smallest S-DAGs below them and replace the other S-DAGs by Abandoned. In general, there will be

[4] Of course, $\prec$ respects the structure of S-DAGs.

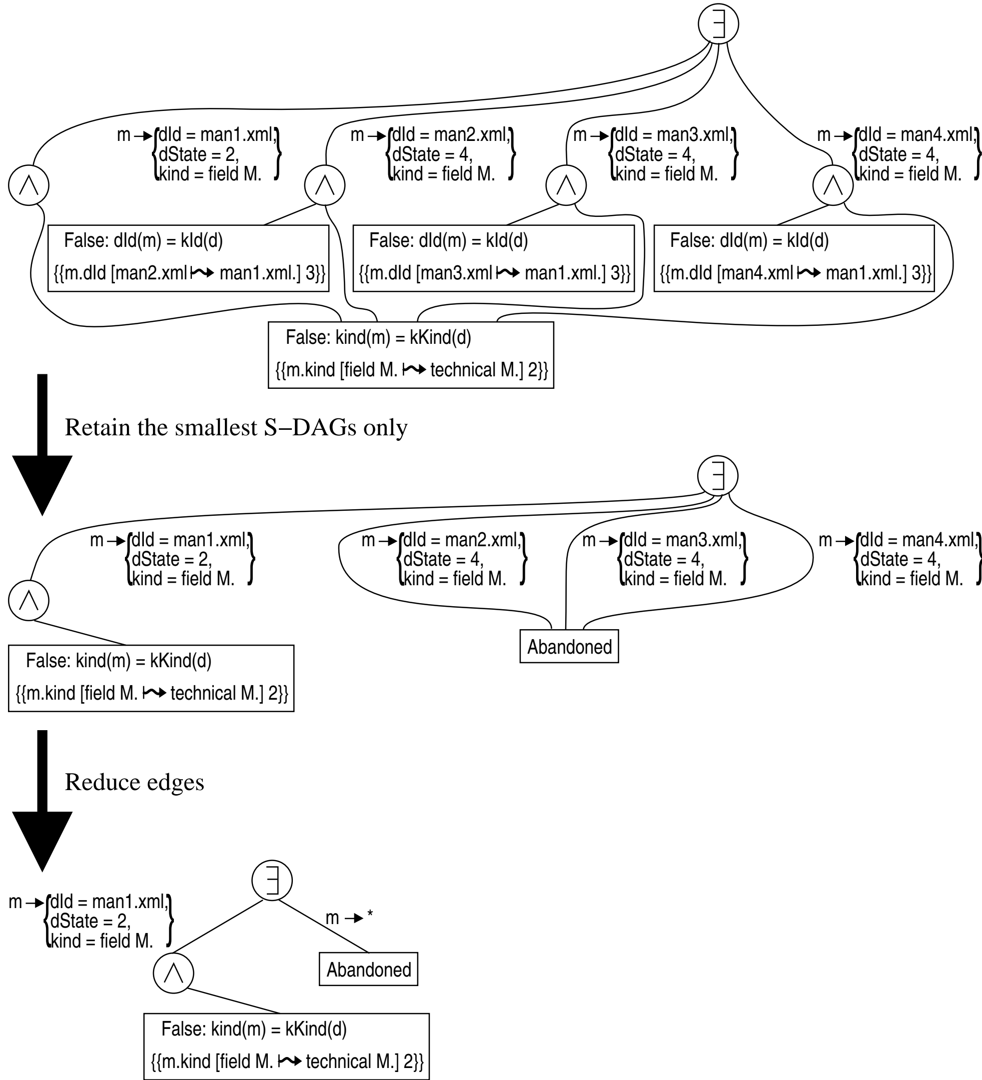


Figure 8.9: Reducing the existential node for m at state 4

many smallest S-DAGs, because $\prec$ is partial and does not necessarily induce a lattice [DP90] on the S-DAGs below a node. Clearly, it is sufficient to repair one S-DAG below an existential node or a disjunction node. The smallest S-DAGs represent repairs that require the least changes to the repository and are thus most likely to be chosen. In contrast, for conjunction nodes, we keep the greatest S-DAGs below them, i.e., we drop those S-DAGs that are subsumed by other S-DAGs. Intuitively, we must repair all S-DAGs below a conjunction node. Assume that we must repair two S-DAGs d_1 and d_2, where $d_1 \prec d_2$; then it is sufficient to repair d_2 only, because d_2 already represents all inconsistencies from d_1.

Notice that by the above reduction strategy we *loose expensive repairs*, which is important, because authors apply repairs manually. Next, we define two S-DAG generation algorithms.

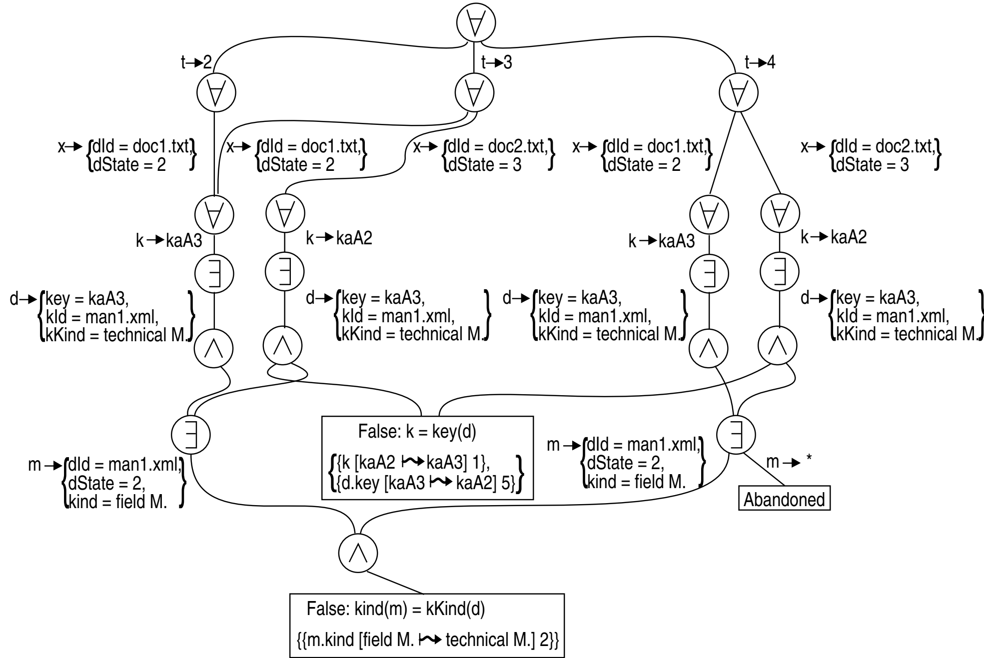


Figure 8.10: Reduced S-DAG for rule ϕ_1 at state 4

8.4 Generating S-DAGs

Our S-DAG generation algorithms are defined by structural induction on a formula. They create nodes for violated subformulae (similar to the report generation algorithms in Sect. 5.3 and 6.3.2). An S-DAG node is built in constant time, as opposed to the computation of cartesian products and unions of diagnoses sets. For each subformula, we reduce the resulting S-DAG, as explained above. For atomic formulae, we evaluate hints. Our algorithms require that negations have been pushed into the formula to process. This is achieved by miniscoping (see Sect. 6.2.2).

Next, we introduce formal notations for S-DAGs, predicate suggestions, and repair actions. In Sect. 8.4.2, we define a basic S-DAG generation algorithm; Sect. 8.4.3 is devoted to incremental S-DAG generation. We illustrate both algorithms in Sect. 8.4.4 by our running example. In Sect. 8.4.5, we define ordering relations for S-DAGs and auxiliary functions.

8.4.1 Preliminaries: S-DAGs, Predicate Suggestions, Repair Actions

Fig. 8.11 shows the abstract syntax of S-DAGs as already explained. We let $\bigcirc$ denote an empty S-DAG. Below conjunction nodes and disjunction nodes, respectively, we label edges by the number of the corresponding subformula.

A predicate suggestion indicates how the truth value of an atomic formula can be inverted. We distinguish between changing the value v_1 of a variable x to v_2, changing the value v_1 of a component l of a variable's value to v_2 (if the

S-DAGs $\mathbb{G}$	
$d \quad ::= \bigcirc$	empty S-DAG
$\quad\mid$ Abandoned	abandoned S-DAG
$\quad\mid \boxed{\phi_{\text{at}}\ b\ \left\{\overline{\{\overline{psug}\}}\right\}}$	predicate leaf, predicate suggestion *psug*
$\quad\mid \odot\ \left\{\overline{(n, d)}\right\}$	junction node; edges with child S-DAG d, edge number n corresponds to subformula number
$\quad\mid \mathbb{Q}\ x\ \left\{\overline{(v, \{\overline{act}\}, d)}\right\}$	quantifier node, variable x; edges with bound value v, actions $\overline{act}$, child S-DAG d
Predicate suggestions $\mathbb{S}$	
$psug ::= x\ [v_1 \rightsquigarrow v_2]\ c$	change the value of x from v_1 to v_2, cost c
$\quad\mid x.f\ [v_1 \rightsquigarrow v_2]\ c$	change value of component f from v_1 to v_2, cost c
$\quad\mid x\ [b:\ \phi_{\text{at}}]$	change x, such that the truth value of ϕ_{at} is b
$\quad\mid x\ [???]$	contradiction: don't know how to change x
Repair actions $\mathbb{C}$	
$act \ ::=$ Add v | Del v	add the value v; delete the value v
$\quad\mid$ Chg $v_1 \rightsquigarrow v_2$	change the value v_1 to v_2
$\quad\mid$ KEEP	do not change anything

Figure 8.11: Abstract syntax of S-DAGs $\mathbb{G}$, predicate suggestions $\mathbb{S}$, and repair actions $\mathbb{C}$

variable has a record type), and inverting the truth value of the atomic formula. Whereas the first two suggestions are generated from hints, the last suggestion is generated, if no hints are given. In addition, we provide a "fall-back" case, which is generated, if a hint proposes to change a variable annotated by KEEP.

In an S-DAG, a quantifier edge carries a set of actions, which propose to either add a value to (Add), or change a value within (Chg), or delete a value from the sphere of this quantifier (Del). A KEEP action proposes to retain the sphere. During consistency checking, we generate S-DAGs without actions; they are added to quantifier edges by S-DAG augmentation after consistency checking (see Sect. 8.5.1).

We denote the set of all S-DAGs by $\mathbb{G}$, the set of all predicate suggestions by $\mathbb{S}$, and the set of all actions by $\mathbb{C}$.

8.4.2 A Basic S-DAG Generation Algorithm

Fig. 8.12 shows the denotational semantics of our basic S-DAG generation algorithm, which corresponds to non-incremental report generation shown in Sect. 5.3; for the definition of auxiliary functions see Sect. 8.4.5. $\mathcal{D}_A^{(kept,neg)}[\![\phi]\!]\eta : \mathbb{A} \times (\wp(\mathcal{X}) \times \mathbb{B}) \times \mathcal{F}^{\mathcal{H}} \times \mathbb{E} \rightarrow \mathbb{G}$ generates the S-DAG for the formula ϕ w.r.t. the structure A and the variable assignment η (we use the non-incremental definition from Fig. 5.9 on pg. 61). We collect variables annotated by KEEP using the additional parameter *kept* — for these variables we do not generate suggestions. The parameter *neg* indicates whether the formula ϕ appears in a negated context; then we have $neg =$ True. We evaluate hints only, if an atomic formula can be responsible for an inconsistency. Initially,

$$
\begin{array}{lll}
\mathcal{D}_A^{(kept,neg)}[\![p(e_1,\ldots,e_n)\ hss]\!]\eta = & \owedge\ undefs & \text{if any } e_i^A \text{ is not defined} \\
& leaf & \text{else if } neg = b \\
& \bigcirc & \text{otherwise}
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & b & = \mathsf{True} & \text{if all } e_i^A \text{ are defined and } (e_1^A,\ldots,e_n^A) \in p^A \\
& & \quad \mathsf{False} & \text{otherwise} \\
& e_i^A & = s_i^A & \text{if } e_i \equiv s_i \\
& & \quad \mathcal{V}_A[\![e_i]\!]\eta & \text{otherwise} \\
& sugss & \multicolumn{2}{l}{= \mathcal{S}_A^{kept}[\![(\mathsf{filterHints}(hss,b,\eta), p(e_1,\ldots,e_n), b)]\!]\eta} \\
& leaf & \multicolumn{2}{l}{= \boxed{p(e_1,\ldots,e_n)\ b\ sugss}} \\
& undefs & \multicolumn{2}{l}{= \left\{ \left(i, \boxed{\downarrow(e_i)\ \mathsf{False}\ \emptyset}\right) \;\middle|\; e_i^A \text{ is not defined} \right\} \cup \{(n+1, leaf)\}}
\end{array}
$$

$$
\mathcal{D}_A^{(kept,neg)}[\![\neg\phi]\!]\eta = \mathcal{D}_A^{(kept,\neg neg)}[\![\phi]\!]\eta
$$

$$
\begin{array}{lll}
\mathcal{D}_A^{(kept,neg)}[\![\phi \wedge \psi]\!]\eta = & \mathsf{reduce}_\wedge(d_\phi, d_\psi) & \text{if } d_\phi \neq \bigcirc \text{ and } d_\psi \neq \bigcirc \\
& \owedge\ \{(1, d_\phi)\} & \text{else if } d_\phi \neq \bigcirc \\
& \owedge\ \{(2, d_\psi)\} & \text{else if } d_\psi \neq \bigcirc \\
& \bigcirc & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{where} & d_\phi & = \mathcal{D}_A^{(kept,neg)}[\![\phi]\!]\eta \\
& d_\psi & = \mathcal{D}_A^{(kept,neg)}[\![\psi]\!]\eta
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D}_A^{(kept,neg)}[\![\phi \vee \psi]\!]\eta = & \mathsf{reduce}_\vee(d_\phi, d_\psi) & \text{if } d_\phi \neq \bigcirc \text{ and } d_\psi \neq \bigcirc \\
& \bigcirc & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{where} & d_\phi & = \mathcal{D}_A^{(kept,neg)}[\![\phi]\!]\eta \\
& d_\psi & = \mathcal{D}_A^{(kept,neg)}[\![\psi]\!]\eta
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D}_A^{(kept,neg)}[\![\forall\ x^k \in e \bullet \phi]\!]\eta = & \boxed{\downarrow(e)\ \mathsf{False}\ \emptyset} & \text{if } \mathcal{V}_A[\![e]\!]\eta \text{ is not defined} \\
& \textcircled{\scriptsize$\forall$}\ x\ F & \text{else if } F \neq \emptyset \\
& \bigcirc & \text{otherwise}
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & kept' & = \{x\} \cup kept & \text{if } k = \mathsf{KEEP} \\
& & \quad kept & \text{otherwise} \\
& F & \multicolumn{2}{l}{= \{(v, \emptyset, d) \mid v \in \mathcal{V}_A[\![e]\!]\eta \text{ and } d \neq \bigcirc\}} \\
& & \multicolumn{2}{l}{\qquad \text{where } d = \mathcal{D}_A^{(kept',neg)}[\![\phi]\!](\eta \cup \{x \mapsto v\})}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D}_A^{(kept,neg)}[\![\exists\ x^k \in e \bullet \phi]\!]\eta = & \bigcirc & \text{if } \mathcal{V}_A[\![e]\!]\eta \text{ is not defined} \\
& \boxed{\mathtt{null}(e)\ \mathsf{True}\ \emptyset} & \text{else if } \mathcal{V}_A[\![e]\!]\eta = [] \\
& \mathsf{reduce}_\exists(x, F) & \text{else if } |F| = |\mathcal{V}_A[\![e]\!]\eta| \\
& \bigcirc & \text{otherwise}
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & kept' & = \{x\} \cup kept & \text{if } k = \mathsf{KEEP} \\
& & \quad kept & \text{otherwise} \\
& F & \multicolumn{2}{l}{= \{(v, \emptyset, d) \mid v \in \mathcal{V}_A[\![e]\!]\eta \text{ and } d \neq \bigcirc\}} \\
& & \multicolumn{2}{l}{\qquad \text{where } d = \mathcal{D}_A^{(kept',neg)}[\![\phi]\!](\eta \cup \{x \mapsto v\})}
\end{array}
$$

Figure 8.12: A basic S-DAG generation algorithm

$\mathcal{D}$ is applied to an empty variable set and an empty variable assignment, i.e., $\mathcal{D}_A^{(\emptyset,\mathsf{False})}[\![\phi]\!]\emptyset$.

For an atomic formula, we first determine its truth value b as usual. The formula is potentially responsible for an inconsistency only if it appears in a positive context and b is False, or if it appears in a negated context and b is True; i.e., $neg = b$. In that case, we generate a leaf carrying the atomic formula, its truth value b, and the predicate suggestion collection resulting from evaluating the hint collection hss. Otherwise, we return an empty S-DAG, which means that this formula is fulfilled for the current assignment. We obtain the hints that correspond to b via filterHints.[5] They are evaluated by the auxiliary function $\mathcal{S}$, defined in Sect. 8.4.5. $\mathcal{S}$ is total, i.e., it drops hints that cannot be evaluated due to undefinedness. If any argument term e_i is not defined, we generate a conjunction node. This node carries the leaf for the violated atomic formula and leafs that indicate which terms are undefined.

For a conjunction, we first generate the S-DAGs for its subformulae. We keep the S-DAGs of violated subformulae and assign a number to them: 1 means that the left hand side of the conjunction is violated, 2 means that the right hand side of the conjunction is violated.[6] If both subformulae are violated, we reduce their S-DAGs via $\mathsf{reduce}_\wedge$, which generates a conjunction node. Applied to two S-DAGs $\mathsf{reduce}_\wedge$ keeps the greater S-DAG, if both S-DAGs are comparable. Otherwise, $\mathsf{reduce}_\wedge$ keeps both S-DAGs. If both subformulae are fulfilled, we return an empty S-DAG. For disjunctions, we use a similar procedure but create a non-empty S-DAG only if both subformulae are violated. We reduce the S-DAGs of violated subformulae by $\mathsf{reduce}_\vee$, which keeps the smallest S-DAGs and generates a disjunction node.

For a universally quantified formula, we first evaluate its subformula w.r.t. possible variable assignment extensions $\eta \cup \{x \mapsto v\}$ (where the value v is a member of the quantifier sphere $\mathcal{V}_A[\![e]\!]\eta$). We store S-DAGs of violated subformulae and their corresponding values in the set F. The values represent edge labels in the S-DAG. If F is not empty, then the universally quantified formula is violated; we create a universal node carrying the set F. Equal S-DAGs are shared "automatically" due to the nature of DAGs. If the sphere term e is not defined, we return a leaf indicating that it is undefinedness of e, which causes an inconsistency. For existentially quantified formulae, we use a similar approach but reduce F by $\mathsf{reduce}_\exists$, which keeps the smallest S-DAGs and replaces edges targeting Abandoned by a dummy. If the sphere of an existentially quantified formula is empty, then we generate a predicate leaf indicating that emptiness of the sphere is responsible for the violation (this is similar to consistency report generation).

Similar to report generation, we generate a non-empty S-DAG for a formula ϕ, if and only if ϕ is violated. The following theorem expresses this property.

[5] For the moment, we ignore variable kinds for hints — kinds are important for incremental S-DAG generation only.

[6] We use natural numbers, because conjunctions and disjunctions can carry more than two subformulae due to flattening. In our presentation of the algorithm, however, we consider binary conjunctions and disjunctions only. We do so for better comprehensibility.

Theorem 8.1 (S-DAGs indicate real inconsistencies) Let ϕ be a consistency rule and A a first-order structure. Then we have:

$$\text{not } A \models_{\emptyset} \phi \quad \Longleftrightarrow \quad \mathcal{D}_A^{(_,\mathsf{False})}[\![\phi]\!]\emptyset \neq \bigcirc$$

Proof: The proof proceeds by straightforward induction on the structure of the consistency rule ϕ; see App. C, Proof C.4. □

Notice that we cannot prove that in a leaf the predicate suggestions for a formula ϕ really invert the truth value of ϕ, under the assignment given by the path from the S-DAG root to this leaf. The reason is that function symbols, used in hints, are defined in Haskell, a full programming language. Therefore, there is no guarantee that a predicate suggestion really inverts the truth value of an atomic formula.

Although S-DAG combination is considerably faster than combination of consistency reports, our basic S-DAG generation algorithm shows poor performance (comparable to non-incremental report generation). Still, every quantifier sphere contributes a polynomial factor; the nesting of quantifiers results in an even exponential computation time behavior. Evaluation of hints causes additional computation costs. For S-DAG generation, we already use static analysis: miniscoping and rule filtering (see Sect. 6.2). In the next section, we adapt the incremental techniques from Sect. 6.3 to our basic S-DAG generation algorithm. In fact, copying of old S-DAGs is cheaper than the copying of old diagnoses, because we can now exploit the structure of S-DAGs.

8.4.3 An Incremental S-DAG Generation Algorithm

Recall our basic ideas for incremental report generation from Sect. 6.3. They directly carry over to incremental S-DAG generation:

- We copy those parts from previous S-DAGs that have not changed.
- We partition quantifier spheres into four sets: *new*, *chg*, *old*, and *del*. Variable assignments mark variables as new and old, respectively. Our incremental algorithm deviates from brute force S-DAG generation mostly in the treatment of quantified formulae.
- Due to miniscoping, only atomic formulae can appear in a negated context. In particular, existential quantifiers cannot "disguise" as universal quantifiers and vice versa. This already simplified our basic S-DAG generation algorithm.

Fig. 8.14 shows the denotational semantics of our incremental S-DAG generator $\mathcal{ID}$, which works similar to the incremental report generator $\mathcal{IR}$, defined in Fig. 6.9 (pg. 81). The function $\mathcal{ID}_A^{(kept,neg)}[\![\phi]\!]\eta : \mathbb{A} \times (\wp(\mathcal{X}) \times \mathbb{B}) \times \mathcal{F}^{\mathcal{H}} \times \mathbb{E}_{\text{incDAG}} \rightarrow \mathbb{G}$ is defined by structural induction on a formula ϕ (like brute force S-DAG generation). For readability, we introduce the global variable Ⓓ, which represents the old S-DAG. Superscripts denote the free variables of a

η	$::=$	$\langle b_1 : b_2 : \ldots : b_n\rangle$	incr. assignment for S-DAG generation (sequence)
b	$::=$	$x \mapsto (v, k)$	incr. variable binding (x variable, v value, k kind)
	$\mid$	$1 \mid 2 \mid \ldots$	junction binding

Figure 8.13: Variable assignments $\mathbb{E}_{\text{incDAG}}$ for incremental S-DAG generation

formula; ϕ^{xs} means that the variables from the set xs are free in ϕ. Notice that xs now also contains variables used in hints. Initially, $\mathcal{ID}$ is applied to an empty variable set and an empty variable assignment, i.e., $\mathcal{D}_A^{(\emptyset,\mathsf{False})}[\![\phi]\!]\langle\rangle$. For formal definitions of new auxiliary functions see Sect. 8.4.5.

For every formula, $\mathsf{notEval}(xs, \eta)$ determines whether (1) its free variables in the set xs are marked as old in the current assignment η and (2) it contains referentially transparent functions and predicates only. In this case, we copy the relevant part from the old S-DAG Ⓓ. Copying part of the old S-DAG requires to navigate through it. In order to identify the relevant part of the old S-DAG efficiently, we change the structure of variable assignments, as shown in Fig. 8.13. First, we also store *junction bindings* in variable assignments, because we need to identify the sub-DAGs of conjunction and disjunction nodes, respectively.[7] The natural number of a junction binding corresponds to the position of a subformula in a conjunction or disjunction. Second, we represent an assignment by a *sequence*, instead of a set, because the order of bindings is important for S-DAG navigation.[8] We denote an assignment extension of η by $\langle \eta : b\rangle$, which means that the binding b is added to the end of the sequence η. In addition, $\eta_{\mathbb{E}}$ denotes the conversion of the incremental assignment η to a non-incremental assignment as defined in Fig. 5.9 (pg. 61). We let $\mathbb{E}_{\text{incDAG}}$ denote the set of all variable assignments for incremental S-DAG generation. Now S-DAG navigation roughly corresponds to navigation in XML documents via XPath axes [W3C99b, W3C03]. We discuss further details about copying in Sect. 8.4.5.

If an atomic formula or a negated formula needs to be re-evaluated, we employ the non-incremental S-DAG generator $\mathcal{D}$. Recall that only atomic formulae can appear in a negated context; thus incremental S-DAG generation performs exactly as non-incremental S-DAG generation for negated formulae. For conjunctions, we extend the current variable assignment η by a junction binding. Then we proceed as in non-incremental S-DAG generation. Notice that below disjunctions we use non-incremental S-DAG generation, because it is unsound to copy parts from the old S-DAG below disjunctions (see Sect. 6.3.2).

For universally quantified formulae, we proceed like in incremental report generation. First, we compute the quantifier sphere incrementally, resulting in the four sets *new*, *chg*, *old*, and *del*. We mark the values in $\mathit{new} \cup \mathit{chg}$ as new and the values in *old* as old. Then the subformula ϕ is evaluated for possible assignment extensions to these values ($\langle \eta : x \mapsto (v, \mathsf{new})\rangle$ and $\langle \eta : x \mapsto (v, \mathsf{old})\rangle$, respectively). If ϕ is violated for an assignment extension, we assign its S-DAG

[7] To simplify notation, we let notEval neglect junction bindings in η.

[8] Application of a variable assignment to a variable carries over from Sect. 5.5.

$$
\begin{array}{l}
\mathcal{ID}_A^{(kept,neg)}[\![p(e_1,\ldots,e_n)^{xs}\ hss]\!]\eta \\
\qquad\qquad = \begin{array}{ll}
\mathsf{copy}(\textcircled{D},\eta) & \text{if } \mathsf{notEval}(xs,\eta) \\
\mathcal{D}_A^{(kept,neg)}[\![p(e_1,\ldots,e_n)\ hss]\!]\eta_{\mathbb{E}} & \text{otherwise}
\end{array}
\end{array}
$$

$$
\mathcal{ID}_A^{(kept,neg)}[\![\neg\phi^{xs}]\!]\eta = \begin{array}{ll}
\mathsf{copy}(\textcircled{D},\eta) & \text{if } \mathsf{notEval}(xs,\eta) \\
\mathcal{D}_A^{(kept,neg)}[\![\neg\phi]\!]\eta_{\mathbb{E}} & \text{otherwise}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{ID}_A^{(kept,neg)}[\![\phi \wedge^{xs} \psi]\!]\eta = \begin{array}{ll}
\mathsf{copy}(\textcircled{D},\eta) & \text{if } \mathsf{notEval}(xs,\eta) \\
\mathsf{reduce}_{\wedge}(d_\phi,d_\psi) & \text{else if } d_\phi \neq \bigcirc \text{ and } d_\psi \neq \bigcirc \\
\textcircled{$\wedge$}\ \{(1,d_\phi)\} & \text{else if } d_\phi \neq \bigcirc \\
\textcircled{$\wedge$}\ \{(2,d_\psi)\} & \text{else if } d_\psi \neq \bigcirc \\
\bigcirc & \text{otherwise}
\end{array} \\
\quad \text{where } d_\phi = \mathcal{ID}_A^{(kept,neg)}[\![\phi]\!]\langle\eta:1\rangle; \qquad d_\psi = \mathcal{ID}_A^{(kept,neg)}[\![\psi]\!]\langle\eta:2\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{ID}_A^{(kept,neg)}[\![\phi \vee^{xs} \psi]\!]\eta = \begin{array}{ll}
\mathsf{copy}(\textcircled{D},\eta) & \text{if } \mathsf{notEval}(xs,\eta) \\
\mathsf{reduce}_{\vee}(d_\phi,d_\psi) & \text{else if } d_\phi \neq \bigcirc \text{ and } d_\psi \neq \bigcirc \\
\bigcirc & \text{otherwise}
\end{array} \\
\quad \text{where } d_\phi = \mathcal{D}_A^{(kept,neg)}[\![\phi]\!]\eta_{\mathbb{E}}; \qquad d_\psi = \mathcal{D}_A^{(kept,neg)}[\![\psi]\!]\eta_{\mathbb{E}}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{ID}_A^{(kept,neg)}[\![\forall^{xs}\ x^k \in e \bullet \phi]\!]\eta \\
\qquad\qquad = \begin{array}{ll}
\mathsf{copy}(\textcircled{D},\eta) & \text{if } \mathsf{notEval}(xs,\eta) \\
\boxed{\downarrow(e)\ \mathsf{False}\ \emptyset} & \text{else if } \mathcal{IV}_A[\![e]\!]\eta \text{ is not defined} \\
\textcircled{$\forall$}\ x\ F & \text{else if } F \neq \emptyset \\
\bigcirc & \text{otherwise}
\end{array} \\
\quad \text{where } kept' = \begin{array}{ll} \{x\} \cup kept & \text{if } k = \mathsf{KEEP} \\ kept & \text{otherwise} \end{array} \\
\qquad\quad (new, chg, old, del) = \mathcal{IV}_A[\![e]\!]\eta_{\mathbb{E}} \\
\qquad\quad ds \ = \left\{\left(v, \mathcal{ID}_A^{(kept',neg)}[\![\phi]\!]\langle\eta : x \mapsto (v,\mathsf{old})\rangle\right) \mid v \in old\right\} \cup \\
\qquad\qquad\quad\ \left\{\left(v, \mathcal{ID}_A^{(kept',neg)}[\![\phi]\!]\langle\eta : x \mapsto (v,\mathsf{new})\rangle\right) \mid v \in new \cup chg\right\} \\
\qquad\quad F \ = \{(v,\emptyset,d) \mid (v,d) \in ds \text{ and } d \neq \bigcirc\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{ID}_A^{(kept,neg)}[\![\exists^{xs}\ x^k \in e \bullet \phi]\!]\eta \\
\qquad\qquad = \begin{array}{ll}
\mathsf{copy}(\textcircled{D},\eta) & \text{if } \mathsf{notEval}(xs,\eta) \\
\bigcirc & \text{else if } \mathcal{IV}_A[\![e]\!]\eta \text{ is not defined} \\
\boxed{\mathtt{null}(e)\ \mathsf{True}\ \emptyset} & \text{else if } new \cup old = \emptyset \\
\mathsf{reduce}_{\exists}(x,F) & \text{else if } |F| = |new \cup old| \\
\bigcirc & \text{otherwise}
\end{array} \\
\quad \text{where } kept' = \begin{array}{ll} \{x\} \cup kept & \text{if } k = \mathsf{KEEP} \\ kept & \text{otherwise} \end{array} \\
\qquad\quad (new_0, chg, old_0, del) = \mathcal{IV}_A[\![e]\!]\eta_{\mathbb{E}} \\
\qquad\quad (new, old) = \begin{array}{ll} (new_0 \cup old_0 \cup chg, \emptyset) & \text{if } del \cup chg \neq \emptyset; \\ (new_0, old_0) & \text{otherwise} \end{array} \\
\qquad\quad ds \ = \left\{\left(v, \mathcal{ID}_A^{(kept',neg)}[\![\phi]\!]\langle\eta : x \mapsto (v,\mathsf{old})\rangle\right) \mid v \in old\right\} \cup \\
\qquad\qquad\quad\ \left\{\left(v, \mathcal{ID}_A^{(kept',neg)}[\![\phi]\!]\langle\eta : x \mapsto (v,\mathsf{new})\rangle\right) \mid v \in new\right\} \\
\qquad\quad F \ = \{(v,\emptyset,d) \mid (v,d) \in ds \text{ and } d \neq \bigcirc\}
\end{array}
$$

Figure 8.14: An incremental S-DAG generation algorithm ($\textcircled{D}$ denotes the old S-DAG from the previous consistency check)

d to the current sphere value v. Finally, we create a universal node, if ϕ is violated for any assignment extension. For existentially quantified formulae, we use a similar approach. Notice, however, that we must take care for changed or deleted values like in incremental report generation.

Next, we illustrate S-DAG generation by our running example.

8.4.4 Examples

We review incremental S-DAG generation at state 4. Fig. 8.15 shows how our incremental S-DAG generation algorithm evaluates rule ϕ_1. In the evaluation tree, vertices represent conjunctions, disjunctions, or quantifier spheres. Old values are printed in grey and new values in black. A path from the tree root to a leaf can be seen as a variable assignment. For convenience, we also show the result S-DAG of each copy action, where we omit the parameter Ⓓ for copy.

The sets $new = \{4\}$, $chg = \emptyset$, $old = \{1, 2, 3\}$, and $del = \emptyset$ compose the sphere of the variable t at state 4. Since t's subformula contains only t freely and is referentially transparent, we can abort re-evaluation for values in *old*. Instead, we copy the relevant parts from the old S-DAG Ⓓ. For $\langle t \mapsto (4, \mathsf{new})\rangle$, we have to re-evaluate t's subformula.

We review evaluation of the conjunction $k = \mathtt{key}(d) \;\wedge\; \exists\; m \ldots$ for the variable assignment

$$\left\langle \begin{array}{l} t \mapsto (4, \mathsf{new}) : x \mapsto (\{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}, \mathsf{old}) : k \mapsto (\mathsf{kaA2}, \mathsf{old}) : \\ d \mapsto (\{\mathtt{key} = \mathsf{kaA3}, \mathtt{kId} = \mathsf{man1.xml}, \mathtt{kKind} = \mathsf{technical\ M.}\}, \mathsf{old}) \end{array} \right\rangle .$$

The corresponding evaluation tree is shown in the bottom part of Fig. 8.15. For the left hand side of the conjunction, we copy part of the old S-DAG, because all variables in the formula $k = \mathtt{key}(d)$ are marked as old in the current assignment. Notice that we adapt the binding $t \mapsto (4, \mathsf{new})$ to the previous repository state: $t \mapsto (3, \mathsf{new})$, similar to incremental report generation. In contrast to incremental report generation, we also need the bindings of existentially quantified variables, because they are stored in S-DAGs, too. The right hand side of the conjunction must be re-evaluated, because it contains the new variable t freely. The sphere of the existential quantifier for m is composed of the following sets:

$$\begin{array}{lcl} new_0 & = & \left\{ \begin{array}{l} \{\mathtt{dId} = \mathsf{man2.xml}, \mathtt{dState} = 4\}, \{\mathtt{dId} = \mathsf{man3.xml}, \mathtt{dState} = 4\}, \\ \{\mathtt{dId} = \mathsf{man4.xml}, \mathtt{dState} = 4\} \end{array} \right\} \\ chg & = & \emptyset \\ old_0 & = & \{\{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 1\}\} \\ del & = & \emptyset \end{array}$$

Since no values are deleted or changed, we copy part of the old S-DAG when m is bound to man1.xml. For the values in new_0, we have to re-evaluate the subformula. In order to generate the existential node for m, we consider the four S-DAGs shown in Fig. 8.16. Clearly, the copied S-DAG is smaller than the other S-DAGs. Therefore, we retain the copied S-DAG only and replace the other S-DAGs by Abandoned. In addition, we replace the bindings of m to

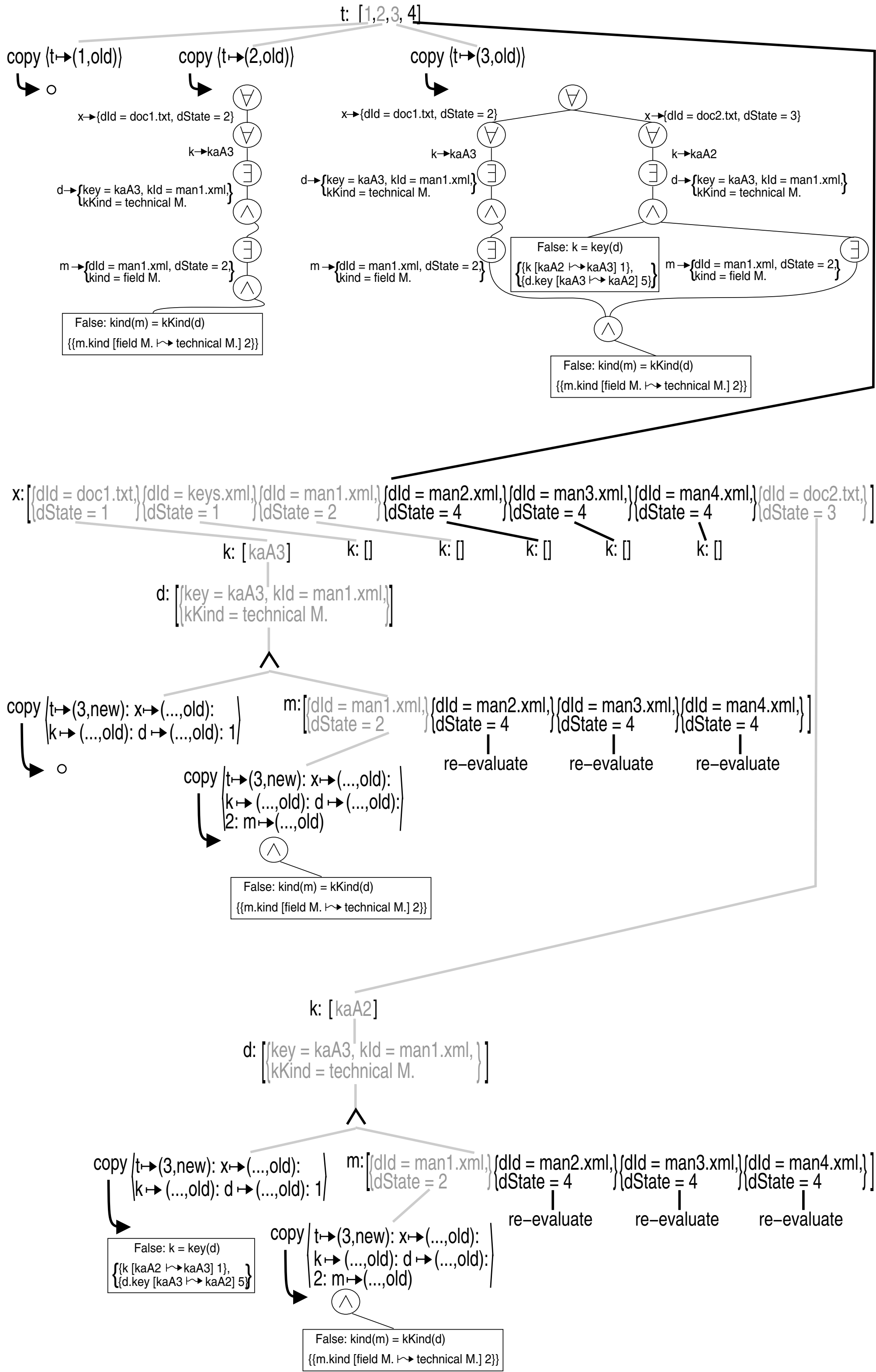


Figure 8.15: Incremental S-DAG generation for rule ϕ_1 at state 4

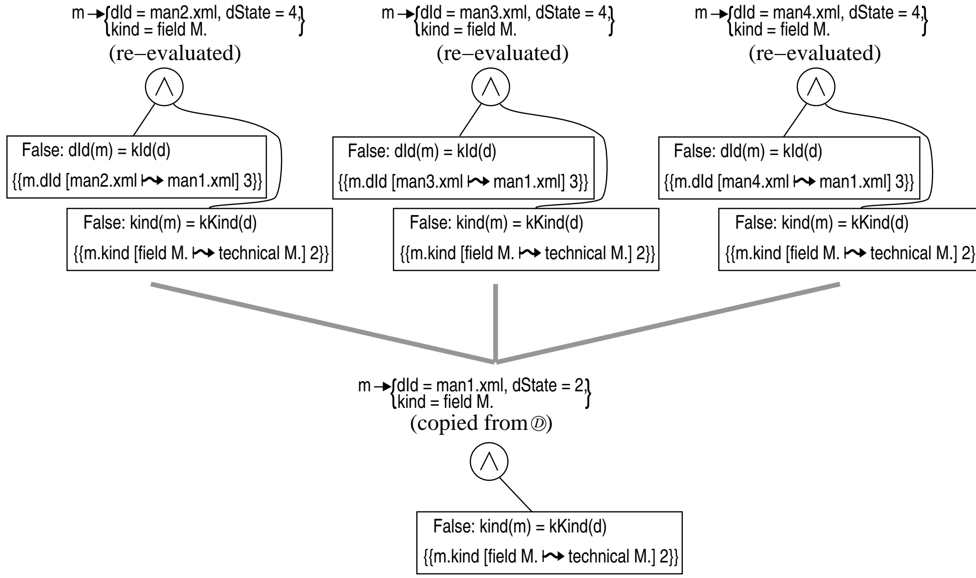


Figure 8.16: Hasse diagram w.r.t. $\prec_c$ of S-DAGs generated for the subformula $\texttt{dId}(m) = \texttt{kId}(d) \ \wedge\ \texttt{kind}(m) = \texttt{kKind}(d)$ in rule ϕ_1 at state 4

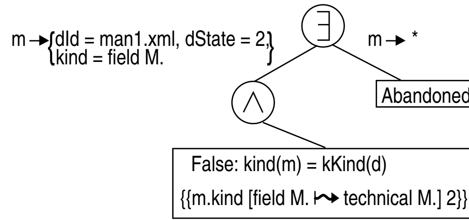


Figure 8.17: Resulting S-DAG for the existential quantifier for m in rule ϕ_1 at state 4

the corresponding manuals by a dummy binding $m \mapsto *$. Fig. 8.17 shows the resulting S-DAG for the existential quantifier for m.

In the next section, we define S-DAG ordering relations and auxiliary functions. In particular, we discuss copying parts of the old S-DAG.

8.4.5 Auxiliary Functions

In this section, we show how hints are evaluated. Also, we introduce two S-DAG ordering relations $\prec$ and $\prec_c$. Finally, we define auxiliary functions needed for (incremental) S-DAG generation.

Fig. 8.18 shows the denotational semantics of the function $\mathcal{S}$, which evaluates the hint collection *hss* for an atomic formula ϕ with boolean result b. Applied to an empty hint collection, $\mathcal{S}$ generates a generic predicate suggestion for each variable free in ϕ.[9] Otherwise, we generate a suggestion for each hint in the

[9] We use $\neg$ as a meta symbol here, meaning to invert a boolean value True to False and vice versa.

Evaluate a hint collection

$$\begin{array}{lcl}
\mathcal{S}[\![.]\!] & : & \wp(\mathcal{X}) \times \mathbb{A} \times (\wp(\wp(\mathcal{H})) \times \mathcal{F}_{at} \times \mathbb{B}) \times \mathbb{E} \rightarrow \wp(\wp(\mathbb{S})) \\
\mathcal{S}_A^{kept}[\![(\emptyset, \phi, b)]\!]\eta & = & \{\ \{x\ [\neg b :\ \phi] \mid x \in \mathsf{fv}(\phi) \text{ and } x \notin \mathit{kept}\}\ \} \\
\mathcal{S}_A^{kept}[\![(hss, _, _)]\!]\eta & = & \{\{h^A \mid h \in hs \text{ and } h^A \text{ is defined}\} \mid hs \in hss\} \\
& & \quad \text{where } h^A = \mathcal{S'}_A^{kept}[\![h]\!]\eta
\end{array}$$

Evaluate a hint

$$\begin{array}{lcl}
\mathcal{S}'[\![.]\!] & : & \wp(\mathcal{X}) \times \mathbb{A} \times \mathcal{H} \times \mathbb{E} \nrightarrow \mathbb{S} \\
\mathcal{S'}_A^{kept}[\![x \rightsquigarrow e\ b\ k\ c]\!]\eta & = & x\ [\mathcal{V}_A[\![x]\!]\eta \rightsquigarrow \mathcal{V}_A[\![e]\!]\eta]\ c \quad \text{if } x \notin \mathit{kept} \\
& & x\ [???] \quad \text{otherwise} \\
\mathcal{S'}_A^{kept}[\![x.l \rightsquigarrow e\ b\ k\ c]\!]\eta & = & x.l\ [\mathcal{V}_A[\![l(x)]\!]\eta \rightsquigarrow \mathcal{V}_A[\![e]\!]\eta]\ c \quad \text{if } x \notin \mathit{kept} \\
& & x\ [???] \quad \text{otherwise}
\end{array}$$

Figure 8.18: Evaluating hint collections

collection hss via $\mathcal{S}'$. If we encounter a hint for a "kept" variable, we generate a suggestion expressing this contradiction. The function $\mathcal{S}'$ is partial, because value calculation is partial. For evaluation of a hint, we require that the hint term is defined. In contrast, $\mathcal{S}$ is total — hints that cannot be evaluated are dropped.

Next, we define two strict ordering relations for S-DAGs. By $\prec$ we consider an S-DAG d_1 smaller than an S-DAG d_2 iff all leaves in d_1 are also contained by d_2 and d_1 represents strictly less inconsistencies than d_2. The relation $\prec_c$ also respects the costs of S-DAGs (provided their costs are defined): We consider an S-DAG d_1 smaller than an S-DAG d_2, if the cost of d_1 is smaller than the cost of d_2. We employ $\prec$ for reducing conjunction nodes, whereas $\prec_c$ is used for reducing disjunction nodes and existential nodes. Clearly, for a conjunction node we must neglect the costs of its sub-DAGs, in order to retain necessary repair actions. In contrast, for disjunction nodes and existential nodes it is sufficient to repair the cheapest S-DAG only and to neglect expensive S-DAGs, even though these S-DAGs may contain other predicate suggestions. We define $\prec$ and $\prec_c$ as follows:

$$\begin{array}{lcl}
d_1 \prec d_2 & :\Leftrightarrow & d_1 \preceq d_2 \ \wedge\ \mathsf{incons}(d_1) < \mathsf{incons}(d_2) \\
d_1 \prec_c d_2 & :\Leftrightarrow & d_1 \preceq_c d_2 \ \wedge\ \mathsf{incons}(d_1) < \mathsf{incons}(d_2)
\end{array}$$

The function $\mathsf{incons} : \mathbb{G} \rightarrow \mathbb{N}$ determines the number of inconsistencies represented by an S-DAG (for brevity, we omit a formal definition). $\preceq$ and $\preceq_c$ are S-DAG ordering relations corresponding to $\prec$ and $\prec_c$, respectively. Fig. 8.19 shows their formal definitions. We abbreviate universal and conjunction nodes by $\textcircled{\scriptsize$\forall\!\wedge$}\ ds$, and existential and disjunction nodes by $\textcircled{\scriptsize$\exists\!\vee$}\ ds$. The sub-DAGs of a node are called ds or ds'; for brevity, we neglect edge labels and deal with the sub-DAGs of a node only. For example, $\textcircled{\scriptsize$\forall\!\wedge$}\ ds \preceq \textcircled{\scriptsize$\forall\!\wedge$}\ ds'$ abbreviates four cases:

$$\textcircled{\scriptsize$\forall$}\ ds \preceq \textcircled{\scriptsize$\forall$}\ ds'; \quad \textcircled{\scriptsize$\forall$}\ ds \preceq \textcircled{\scriptsize$\wedge$}\ ds'; \quad \textcircled{\scriptsize$\wedge$}\ ds \preceq \textcircled{\scriptsize$\forall$}\ ds'; \text{ and } \quad \textcircled{\scriptsize$\wedge$}\ ds \preceq \textcircled{\scriptsize$\wedge$}\ ds'.$$

The definition of $\preceq$ is straightforward. For example, a universal node u_1 is smaller than a universal node u_2 iff each sub-DAG of u_1 is smaller than a sub-DAG of u_2. A universal node u is smaller than an existential node e iff each

Partial order for S-DAGs that neglects costs ($\preceq$)

	$\preceq$		$\subseteq$	$\mathbb{G} \times \mathbb{G}$
d	$\preceq$	$\boxed{\textsf{Abandoned}}$		
$\owedge\ ds$	$\preceq$	$\owedge\ ds'$	$:\Leftrightarrow$	$\forall\, d \in ds \bullet \exists\, d' \in ds' \bullet d \preceq d'$
$\owedge\ ds$	$\preceq$	$\ovee\ ds'$	$:\Leftrightarrow$	$\forall\, d \in ds \bullet \forall\, d' \in ds' \bullet d \preceq d'$
$\owedge\ ds$	$\preceq$	$\boxed{p}$	$:\Leftrightarrow$	$\forall\, d \in ds \bullet d \preceq \boxed{p}$
$\ovee\ ds$	$\preceq$	$\owedge\ ds'$	$:\Leftrightarrow$	$\exists\, d \in ds \bullet \exists\, d' \in ds' \bullet d \preceq d'$
$\ovee\ ds$	$\preceq$	$\ovee\ ds'$	$:\Leftrightarrow$	$\forall\, d \in ds \bullet \exists\, d' \in ds' \bullet d \preceq d'$
$\ovee\ ds$	$\preceq$	$\boxed{p}$	$:\Leftrightarrow$	$\exists\, d \in ds \bullet d \preceq \boxed{p}$
$\boxed{p}$	$\preceq$	$\owedge\ ds$	$:\Leftrightarrow$	$\exists\, d \in ds \bullet \boxed{p} \preceq d$
$\boxed{p}$	$\preceq$	$\ovee\ ds$	$:\Leftrightarrow$	$\forall\, d \in ds \bullet \boxed{p} \preceq d$
$\boxed{p\ b\ sss}$	$\preceq$	$\boxed{p'\ b'\ sss'}$	$:\Leftrightarrow$	$p = p' \;\wedge\; b = b'$

Partial order for S-DAGs that respects costs ($\preceq_c$)

	$\preceq_c$		$\subseteq$	$\mathbb{G} \times \mathbb{G}$	
d	$\preceq_c$	$\boxed{\textsf{Abandoned}}$			
$\owedge\ ds$	$\preceq_c$	$\owedge\ ds'$	$:\Leftrightarrow$	$\mathsf{cost}\,(\owedge\ ds) \sqsubset \mathsf{cost}\,(\owedge\ ds')$	$\blacktriangleright$
				$\forall\, d \in ds \bullet \exists\, d' \in ds' \bullet d \preceq_c d'$	
$\owedge\ ds$	$\preceq_c$	$\ovee\ ds'$	$:\Leftrightarrow$	$\mathsf{cost}\,(\owedge\ ds) \sqsubset \mathsf{cost}\,(\ovee\ ds')$	$\blacktriangleright$
				$\forall\, d \in ds \bullet \forall\, d' \in ds' \bullet d \preceq_c d'$	
$\owedge\ ds$	$\preceq_c$	$\boxed{p}$	$:\Leftrightarrow$	$\mathsf{cost}\,(\owedge\ ds) \sqsubset \mathsf{cost}\left(\boxed{p}\right)$	$\blacktriangleright$
				$\forall\, d \in ds \bullet d \preceq_c \boxed{p}$	
$\ovee\ ds$	$\preceq_c$	$\owedge\ ds'$	$:\Leftrightarrow$	$\mathsf{cost}\,(\ovee\ ds) \sqsubset \mathsf{cost}\,(\owedge\ ds')$	$\blacktriangleright$
				$\exists\, d \in ds \bullet \exists\, d' \in ds' \bullet d \preceq_c d'$	
$\ovee\ ds$	$\preceq_c$	$\ovee\ ds'$	$:\Leftrightarrow$	$\mathsf{cost}\,(\ovee\ ds) \sqsubset \mathsf{cost}\,(\ovee\ ds')$	$\blacktriangleright$
				$\forall\, d \in ds \bullet \exists\, d' \in ds' \bullet d \preceq_c d'$	
$\ovee\ ds$	$\preceq_c$	$\boxed{p}$	$:\Leftrightarrow$	$\mathsf{cost}\,(\ovee\ ds) \sqsubset \mathsf{cost}\left(\boxed{p}\right)$	$\blacktriangleright$
				$\exists\, d \in ds \bullet d \preceq_c \boxed{p}$	
$\boxed{p}$	$\preceq_c$	$\owedge\ ds$	$:\Leftrightarrow$	$\mathsf{cost}\left(\boxed{p}\right) \sqsubset \mathsf{cost}\,(\owedge\ ds)$	$\blacktriangleright$
				$\exists\, d \in ds \bullet \boxed{p} \preceq_c d$	
$\boxed{p}$	$\preceq_c$	$\ovee\ ds$	$:\Leftrightarrow$	$\mathsf{cost}\left(\boxed{p}\right) \sqsubset \mathsf{cost}\,(\ovee\ ds)$	$\blacktriangleright$
				$\forall\, d \in ds \bullet \boxed{p} \preceq_c d$	
$\boxed{p\ b\ sss}$	$\preceq_c$	$\boxed{p'\ b'\ sss'}$	$:\Leftrightarrow$	$\mathsf{cost}\left(\boxed{p\ b\ sss}\right) \sqsubset \mathsf{cost}\left(\boxed{p'\ b'\ sss'}\right)$	$\blacktriangleright$
				$p = p' \;\wedge\; b = b'$	

Figure 8.19: Partial orders for S-DAGs: $\preceq$, $\preceq_c$ ($\boxed{p}$ abbreviates a predicate leaf $\boxed{p\ b\ sss}$)

Costs of an S-DAG

$$\begin{array}{lcl}
\mathsf{cost} & : & \mathbb{G} \nrightarrow \mathbb{N} \\
\mathsf{cost}\,(\bigwedge ds) & = & \sum_{d \in ds} \mathsf{cost}(d) \\
\mathsf{cost}\,(\bigvee ds) & = & \min_{d \in ds} \mathsf{cost}(d) \\
\mathsf{cost}\left(\boxed{p\ b\ sss}\right) & = & \min_{ss \in sss} \left(\sum_{s \in ss} \mathsf{sugcost}(s) \right)
\end{array}$$

Costs of a predicate suggestion

$$\begin{array}{lcl}
\mathsf{sugcost} & : & \mathbb{S} \nrightarrow \mathbb{N} \\
\mathsf{sugcost}\,(x\ [v \rightsquigarrow v']\ c) & = & c \\
\mathsf{sugcost}\,(x.l\ [v \rightsquigarrow v']\ c) & = & c
\end{array}$$

Figure 8.20: Calculating costs for S-DAGs

sub-DAG of u is smaller than all sub-DAGs of e. For predicate leafs, we require that they describe the same inconsistency.

The formal definition for $\preceq_c$ follows the pattern of $\preceq$ but also takes S-DAG costs into account, if they are defined. If the costs of two S-DAGs are equal or one cost is undefined, we compare their sub-DAGs. Formally, the costs c_1 and c_2 are in the relation $\sqsubset$ iff both costs are defined and $c_1 < c_2$; they are not in the relation $\sqsubset$ iff $c_1 > c_2$; $\sqsubset$ is undefined, if $c_1 = c_2$, or one cost is undefined. We let $e_1 \blacktriangleright e_2$ denote "if the expression e_1 is defined then e_1 else e_2."

This is necessary, because the function cost is partial. It determines the cost of an S-DAG (see Fig. 8.20). We employ a simple minimax algorithm: To an alternative node we assign the cost of its cheapest sub-DAG. Otherwise, we assign the sum of the costs of all sub-DAGs. The cost of a predicate leaf is determined by its hints. The cost function is partial, because the rule designer may omit costs in hints. It is understood that for a node d the costs for all sub-DAGs must be defined; otherwise, $\mathsf{cost}(d)$ is not defined.

Fig. 8.21 shows the definitions of auxiliary functions used for S-DAG generation. filterHints determines the hints relevant for an atomic formula ϕ. A hint is relevant, if its boolean value corresponds to ϕ's boolean result and the variable kind of the hint equals the variable kind of the hint's variable in the current assignment η. If no variable kind is given for a hint, we consider the boolean value only. The function $\mathsf{reduce}_\wedge$ reduces the sub-DAGs below a conjunction node and retains the greatest S-DAGs; finally it generates a conjunction node. The smaller S-DAG is replaced by $\boxed{\mathsf{Abandoned}}$. In contrast, $\mathsf{reduce}_\vee$ reduces the sub-DAGs below a disjunction node and retains the smallest S-DAGs; finally it generates a disjunction node. The greater S-DAG is replaced by $\boxed{\mathsf{Abandoned}}$. We reduce the sub-DAGs below existential nodes by $\mathsf{reduce}_\exists$. Notice that $\mathsf{reduce}_\wedge$ neglects the costs of S-DAGs, whereas $\mathsf{reduce}_\vee$ and $\mathsf{reduce}_\exists$ respect the costs of S-DAGs.

Copying parts of the old S-DAG is an integral part of incremental S-DAG generation. The function $\mathsf{copy}(d, \eta)$ determines the relevant part of an S-DAG d w.r.t. the current variable assignment η. During S-DAG generation, we attach variable bindings and junctions bindings to the *end* of the current assignment.

Filter hints

$$
\begin{array}{lcl}
\mathsf{filterHints} & : & \wp(\wp(\mathcal{H})) \times \mathbb{B} \times \mathbb{E} \to \wp(\wp(\mathcal{H})) \\
\mathsf{filterHints}(hss, b, \eta) & = & \{\{h \mid \mathsf{relevant}(h, b, \eta) \wedge h \in hs\} \mid hs \in hss\}
\end{array}
$$

Is a hint relevant for evaluation?

$$
\begin{array}{lcl}
\mathsf{relevant} & : & \mathcal{H} \times \mathbb{B} \times \mathbb{E} \to \mathbb{B} \\
\mathsf{relevant}(x \ \rightsquigarrow \ e\ b\ k\ c, b', \eta) & = & b = b' \ \wedge \ x \mapsto (_, k) \in \eta \\
\mathsf{relevant}(x.l \ \rightsquigarrow \ e\ b\ k\ c, b', \eta) & = & b = b' \ \wedge \ x \mapsto (_, k) \in \eta
\end{array}
$$

Reduce sub-DAGs for conjunctions

$$
\begin{array}{lcll}
\mathsf{reduce}_\wedge & : & \mathbb{G} \times \mathbb{G} \to \mathbb{G} & \\
\mathsf{reduce}_\wedge(d_1, d_2) & = & \owedge \ \{(1, \boxed{\mathsf{Abandoned}}), (2, d_2)\} & \text{if } d_1 \prec d_2 \\
 & & \owedge \ \{(1, d_1), (2, \boxed{\mathsf{Abandoned}})\} & \text{else if } d_2 \prec d_1 \\
 & & \owedge \ \{(1, d_1), (2, d_2)\} & \text{otherwise}
\end{array}
$$

Reduce sub-DAGs for disjunctions

$$
\begin{array}{lcll}
\mathsf{reduce}_\vee & : & \mathbb{G} \times \mathbb{G} \to \mathbb{G} & \\
\mathsf{reduce}_\vee(d_1, d_2) & = & \ovee \ \{(1, d_1), (2, \boxed{\mathsf{Abandoned}})\} & \text{if } d_1 \prec_c d_2 \\
 & & \ovee \ \{(1, \boxed{\mathsf{Abandoned}}), (2, d_2)\} & \text{else if } d_2 \prec_c d_1 \\
 & & \ovee \ \{(1, d_1), (2, d_2)\} & \text{otherwise}
\end{array}
$$

Reduce sub-DAGs for existentially quantified formulae

$$
\begin{array}{lcl}
\mathsf{reduce}_\exists & : & \mathcal{X} \times \wp(\mathbb{V} \times \wp(\mathbb{C}) \times \mathbb{G}) \to \mathbb{G} \\
\mathsf{reduce}_\exists(x, es) & = & \textcircled{$\exists$}\ x \ \{\mathsf{replace}(e, es \setminus \{e\}) \mid e \in es\}
\end{array}
$$

Replace sub-DAGs that are too expensive by $\boxed{\mathsf{Abandoned}}$

$$
\begin{array}{lcll}
\mathsf{replace} & : & (\mathbb{V} \times \wp(\mathbb{C}) \times \mathbb{G}) \times \wp(\mathbb{V} \times \wp(\mathbb{C}) \times \mathbb{G}) & \\
 & & \to \wp(\mathbb{V} \times \wp(\mathbb{C}) \times \mathbb{G}) & \\
\mathsf{replace}((v, acts, d), es) & = & \big(*, \emptyset, \boxed{\mathsf{Abandoned}}\big) & \text{if } \exists \ (_, _, d') \in es \ \bullet \\
 & & & \quad d' \prec_c d \\
 & & (v, acts, d) & \text{otherwise}
\end{array}
$$

Copy the relevant part of the old S-DAG

$$
\begin{array}{lcl}
\mathsf{copy} & : & \mathbb{G} \times \mathbb{E}_{\text{incDAG}} \to \mathbb{G} \\
\mathsf{copy}(d, \langle\rangle) & = & d \\
\mathsf{copy}(\boxed{\mathsf{Abandoned}}, _) & = & \boxed{\mathsf{Abandoned}} \\
\mathsf{copy}(\odot \ es, \langle n \ : \ \eta\rangle) & = & d' \quad \text{if } ds' = \{d'\} \\
 & & \bigcirc \quad \text{otherwise}
\end{array}
$$

$$
\text{where } ds' \ = \ \{\mathsf{copy}(d, \eta) \mid n = n' \ \wedge \ (n', d) \in es\}
$$

$$
\begin{array}{lcll}
\mathsf{copy}(\textcircled{Q}\ x \ es, \langle x \mapsto (v, k) \ : \eta\rangle) & = & d' & \text{if } ds' = \{d'\} \\
 & & \bigcirc & \text{else if } ds' = \emptyset \text{ and } Q = \forall \\
 & & \boxed{\mathsf{Abandoned}} & \text{else if } ds' = \emptyset \text{ and } Q = \exists
\end{array}
$$

$$
\begin{array}{lcll}
\text{where } (v', k') & = & (\mathsf{previous\ check\ state}, \mathsf{old}) & \text{if } v = \mathsf{current\ state} \\
 & & (v, k) & \text{otherwise} \\
ds' & = & \{\mathsf{copy}(d, \eta) \mid (v' = v_e \vee k' = \mathsf{new}) \ \wedge \ (v_e, _, d) \in es\} &
\end{array}
$$

Figure 8.21: Auxiliary functions for S-DAG generation

Therefore, the order of an assignment matches the order of nodes in a path of an S-DAG. Thus, basically, copy corresponds to recursive search within a labeled tree or DAG. An important question to consider is whether the result of copy is well defined. For conjunction and disjunction nodes, respectively, at most one edge can match the junction binding of η. Consequently, the S-DAG set ds' contains at most one S-DAG. For quantifier nodes, all edges match, if the variable kind k' is new. All S-DAGs returned by $\mathsf{copy}(d, \eta)$ are, however, equal. That is, because we copy part from the old S-DAG for a formula ϕ only if all free variables in ϕ are old and ϕ contains referentially transparent symbols only. Thus, the S-DAG set ds' contains at most one S-DAG.

8.5 Interactive Repair

At state 4, for rule ϕ_1, the algorithms from the previous section generate the S-DAG shown in Fig. 8.10 (pg. 108). This S-DAG lacks, however, sufficient information about how inconsistencies can be best repaired, e.g., by adding, changing, or deleting content. In order to support interactive repair, we *augment* S-DAGs by repair actions (see Sect. 8.5.1). This can be done at any time and is fully independent of consistency checking itself. In consequence, the repository is not locked during S-DAG augmentation. From an augmented S-DAG, authors can choose repair actions in a trial and error process, which we illustrate in Sect. 8.5.2. In Sect. 8.5.3, we discuss the impact of repair actions to the document structure. Finally, we define our S-DAG augmentation algorithm in Sect. 8.5.4.

8.5.1 Augmenting S-DAGs

Our basic idea is to annotate quantifier edges by *repair actions*. An action proposes to either add a value to (Add), or change a value within (Chg), or delete a value from the sphere of a quantifier (Del). Thus, actions propose to add, change, or delete documents or document content. We augment the S-DAG from Fig. 8.10 (pg. 108) as shown in Fig. 8.22; actions are marked grey. For example, we propose to change the key kaA2 to kaA3 or to delete the document doc1.txt. The edge below the universal node for the temporal variable t is annotated by KEEP, which means that we must not change t. Actions for a quantifier edge are obtained from the predicate suggestions in its target S-DAG. Clearly, it makes sense to augment the current part of an S-DAG only. Due to the nature of DMSs, we cannot change older document versions. Previous inconsistencies document, however, the development of the repository, which has proven a quite useful feature. The current part of an S-DAG contains all paths from the S-DAG root to a leaf that contain a binding to the current repository state or do not contain a binding to a repository state at all.

Inconsistencies resulting from *missing document content* can be resolved either by adding new content or by changing content that we regard as defective; deleting content cannot resolve this kind of inconsistency. The challenge for an

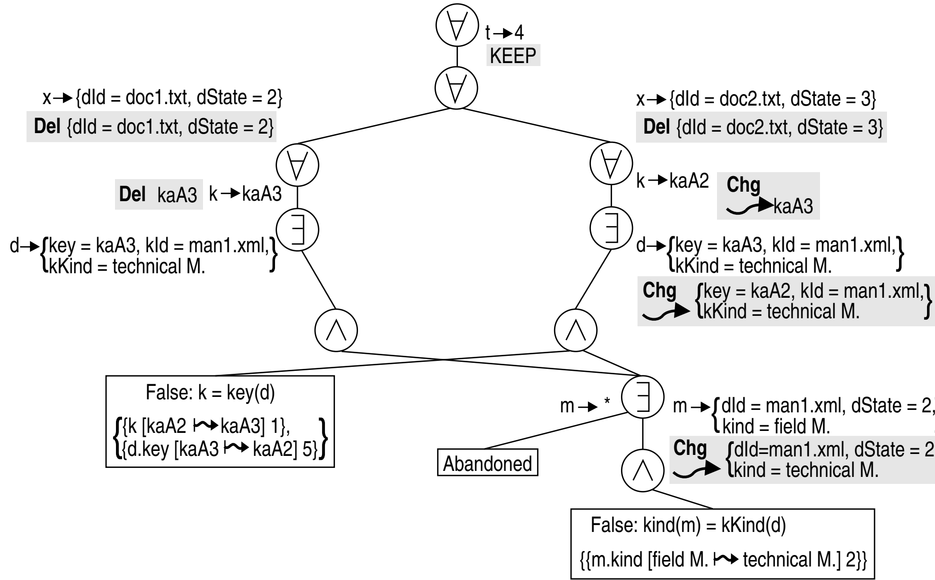


Figure 8.22: Augmented S-DAG for rule ϕ_1 at state 4

existential node (representing missing content) is to find a practical criterion to decide whether a value should be changed or added. Our criterion aims at *minimal change*. We propose to change a value, if we find a minimal defective value in the quantifier sphere, i.e., one of the S-DAGs below the existential node is minimal w.r.t. the S-DAG ordering $\prec_c$. In other words, exactly one edge does not target Abandoned. In this case, it is most likely that missing content is actually defective content. Otherwise, i.e., if there are some "equally good" alternatives, we propose to insert a new value to the quantifier sphere. Usually, this value is only partially defined by the predicate suggestions below the existential node. We handle missing information by "value skeletons" in which some record components can be undefined. This is similar to handling null values in databases [GL97].

Inconsistencies resulting from *dubious document content* can be resolved either by changing this content or by deleting it; adding new content cannot resolve this kind of inconsistency. For universal nodes (representing dubious content), it is easier to derive actions, because we know exactly which values cause inconsistencies. We have, however, to decide whether an offending value should be changed or deleted. Again, we employ the criterion of minimal change. Assume that we augment a universal node for a variable x and follow an edge binding a value v, i.e., v is considered dubious. If the edge's target S-DAG contains a predicate suggestion that proposes to change x, then we derive an action to change v. Otherwise, we derive an action to delete v. Clearly, in this case we do not know how the inconsistency can be resolved; thus, we propose a "fall-back" solution.

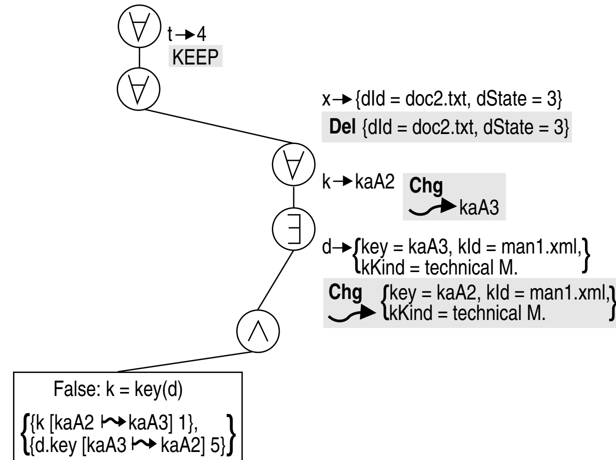


Figure 8.23: Augmented S-DAG resulting from changing the kind of the manual man1.xml to technical M.

8.5.2 Choosing Repair Actions Interactively

S-DAG augmentation derives repair actions for quantifier edges. It remains, however, unclear which actions must be applied together, in order to resolve all inconsistencies w.r.t. the corresponding rule. In our interactive approach, the author walks through an augmented S-DAG and chooses actions individually. Our system can apply a chosen repair to the S-DAG and re-augment it. If this results in an empty S-DAG, all inconsistencies are resolved. Otherwise, the remaining inconsistencies can be resolved by choosing another action. Obviously, an action attached to a quantifier edge can resolve inconsistencies in a leaf below this edge, which may cause the leaf to disappear. If each edge below a universal node or a conjunction node leads to an empty S-DAG, then this node can be deleted. An existential node or a disjunction node can be deleted, if one of its edges targets an empty S-DAG. This procedure cascades up the whole S-DAG.

For our example S-DAG in Fig. 8.22, it makes sense to repair the existential node for m, because it has two incoming paths and, therefore, is responsible for two inconsistencies. We choose to change the kind of the manual man1.xml from field M. to technical M. This causes the right hand leaf to disappear. The conjunction above this leaf is not inconsistent anymore; so is the existential node for m. Both nodes are deleted. This procedure cascades up the S-DAG structure, such that the complete S-DAG on the left hand side of the universal node for x collapses. We arrive at the S-DAG shown in Fig. 8.23, which we can repair, e.g., by changing the key kaA2 to kaA3 in the document doc1.txt.

Choosing actions from augmented S-DAGs is an interactive trial and error process, by which authors can determine reasonable actions that resolve inconsistencies. An action may, however, cause new inconsistencies. We can recognize this only after applying the action to the documents in the repository and checking the repository for consistency again. For example, changing the key component of the key definition for kaA3 (as proposed by the action for d

in Fig. 8.23) introduces new inconsistencies for the reference to the key kaA3. We discuss this issue in detail in Chapter 9.

8.5.3 Repair Actions and Document Structure

An important question is the impact of a repair action to the *document structure*. Since hints are type-checked, most actions respect the document structure, provided the document type resembles this structure correctly. Static type-checking cannot detect whether adding or deleting content could break the document structure (this would require dynamic type-checking). Consequently, actions are type safe by construction, if they propose to change content in a document. For actions proposing to add content, static type-checking only can guarantee that the content to be added is type safe, provided the document structure permits to add content at all. Actions proposing to delete content are not affected by type-checking, because we generate these fall-back solutions only if an S-DAG lacks sufficient predicate suggestions.

8.5.4 An S-DAG Augmentation Algorithm

In this section, we define our S-DAG augmentation algorithm. For each quantifier edge, we derive a set of alternative repair actions; the formal definition of actions is shown in Fig. 8.11 (pg. 109).

Formally, we let the function $\mathcal{A}(d)$ augment the quantifier edges of an S-DAG d (see Fig. 8.24); for auxiliary functions see Fig. 8.25. $\mathcal{A}$ is defined by structural induction on the structure of S-DAGs. The only interesting cases are for quantifier nodes; other nodes are not affected by $\mathcal{A}$. For nodes annotated by KEEP, we do not derive actions; for nodes annotated by CHG we derive actions to change values only.

For an existential node, we first augment its sub-DAGs resulting in the edge set es'. In this set, minDAG tries to find an edge the target S-DAG of which requires a minimal repair, i.e., (1) it is not shared, (2) it is not equal to Abandoned, and (3) all other edges lead to Abandoned. Such an edge is augmented by actions that change its value. Otherwise, minDAG is not defined; then all edges are augmented by actions proposing to add values.

Assume that minDAG has found a minimal edge (v, d), where v is the value that caused the inconsistency and, therefore, should be changed. By filterSug(x, d) we obtain all predicate suggestions in the sub-DAG d that propose to change the quantified variable x. The predicate suggestions $sugss_d$ form a collection, in which each set is an alternative. We apply each suggestion set $sugs \in sugss_d$ to the bound value v. The resulting values vs_d are those values the original value v can be changed to, in order to resolve an inconsistency. We apply predicate suggestions by applySugs$(v, sugs)$. Since predicate suggestions can be contradictive, applySugs first determines whether the suggestions do not conflict, i.e., there exists a most general unifier for the values proposed by the

$$
\begin{array}{lcl}
\mathcal{A} & : & \mathbb{G} \rightarrow \mathbb{G} \\
\mathcal{A}\left(\boxed{\mathsf{Abandoned}}\right) & = & \boxed{\mathsf{Abandoned}} \\
\mathcal{A}\left(\boxed{p}\right) & = & \boxed{p} \\
\mathcal{A}\,(\textcircled{$\wedge$}\ es) & = & \textcircled{$\wedge$}\ \{(n, \mathcal{A}\,(d)) \mid (n,d) \in es\} \\
\mathcal{A}\,(\textcircled{$\vee$}\ es) & = & \textcircled{$\vee$}\ \{(n, \mathcal{A}\,(d)) \mid (n,d) \in es\} \\
\mathcal{A}\,(\textcircled{$\exists$}\ x\ es\ \mathsf{KEEP}) & = & \textcircled{$\exists$}\ x\ \{(v, \{\mathsf{KEEP}\}, \mathcal{A}\,(d)) \mid (v, _, d) \in es\} \\
\mathcal{A}\,(\textcircled{$\exists$}\ x\ es\ annot) & = & \textcircled{$\exists$}\ x\ (\{(v, acts_d, d)\} \cup es'') \quad \text{if } \mathsf{minDAG}(es') \text{ is defined} \\
 & & \textcircled{$\exists$}\ x\ es_{\mathsf{Add}} \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\text{where } es' & = & \{(v, \emptyset, \mathcal{A}\,(d)) \mid (v, _, d) \in es\} \\
((v,d), es'') & = & \mathsf{minDAG}(es') \\
sugss_d & = & \mathsf{filterSug}(x, d) \\
vs_d & = & \{\mathsf{applySugs}(v, sugs) \mid sugs \in sugss_d\} \\
acts_d & = & \{\mathsf{Chg}\ v \rightsquigarrow v' \mid v' \in vs_d\} \\
v_0 & = & \mathsf{mkSkeleton}(\tau_x) \\
es_{\mathsf{Add}} & = & \{(v, acts, d) \mid (v, _, d) \in es'\} \\
 & & \text{where } acts = \emptyset \quad \text{if } annot = \mathsf{CHG} \\
 & & \qquad \{\mathsf{Add}\ \mathsf{applySugs}(v_0, sugs) \mid sugs \in \mathsf{filterSug}(x,d)\} \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{A}\,(\textcircled{$\forall$}\ x\ es\ \mathsf{KEEP}) & = & \textcircled{$\forall$}\ x\ \{(v, \{\mathsf{KEEP}\}, \mathcal{A}\,(d)) \mid (v, _, d) \in es\} \\
\mathcal{A}\,(\textcircled{$\forall$}\ x\ es\ annot) & = & \textcircled{$\forall$}\ x\ es''
\end{array}
$$

$$
\begin{array}{lcl}
\text{where } es' & = & \{(v, \emptyset, \mathcal{A}\,(d)) \mid (v, _, d) \in es\} \\
es'' & = & \{(v, acts, d) \mid (v, _, d) \in es'\} \\
 & & \text{where } sugss = \mathsf{filterSug}(x, d) \\
 & & \quad acts = \emptyset \quad \text{if } sugss = \emptyset \text{ and } annot = \mathsf{CHG} \\
 & & \qquad \{\mathsf{Del}\ v\} \quad \text{else if } sugss = \emptyset \\
 & & \qquad \{\mathsf{Chg}\ v \rightsquigarrow \mathsf{applySugs}(v, sugs) \mid sugs \in sugss\} \quad \text{otherwise}
\end{array}
$$

Figure 8.24: An S-DAG augmentation algorithm (for auxiliary functions see Fig. 8.25)

predicate suggestions. In that case, it is safe to apply the predicate suggestions. Otherwise, applySugs is not defined.[10]

Assume that minDAG cannot find a minimal edge. Then, for each edge, we determine the alternatives of values to insert. We apply the predicate suggestions calculated by filterSug to an empty value skeleton v_0. One of the resulting values should be inserted, in order to resolve the inconsistency. The skeleton v_0 is built by mkSkeleton applied to the type τ_x of the quantifier variable x. We denote an undefined value by $\perp$; the lifted set $\mathbb{V}^{\perp}$ contains all values from $\mathbb{V}$ and the undefined value $\perp$. We use value skeletons, in order to handle incomplete information, which is similar to handling null values in databases [GL97]. For record values, it is likely that the predicate suggestions below the edge determine the value to insert partially only.

[10] The result set of a set comprehension does not contain undefined values, they are removed automatically.

Find the minimal edge below an existential quantifier

$$
\begin{array}{lcl}
\mathsf{minDAG} & : & \wp(\mathbb{V} \times \wp(\mathbb{C}) \times \mathbb{G}) \nrightarrow ((\mathbb{V} \times \mathbb{G}) \times \wp(\mathbb{V} \times \wp(\mathbb{C}) \times \mathbb{G})) \\
\mathsf{minDAG}(\{(v, _, d)\} \uplus \{(*, _, \boxed{\mathsf{Abandoned}})\}) & & \\
 & = & \left((v, d), \left\{\left(*, \emptyset, \boxed{\mathsf{Abandoned}}\right)\right\}\right) \\
\mathsf{minDAG}(\{(v, _, d)\}) & = & ((v, d), \emptyset)
\end{array}
$$

Filter predicate suggestions that change a given variable

$$
\begin{array}{lcl}
\mathsf{filterSug} & : & \mathcal{X} \times \mathbb{G} \rightarrow \wp(\wp(\mathbb{S})) \\
\mathsf{filterSug}(x, \bigcirc\!\!\!\!\wedge\ ds) & = & \overline{\otimes}\{\mathsf{filterSug}(x, d) \mid d \in ds\} \\
\mathsf{filterSug}(x, \bigcirc\!\!\!\!\vee\ ds) & = & \bigcup_{d \in ds} \mathsf{filterSug}(x, d) \\
\mathsf{filterSug}\left(x, \boxed{\mathsf{Abandoned}}\right) & = & \{\emptyset\} \\
\mathsf{filterSug}\left(x, \boxed{p\ b\ sss}\right) & = & \{\{s \mid s \in ss \wedge \mathsf{var}(x, s)\} \mid ss \in sss\} \\
\mathsf{var}(x, x'\ [_]) & \Leftrightarrow & x = x' \\
\mathsf{var}(x, x'.l\ [_]) & \Leftrightarrow & x = x' \\
\mathsf{var}(_, _) & \Leftrightarrow & \mathsf{False}
\end{array}
$$

Apply a predicate suggestion set

$$
\begin{array}{lcl}
\mathsf{applySugs} & : & \mathbb{V} \times \wp(\mathbb{S}) \nrightarrow \mathbb{V} \\
\mathsf{applySugs}(v, sugs) & = & \mathsf{fold}(\mathsf{applySug}, v, sugs) \quad \text{if } sugs \text{ do not conflict} \\
\mathsf{applySug}(v, x\ \ [v_0 \rightsquigarrow v_1]) & = & v_1 \\
\mathsf{applySug}(K_R\ \{\overline{l_i = v_i}\}, x.l\ \ [v \rightsquigarrow v']) & & \\
 & = & K_R\ \ \{\overline{l_i = v_i}\} \setminus \{l = _\} \ \cup\ \{l = v'\} \\
\mathsf{applySug}(v, _) & = & v
\end{array}
$$

Create a value skeleton of a given type

$$
\begin{array}{lcl}
\mathsf{mkSkeleton} & : & \mathbf{T} \rightarrow \mathbb{V}^{\perp} \\
\mathsf{mkSkeleton}(R_n\ _) & = & K_R\ \ \{\overline{l_i = \perp}\} \\
\qquad \text{where } \{\overline{l_i}\} = \widehat{R} & & \\
\mathsf{mkSkeleton}(\tau) & = & \perp
\end{array}
$$

Figure 8.25: Auxiliary functions for S-DAG augmentation ($\overline{\otimes}$ denotes the cartesian product plus binary union)

For a universal node, we augment its sub-DAGs. Then, for each edge, we collect the predicate suggestions affecting the quantifier variable. If the edge's sub-DAG does not contain sufficient suggestions, we derive an action to delete the affected value (provided that the universally quantified variable is not annotated by CHG). Otherwise, we apply the predicate suggestions to the affected value, similar to above.

8.6 Summary

In this chapter, we generate concrete repair actions and visualize them via S-DAGs, which provide the key to make our approach to repair generation feasible. In contrast to database constraint maintenance approaches, we suggest repair actions only; we do not aim at automatic repair. This is mainly, because, in general, we cannot anticipate where to apply a repair action. Annotations from the rule designer (hints, repair costs, quantifier annotations), and thus

incorporation of domain knowledge, provide the basis for generating useful repair actions and lowering computational complexity. Hints provide a suitable basis for flexible inconsistency handling. Our approach follows the principle of minimal change and tries to preserve the changes made to the repository. We divide S-DAG generation in two parts: (1) during consistency checking S-DAGs are generated with predicate suggestions only; (2) at author request S-DAGs are augmented by concrete repair actions.

S-DAGs generated during consistency checking contain minimal information from which the actual repair actions can be derived. An S-DAG already contains some more information than a consistency report, namely predicate suggestions and explicit value bindings for existential quantifiers. By S-DAG *reduction* we loose redundant actions and actions that are considered too expensive by the rule designer. This is a major contribution of our S-DAG approach. Recall that we target at *interactive* repair; therefore, we generate a few repair actions only. S-DAGs are generated incrementally, similar to consistency reports.

Our system augments S-DAGs on author demand. Then augmented S-DAGs (one S-DAG for each rule) can be used for interactive repair. Authors may choose actions from S-DAGs. The effect of an action can be seen by applying it to the S-DAG. It is, however, still unclear what impact an action has to other rules. An action could also resolve inconsistencies of other rules or introduce new inconsistencies.

Our S-DAG approach goes beyond consistency reports, which just show inconsistencies. S-DAGs visualize repair actions for a rule in a convenient way to authors and can be generated as fast as consistency reports. In particular, we avoid the explosion of repair alternatives. S-DAGs provide, however, no means to explore the interaction of repair actions for different rules. In the next chapter, we eliminate this weakness by deriving a single repair collection for all rules from their S-DAGs.

Chapter 9
Repair Collections

Whereas S-DAGs provide a computationally tractable approach to generating useful repair actions, they suffer from an inherent weakness: Authors choose actions for each rule *separately*, independent of their effect on other rules. This becomes tedious when many rules are violated since the interaction between actions and potential negative impacts of an action towards overall consistency remain unclear. In this chapter, we derive a *single* repair collection[1] for *multiple* consistency rules from their S-DAGs. The repair collection contains alternative repair sets, each of which includes repairs that are necessary to resolve all inconsistencies in the repository. Our approach guarantees that (1) each repair set only contains repairs that do not contradict each other, and (2) the repair sets provide mutually independent alternatives. The collection can be sorted w.r.t. user-defined preference metrics, which are based on repair ratings. We rate a repair according to its cost, the inconsistencies it resolves, and the rules that may be violated by applying it. From the repair collection, authors can choose a repair set of a high ranking and then manually apply its repairs to the documents in the repository. Often, time and cost restrictions prohibit the resolution of all inconsistencies at once. Therefore, we support *partial inconsistency resolution*. Instead of applying a complete repair set, authors can choose those repairs that resolve the most troubling inconsistencies at a small cost. Thus, our approach facilitates flexible inconsistency management. We proceed as shown in Fig. 9.2:

1. From each S-DAG, we derive a repair collection.
2. We merge all repair collections to an overall repair collection, which contains repair sets that resolve inconsistencies for all rules.
3. We sort this repair collection w.r.t. user-defined metrics.

This chapter is organized as follows: First, we give an informal overview. In Sect. 9.2, we show how we derive an overall repair collection from the S-DAGs

[1]Recall that by "repair collection" we mean a set of sets of repairs.

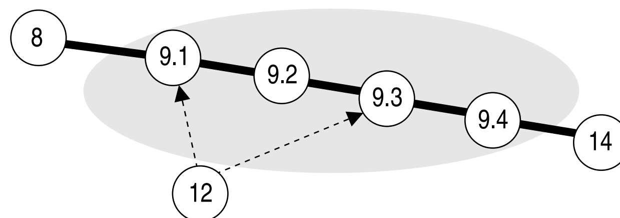


Figure 9.1: Chapter 9 in context

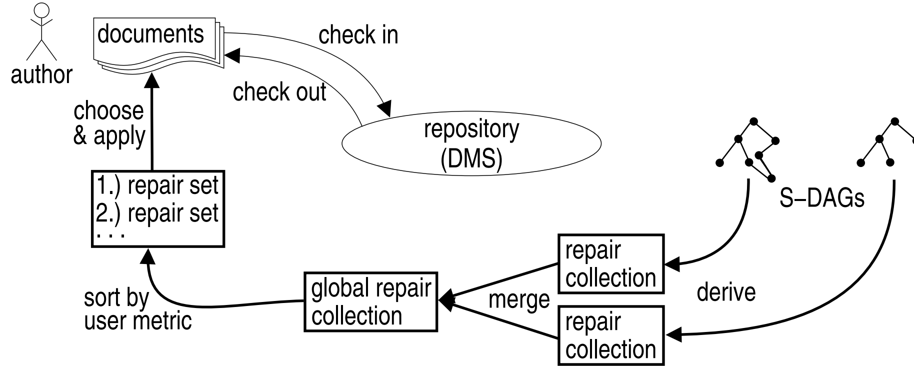


Figure 9.2: Repairing inconsistencies (overview)

of the individual rules. In Sect. 9.3, we put repair collections to practice, i.e., we sort them by user-defined metrics and show how a sorted repair collection can be used to resolve inconsistencies. Sect. 9.4 summarizes this chapter. Fig. 9.1 illustrates the context of this chapter. Performance tests for deriving repair collections for a larger example can be found in Sect. 12.5 (pg. 211) — our running example turned out to be too simple for significant performance tests.

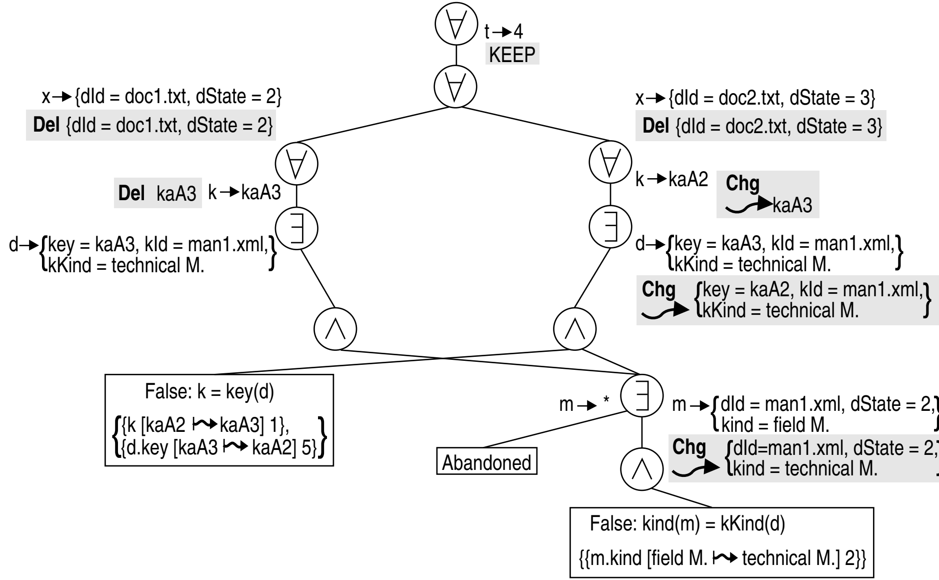
9.1 Informal Overview

Instead of using the trial and error process from Sect. 8.5 to determine a good repair set, we derive a single repair collection for all rules. Fig. 9.3 shows our example rules and their augmented S-DAGs at state 4 of our example repository (we use the miniscoped variants of ϕ_1 and ϕ_2, respectively).[2] We assign a high priority to ϕ_1 and a medium priority to ϕ_2. Recall that augmented S-DAGs represent inconsistencies and repair actions of the current repository state only.

From each S-DAG, we derive a repair collection. Fig. 9.4 shows the repairs we derive from our example S-DAGs. The first component of a repair contains its rules and its quantified variables (repairs may apply to multiple rules); the second component contains the sphere term of the quantified variable. The third component determines the context of a repair. The context contains bindings of all variables the repair depends on directly or indirectly. By these means we identify repairs. The fourth component denotes the proposed action. Finally, we *rate* repairs by (1) how many inconsistencies they resolve at which priority, (2) which rules may be impacted by the repair, and (3) the cost of the repair. Ratings are useful for sorting repairs by user-defined metrics. For example, we derive the repair rep_{Chgman1} for the rule ϕ_1 and the variable m. The repair suggests to change the kind of the manual man1.xml towards

[2]We use *augmented* S-DAGs, because actions provide useful information to guide repair derivation.

$$
\begin{aligned}
\phi_1^{\mathsf{high}} = \;& \forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\
& \exists\, d \in \texttt{concatMap}(\texttt{kDefs}, \texttt{repResDs}(t)) \bullet \\
& \quad k = \texttt{key}(d) \qquad \left\{ \begin{array}{l} \{k \rightsquigarrow \texttt{key}(d)\ \mathsf{False}\ \ 1\}, \\ \{d.\texttt{key} \rightsquigarrow k\ \mathsf{False}\ \ 5\} \end{array} \right\} \qquad \wedge \\
& \quad \exists\, m \in \texttt{repManDs}(t) \bullet \\
& \quad \texttt{dId}(m) = \texttt{kId}(d) \qquad \{\{m.\texttt{dId} \rightsquigarrow \texttt{kId}(d)\ \mathsf{False}\ \ 3\}\} \qquad \wedge \\
& \quad \texttt{kind}(m) = \texttt{kKind}(d) \qquad \{\{m.\texttt{kind} \rightsquigarrow \texttt{kKind}(d)\ \mathsf{False}\ 2\}\}
\end{aligned}
$$



$$
\begin{aligned}
\phi_2^{\mathsf{medium}} = \;& \forall\, t_1^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, t_2^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \\
& \neg(t_1 < t_2) \vee \\
& \left(\begin{array}{l} \forall\, m_1^{\mathsf{KEEP}} \in \texttt{repManDs}(t_1) \bullet \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \quad \texttt{dId}(m_1) = \texttt{dId}(m_2) \qquad \{\{m_2.\texttt{dId} \rightsquigarrow \texttt{dId}(m_1)\ \mathsf{False}\ \ 3\}\} \qquad \wedge \\ \quad \texttt{kind}(m_1) = \texttt{kind}(m_2) \quad \{\{m_2.\texttt{kind} \rightsquigarrow \texttt{kind}(m_1)\ \mathsf{False}\ \ 2\}\} \end{array} \right)
\end{aligned}
$$

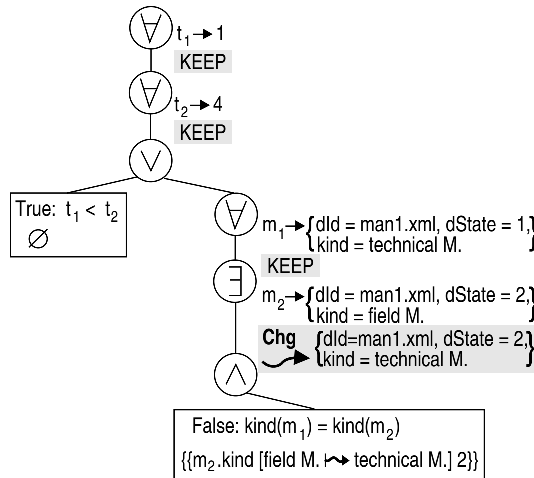


Figure 9.3: Augmented S-DAGs for rule ϕ_1 and ϕ_2, respectively, at state 4

$$
\begin{array}{lll}
rep_{\mathsf{Chgman1}} & = \mathsf{Rep}\ \{\phi_1(m)\} & \mathtt{repManDs}(t) \\
& \quad \{t \mapsto 4\} & \\
& \quad \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 2, \ldots\}.\mathtt{kind} \rightsquigarrow \mathsf{technical\ M.} & \\
& \quad \mathsf{Rate}\ \{2\ (\mathsf{high})\}\ \{\phi_1, \phi_2\}\ 2 & \\
rep_{\mathsf{Chgman1'}} & = \mathsf{Rep}\ \{\phi_2(m_2)\} & \mathtt{repManDs}(t) \\
& \quad \{t \mapsto 4\} & \\
& \quad \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 2, \ldots\}.\mathtt{kind} \rightsquigarrow \mathsf{technical\ M.} & \\
& \quad \mathsf{Rate}\ \{1\ (\mathsf{medium})\}\ \{\phi_1, \phi_2\}\ 2 & \\
rep_{\mathsf{Chgdoc2}} & = \mathsf{Rep}\ \{\phi_1(k)\} & \mathtt{refs}(x) \\
& \quad \{t \mapsto 4, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\} & \\
& \quad \mathsf{Chg\ kaA2} \rightsquigarrow \mathsf{kaA3} & \\
& \quad \mathsf{Rate}\ \{1\ (\mathsf{high})\}\ \{\phi_1\}\ 1 & \\
rep_{\mathsf{Chgkeys}} & = \mathsf{Rep}\ \{\phi_1(d)\} & \mathtt{concatMap}(\mathtt{kDefs}, \mathtt{repResDs}(t)) \\
& \quad \{t \mapsto 4\} & \\
& \quad \mathsf{Chg}\ \{\mathtt{key} = \mathsf{kaA3}, \ldots\}.\mathtt{key} \rightsquigarrow \mathsf{kaA2} & \\
& \quad \mathsf{Rate}\ \{1\ (\mathsf{high})\}\ \{\phi_1\}\ 5 & \\
rep_{\mathsf{DelkaA2}} & = \mathsf{Rep}\ \{\phi_1(k)\} & \mathtt{refs}(x) \\
& \quad \{t \mapsto 4, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\} & \\
& \quad \mathsf{Del\ kaA2} & \\
& \quad \mathsf{Rate}\ \{2\ (\mathsf{high})\}\ \{\phi_1\} & \\
rep_{\mathsf{DelkaA3}} & = \mathsf{Rep}\ \{\phi_1(k)\} & \mathtt{refs}(x) \\
& \quad \{t \mapsto 4, x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 2\}\} & \\
& \quad \mathsf{Del\ kaA3} & \\
& \quad \mathsf{Rate}\ \{1\ (\mathsf{high})\}\ \{\phi_1\} & \\
rep_{\mathsf{Deldoc1}} & = \mathsf{Rep}\ \{\phi_1(x)\} & \mathtt{repDs}(t) \\
& \quad \{t \mapsto 4\} & \\
& \quad \mathsf{Del}\ \{\mathtt{dId} = \mathsf{doc1.txt}, \mathtt{dState} = 2\} & \\
& \quad \mathsf{Rate}\ \{1\ (\mathsf{high})\}\ \{\phi_1\} & \\
rep_{\mathsf{Deldoc2}} & = \mathsf{Rep}\ \{\phi_1(x)\} & \mathtt{repDs}(t) \\
& \quad \{t \mapsto 4\} & \\
& \quad \mathsf{Del}\ \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\} & \\
& \quad \mathsf{Rate}\ \{2\ (\mathsf{high})\}\ \{\phi_1\} &
\end{array}
$$

Figure 9.4: Repairs generated for the rules ϕ_1 and ϕ_2

technical M., where man1.xml resides in the sphere obtained by `repManDs` applied to state 4. The repair resolves two inconsistencies for high priority rules and imposes a cost of 2; applying rep_{Chgman1} might violate rules ϕ_1 and ϕ_2.

From the augmented S-DAG for rule ϕ_1, we derive the following repair collection. Each of the repair sets below resolves all inconsistencies for ϕ_1.

$$
\left\{
\begin{array}{l}
\{rep_{\mathsf{Deldoc1}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{Deldoc1}}, rep_{\mathsf{DelkaA2}}\}, \{rep_{\mathsf{Deldoc2}}, rep_{\mathsf{Chgman1}}\}, \\
\{rep_{\mathsf{Chgkeys}}, rep_{\mathsf{Chgman1}}\}, \{rep_{\mathsf{Chgdoc2}}, rep_{\mathsf{Chgman1}}\}, \{rep_{\mathsf{DelkaA2}}, rep_{\mathsf{Chgman1}}\}, \\
\{rep_{\mathsf{DelkaA3}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{DelkaA3}}, rep_{\mathsf{DelkaA2}}\}
\end{array}
\right\}
$$

From the augmented S-DAG for ϕ_2, we derive the repair collection

$$
\{\ \{rep_{\mathsf{Chgman1'}}\}\ \}
$$

We combine these collections to an overall repair collection for both rules:

$$\left\{ \begin{array}{l} \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{DelkaA2}}\}, \\ \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgkeys}}\} \end{array} \right\}$$

The repair $rep_{\mathsf{Chgman1''}}$ results from merging rep_{Chgman1} and $rep_{\mathsf{Chgman1'}}$, which essentially propose the same action. Of course, $rep_{\mathsf{Chgman1''}}$ resolves inconsistencies for both rules ϕ_1 and ϕ_2:

$$\begin{array}{lll} rep_{\mathsf{Chgman1''}} = \mathsf{Rep} & \{\phi_1(m), \phi_2(m_2)\} & \mathtt{repManDs}(t) \\ & \{t \mapsto 4\} & \\ & \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 2, \ldots\}.\mathtt{kind} \rightsquigarrow \mathsf{technical\ M.} & \\ & \mathsf{Rate}\ \{2\ (\mathsf{high}), 1(\mathsf{medium})\}\ \{\phi_1, \phi_2\}\ 2 & \end{array}$$

In the above repair collections, each set is a *real alternative*, i.e., it is not a subset of another set in the collection. Within each repair set, all repairs are *compatible*, i.e., they can be applied altogether. Our approach guarantees these properties. Repairs that change document content are compatible to the document structure. In contrast, we cannot guarantee that repairs proposing to insert content or to delete content are compatible to the document structure (we have already discussed this issue in Sect. 8.5.3).

Finally, we sort the overall repair collection by a user-defined preference metric, which results in:

1.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgdoc2}}\}$,
2.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgkeys}}\}$,
3.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{DelkaA2}}\}$,
4.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Deldoc2}}\}$

Thus, authors would probably choose to change the key reference kaA2 to kaA3 in the document doc1.txt and to change the kind of the manual man1.xml to technical M. Above, we have considered changing content preferable to deleting content in a document. This decision depends, however, on the actual application.

9.2 Deriving Repair Collections from S-DAGs

In this section, we define algorithms that derive a single repair collection for a whole rule system from the augmented S-DAGs of the individual rules. In the repair collection, each set is an alternative; within each repair set, all repairs must be applied simultaneously, in order to resolve all inconsistencies in the repository. From the collection, authors may choose an arbitrary repair set.

In Sect. 9.2.1, we introduce our formal notion of "repair" and other technical prerequisites. Deriving the repair collection for a rule system requires to derive a repair collection for each rule individually, which we explain in Sect. 9.2.2. Sect. 9.2.3 presents a straightforward method for deriving a repair collection for a whole rule system. We exemplify our algorithms in Sect. 9.2.4. Sect. 9.2.5 is devoted to repair ratings. In the derivation of repair collections, *hitting*

Repairs $\mathbb{R}$	
$rep ::= \mathsf{Rep}\ \{\overline{i\ (x_i)}\}\ e\ \eta\ act\ rate$	perform action act in the sphere e for the variables x_i in rules i under the context η
Actions (extended from Fig. 8.11 pg. 109)	
$act ::= \mathsf{Add}\ v \mid \mathsf{Del}\ v$	add value v; delete value v
$\mid \mathsf{Chg}\ v_1 \rightsquigarrow v_2$	change value v_1 to v_2
$\mid \mathsf{Chg}\ v_1.l \rightsquigarrow v_2$	change component l in value v_1 to v_2
$\mid\ ??\ [b:\ \phi]$	change the free variables in the atomic formula ϕ, such that its truth value is b
Ratings	
$rate ::= \mathsf{Rate}\ \{\overline{n_j\ (pr_j)}\}\ \{\bar{i}\}\ c$	resolve n_j inconsistencies at priority pr_j, impacted rules i, cost c
$pr ::= \mathsf{high} \mid \mathsf{medium} \mid \mathsf{low}$	priority of a rule

Figure 9.5: Abstract syntax of repairs $\mathbb{R}$

collections play a central rôle; we show our adaptations to a well-known hitting set algorithm [GSW89] in Sect. 9.2.6.[3]

9.2.1 Repairs: Compatibility and Subsumption

A repair represents an action that can resolve inconsistencies and specifies where this action takes place, i.e., the rule and the sphere. Formally, we define repairs as shown in Fig. 9.5. A repair carries rule identifiers i and variables x_i (for which the repair resolves inconsistencies), a term e and a context η (representing the sphere in which the repair takes place), an action act (representing the actual change to the repository), and a rating $rate$. The context η contains bindings to all variables the free variables of e depend on, directly or indirectly.[4] The bindings in η are part of the augmented S-DAG already; they do not need to be recomputed. In order to lower computational complexity and to avoid access to the repository, we store the sphere term e only; the sphere is not re-computed. The set of all repairs is denoted by $\mathbb{R}$. We extend the formal definition of actions from Fig. 8.11 (pg. 109). Similar to predicate suggestions, we also support partial changes that apply to record components only. An action $??\ [b:\ \phi]$ means to change the variables free in the atomic formula ϕ, such that ϕ results in the truth value b. Actions are generated from predicate suggestions in S-DAG leaves. A *rating* indicates how many inconsistencies n_j are resolved for rules with a priority pr_j, which rules i could potentially be broken, and what cost c is imposed by the repair. For our purposes, a simple priority model (using three levels only) has proved sufficient.

For example, from the S-DAG for ϕ_1 we can derive the following repairs for the variable k:

[3] Throughout, we use the term "hitting collection" to mean a special repair collection.

[4] A variable z depends on a variable y iff z's quantifier sphere term contains y. Computing the transitive closure of the depends-on relation, we find the set of all variables a term e depends on.

$$
\begin{array}{lll}
\mathit{rep}_{\mathsf{Chgdoc2}} & = & \mathsf{Rep}\ \{\phi_1(k)\} \qquad\qquad \mathtt{refs}(x) \\
& & \{t \mapsto \mathsf{4}, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\} \\
& & \mathsf{Chg\ kaA2} \rightsquigarrow \mathsf{kaA3} \qquad\qquad \mathsf{Rate}\ \{1\ (\mathsf{high})\}\ \{\phi_1\}\ 1 \\
\mathit{rep}_{\mathsf{DelkaA2}} & = & \mathsf{Rep}\ \{\phi_1(k)\} \qquad\qquad \mathtt{refs}(x) \\
& & \{t \mapsto \mathsf{4}, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = 3\}\} \\
& & \mathsf{Del\ kaA2} \qquad\qquad \mathsf{Rate}\ \{2\ (\mathsf{high})\}\ \{\phi_1\}
\end{array}
$$

Above, $\mathit{rep}_{\mathsf{Chgdoc2}}$ suggests to change the key kaA2 to kaA3 in the document doc2.txt at state 4. The repair resolves one inconsistency for a high priority rule, which might cause inconsistencies for the rule ϕ_1 and imposes a cost of 1. In contrast, $\mathit{rep}_{\mathsf{DelkaA2}}$ suggests to delete the key kaA2 from doc2.txt.

Clearly, the repairs above must not appear in the same repair set: We cannot change the key kaA2 to kaA3 and at the same time delete the key kaA2. Similarly, two repairs that change a value in different ways contradict each other. Obviously, the repairs within a repair set must be compatible. Two repairs are *compatible* iff (1) they affect different spheres, or (2) they affect different values in the same sphere, or (3) they affect the same sphere and the same values and their proposed actions are unifyable. Two repairs $\mathsf{Rep}\ _\ e_1\ \eta_1\ _\ _$ and $\mathsf{Rep}\ _\ e_2\ \eta_2\ _\ _$ affect the same sphere, if their sphere terms and contexts are equal, i.e., $e_1 = e_2$ and $\eta_1 = \eta_2$. For deciding whether two actions are unifyable, we have developed a unification algorithm for actions. It works like well known unification algorithms in type systems [HS86, Mit90] or resolution calculi [Apt90]. For example, our unification algorithm prohibits to unify actions that propose to delete a value v on the one hand, and propose to change a value v_1 to a value v_2 on the other hand, if either v equals v_1 (as in the example above) or v equals v_2. For brevity, we omit the formal definition of our unification algorithm.

It is possible to merge compatible repairs that propose actions to the same value within the same sphere. Unless the repairs propose the *same* action, we are reluctant to merge them, because we want to derive "small" repairs, in order to facilitate *partial inconsistency resolution* (see Sect. 9.3.2). Merging repairs too early provides too coarse a picture of possible repairs. Of course, repairs can be merged on author demand once the repair collection has been computed.

Our unification algorithm can also be used to determine whether a repair subsumes another repair. Formally, repair rep_1 with action act_1 *subsumes* repair rep_2 with action act_2 iff the actions act_1 and act_2 unify to act_1. Intuitively, the changes to the repository performed by act_2 are all included in act_1 already. In the below example, act_1 subsumes act_2:

$$
\begin{array}{lll}
\mathit{act}_1 & = & \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 2, \mathtt{kind} = \mathsf{field\ M.}\} \\
& & \quad \rightsquigarrow \{\mathtt{dId} = \mathsf{man2.xml}, \mathtt{dState} = 2, \mathtt{kind} = \mathsf{technical\ M.}\} \\
\mathit{act}_2 & = & \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = 2, \mathtt{kind} = \mathsf{field\ M.}\}.\mathtt{kind} \rightsquigarrow \mathsf{technical\ M.}
\end{array}
$$

Both compatibility and subsumption easily lift to repair sets: A repair set rs is compatible to a repair set rs', if all repairs in $\mathit{rs} \cup \mathit{rs}'$ are compatible to each other. A repair set rs is subsumed by a repair set rs', if each repair in rs is subsumed by a repair in rs'. Intuitively, rs' requires more changes to

the repository than *rs*. Then, if both repair sets can be used alternatively to resolve inconsistencies it is sufficient to keep *rs* only. We need the notions of subsumption and compatibility for repair sets in the following sections to derive useful repairs that do not contradict each other.

9.2.2 A Repair Derivation Algorithm

We derive repairs for individual rules following our initial ideas shown in Sect. 7.2. Instead of deriving repairs during consistency checking, we use augmented S-DAGs. This has two advantages:

1. The computationally expensive derivation of repairs is separated from consistency checking, i.e., the generation of S-DAGs. We avoid accessing the repository and, therefore, do not need to lock it. We do not compute any values during the derivation of repairs. Instead, we look up values in the augmented S-DAG.

2. In augmented S-DAGs, actions provide useful information about which kinds of repairs require minimal changes to the repository. Thus, our algorithm to derive a repair collection becomes more efficient and results in more compact repair collections.

Since we tolerate inconsistencies, the invalidation of repair collections by new check-ins to the repository is not an issue.

Recall our basic strategy: From each leaf in an S-DAG, we obtain a basic repair collection. Below disjunction and existential nodes, respectively, we join the repair collections derived from the target S-DAGs. Below conjunction and universal nodes, respectively, we compute (something like) the cartesian product of the repair collections derived from the target S-DAGs.

Our derivation algorithm walks through an S-DAG from the root to the leaves. The variable bindings from quantifier edges are collected. A quantifier action provides information about the kind of repair to be derived from a predicate suggestion in a leaf. For example, if a quantifier action proposes to change a variable x then for x we derive repairs to change values. We derive repair collections from the leaves, in order to determine repairs that must be applied together to resolve inconsistencies. Quantifier actions are not sufficient for this purpose.

Fig. 9.6 shows the denotational semantics of our repair derivation algorithm. A *variable mode* set μ associates each variable with a mode: Add, Chg, Del, or KEEP (meaning obvious). Variable modes are obtained from quantifier actions. We can now define the function $\mathcal{REP}\,(d)\ \ [\eta, \mu]$, which derives a repair collection from the S-DAG d for the variable assignment η and the variable mode set μ. For the variable assignment η, we use the non-incremental definition from Fig. 5.9 (pg. 61). Initially, $\mathcal{REP}$ is applied to an empty variable assignment and an empty mode set; i.e., $\mathcal{REP}\,(d)\ \ [\emptyset, \emptyset]$ derives the repair collection for the S-DAG d. Metadata, required for repairs, depend on the rule ϕ itself,

global constants: ϕ original rule
i rule identifier of ϕ
D S-DAG for ϕ

$$
\begin{array}{lll}
\mathcal{REP} & : & \mathbb{G} \times \mathbb{E} \times \wp(\mathcal{X} \times \mathcal{M}) \to \wp(\wp(\mathbb{R})) \\
\mathcal{REP}\ (\boxed{\mathsf{Abandoned}})\ [\eta, \mu] & = & \{\emptyset\} \\
\mathcal{REP}\ \left(\boxed{p\ b\ sss}\right)\ [\eta, \mu] & = & \{\{\mathsf{mkRep}(s, \phi, i, \eta, \mu, D) \mid s \in ss\} \mid ss \in sss\} \\
\mathcal{REP}\ (\bigcirc\!\!\!\!\wedge\ es)\ [\eta, \mu] & = & \mathsf{hittingColl}(rsss) \\
\quad \text{where } rsss = \{\mathcal{REP}\ (d)\ [\eta, \mu] \mid (_, d) \in es\} & & \\
\mathcal{REP}\ (\bigcirc\!\!\!\!\vee\ es)\ [\eta, \mu] & = & \mathsf{rmSubsumes}(rss) \\
\quad \text{where } rss = \bigcup_{(_, d) \in es} \mathcal{REP}\ (d)\ [\eta, \mu] & & \\
\mathcal{REP}\ (\bigcirc\!\!\!\!\forall\ x\ es\ an)\ [\eta, \mu] & = & \mathsf{hittingColl}(rsss') \\
\quad \text{where } rsss\ = \{(v, \eta', \mathcal{REP}\ (d)\ [\eta', \mu']) \mid (v, acts, d) \in es\} & & \\
\qquad\qquad \text{where } \eta'\ = \eta \cup \{x \mapsto v\} & & \\
\qquad\qquad\qquad \mu'\ = \mu \cup \{(x, \mathsf{mode}(acts))\} & & \\
\qquad\quad rsss' = \{rss \cup del \mid (v, \eta', rss) \in rsss\} & & \\
\qquad\qquad \text{where } del = \emptyset \qquad\qquad \text{if } an = \mathsf{KEEP} \text{ or } an = \mathsf{CHG} & & \\
\qquad\qquad\qquad \{\mathsf{mkDelRep}(x, v, \phi, i, \eta', D)\} \text{ otherwise} & & \\
\mathcal{REP}\ (\bigcirc\!\!\!\!\exists\ x\ es)\ [\eta, \mu] & = & \mathsf{rmSubsumes}(rss) \\
\quad \text{where } rss = \bigcup_{(v, acts, d) \in es} \mathcal{REP}\ (d)\ [\eta', \mu'] & & \\
\qquad\qquad \text{where } \eta'\ = \eta \cup \{x \mapsto v\} & & \\
\qquad\qquad\qquad \mu'\ = \mu \cup \{(x, \mathsf{mode}(acts))\} & &
\end{array}
$$

Figure 9.6: Deriving repair collections from S-DAGs ($\mathcal{M}$ denotes variable modes)

its identifier i, and its whole S-DAG D; these are global constants for $\mathcal{REP}$. Auxiliary functions are defined in Fig. 9.7 (pg. 139).

For each predicate suggestion in a leaf, we derive a repair with the help of mkRep. A suggestion for a variable x is "applied" to the value of x as given by the current variable assignment η. The mode of x, as determined by the mode set μ, is responsible for the kind of the repair. Assume that we want to derive the repair collection for the right hand leaf in the S-DAG for ϕ_1 (see Fig. 9.3) carrying the suggestion collection $\{\{m.\mathtt{kind}\ [\text{field M.} \rightsquigarrow \text{technical M.}]\ 2\}\}$. The variable mode of m is Chg; therefore, we propose to change the kind of the current value of m to technical M. The rule ϕ, its identifier i, and its S-DAG D are all needed to complete the repair: e.g., its rating is determined by applying it to the S-DAG D (see Sect. 9.2.5).

For a conjunction node and a universal node, respectively, we compute the repair collections *rsss* for the S-DAGs below it and determine their hitting collection. The *hitting collection* is the smallest repair collection in which each repair set subsumes at least one repair set from each repair collection in *rsss*. Thus, the repair sets in the hitting collection are necessary to repair the conjunction and universal quantification, respectively. Recall that below a conjunction node or a universal node all inconsistencies must be repaired. Our

hitting collection algorithm takes care that the repairs within the repair sets of the hitting collection are compatible, i.e., they can be applied altogether. For example, if we compute the following repair collections from three S-DAGs below a universal node

$$rsss = \left\{ \begin{array}{l} \{\{rep_1, rep_2\}, \{rep_2, rep_3\}\}, \\ \{\{rep_4\}, \{rep_1, rep_3\}\}, \\ \{\{rep_5, rep_6\}, \{rep_2\}\} \end{array} \right\}$$

then the hitting collection is (provided that repairs in each set are compatible):

$$\mathsf{hittingColl}(rsss) = \{\ \{rep_1, rep_2, rep_4\}, \{rep_1, rep_2, rep_3\}, \{rep_2, rep_3, rep_4\}\ \}$$

The repairs rep_5 and rep_6 do not contribute to the hitting collection, because their alternative rep_2 is always needed — it is a member of both sets in the first collection of *rsss*. If, however, the repairs rep_1 and rep_4 contradicted each other, the first repair set above would not be part of the hitting collection. Our hitting collection algorithm, defined in Sect. 9.2.6, is an adaptation of a well-known hitting set algorithm [GSW89].

For disjunction nodes and existential nodes, respectively, we join the repair collections from the S-DAGs below them. This is sufficient, because only one of these S-DAGs must be repaired, but it is unclear which one. Finally, we remove those repair sets that subsume other repair sets with the help of rmSubsumes. Clearly, if a subsumed repair set is already sufficient to resolve inconsistencies, the "larger" repair set is not required anymore.

For both kinds of quantifier nodes, we extend the variable assignment η and the variable mode set μ. The variable assignment η is extended by the binding $x \mapsto v$, where the variable x is obtained from the quantifier node and the value v is obtained from the edge processed. The variable mode set μ is extended by the mode of the actions for v. For universal quantifiers, we also derive repairs to delete the offending values with the help of mkDelRep. Clearly, deleting document content or even whole documents are "fall-back" solutions. In contrast to S-DAG augmentation, we generate these fall-back solutions always for universal quantifiers. Dropping these repairs too early could result in "un-repairable" S-DAGs due to incompatible repairs. Usually, repairs proposing to delete content rank among the least positions in the sorted repair collection, if there are better alternatives; see Sect. 9.3.

Fig. 9.7 contains the definitions of some auxiliary functions necessary for repair derivation. The function determineRating is described in Sect. 9.2.5; hittingColl is defined in Sect. 9.2.6.

9.2.3 Merging Repair Collections from Different Rules

We now want to derive a repair collection for both rules, ϕ_1 and ϕ_2. Our approach is simple: We consider a rule system as a conjunction of the individual rules and employ the strategy for conjunctions from the previous section. Thus, we determine the repair collection for each rule separately and then compute the hitting collection of all these collections. We must, however, adapt the

Derive a repair from a predicate suggestion

$$
\begin{array}{lll}
\mathsf{mkRep} & : & \mathbb{S} \times \mathcal{F} \times \mathbb{I} \times \mathbb{E} \times \wp(\mathcal{X} \times \mathcal{M}) \times \mathbb{G} \to \mathbb{R} \\
\mathsf{mkRep}(x\ [v \rightsquigarrow v']\ c, \phi, i, \eta, ms, d) & = & \mathsf{Rep}\ \{i(x)\}\ e\ \eta'\ act\ r \\
\quad \text{where } e & = & \mathsf{domVar}(x, \phi) \\
\quad \eta' & = & \eta_{[\mathsf{depVars}(\phi, e)]} \\
\quad act & = & \mathsf{Add}\ v' \quad \text{if } (x, \mathsf{Add}) \in ms \\
 & & \mathsf{Chg}\ v \rightsquigarrow v' \quad \text{else if } (x, \mathsf{Chg}) \in ms \\
\quad r & = & \mathsf{determineRating}(x, c, d, act) \\
\mathsf{mkRep}(x.l\ [v \rightsquigarrow v']\ c, \phi, i, \eta, ms, d) & = & \mathsf{Rep}\ \{i(x)\}\ e\ \eta'\ act\ r \\
\quad \text{where } e & = & \mathsf{domVar}(x, \phi) \\
\quad \eta' & = & \eta_{[\mathsf{depVars}(\phi, e)]} \\
\quad v_x & = & \eta(x) \\
\quad v_0 & = & \mathsf{mkSkeleton}(\tau_x) \\
\quad v'' & = & \mathsf{applySug}(v_0, x.l\ [v \rightsquigarrow v']) \\
\quad act & = & \mathsf{Add}\ v'' \quad \text{if } (x, \mathsf{Add}) \in ms \\
 & & \mathsf{Chg}\ l.v_x \rightsquigarrow v' \quad \text{else if } (x, \mathsf{Chg}) \in ms \\
\quad r & = & \mathsf{determineRating}(x, c, d, act) \\
\mathsf{mkRep}(x\ [b:\ p], \phi, i, \eta, ms, d) & = & \mathsf{Rep}\ \{i(x)\}\ e\ \eta'\ act\ r \\
\quad \text{where } e & = & \mathsf{domVar}(x, \phi) \\
\quad \eta' & = & \eta_{[\mathsf{depVars}(\phi, e) \cup \mathsf{fv}(p)]} \\
\quad act & = & ??\ [b:\ p] \\
\quad r & = & \mathsf{determineRating}(x, 0, d, act)
\end{array}
$$

Generate a repair to delete a value

$$
\begin{array}{lll}
\mathsf{mkDelRep} & : & \mathcal{X} \times \mathbb{V} \times \mathcal{F} \times \mathbb{I} \times \mathbb{E} \times \mathbb{G} \to \mathbb{R} \\
\mathsf{mkDelRep}(x, v, \phi, i, \eta, d) & = & \mathsf{Rep}\ \{i(x)\}\ e\ \eta'\ act\ r \\
\quad \text{where } e & = & \mathsf{domVar}(x, \phi) \\
\quad \eta' & = & \eta_{[\mathsf{depVars}(\phi, e)]} \\
\quad act & = & \mathsf{Del}\ v \\
\quad r & = & \mathsf{determineRating}(x, 0, d, act)
\end{array}
$$

Remove repair sets that subsume other repair sets

$$
\begin{array}{lll}
\mathsf{rmSubsumes} & : & \wp(\wp(\wp(\mathbb{R}))) \to \wp(\wp(\wp(\mathbb{R}))) \\
\mathsf{rmSubsumes}(\emptyset) & = & \emptyset \\
\mathsf{rmSubsumes}(\{rs\} \uplus rss) & & \\
\quad = \mathsf{rmSubsumes}(rss) & & \text{if } \exists\ rs' \in rss \bullet rs' \sqsubseteq rs \\
\quad\ \ \{rs\} \cup \mathsf{rmSubsumes}(rss) & & \text{otherwise}
\end{array}
$$

Figure 9.7: Auxiliary functions for deriving repair collections; for mkSkeleton and applySug see Fig. 8.25 (pg. 127); omitted definitions: $\mathsf{domVar}(x, \phi)$ (sphere term of the variable x in the formula ϕ), $\mathsf{depVars}(\phi, e)$ (dependent variables of the term e in the formula ϕ), $\mathsf{mode}(acts)$ (variable mode determined by actions $acts$), $\eta_{[xs]}$ (domain restriction of the variable assignment η to the variables in the set xs), $rs' \sqsubseteq rs$ (the repair set rs' is subsumed by the repair set rs)

ratings of the repairs, because ratings are computed separately for each rule. Our hitting collection algorithm adapts the ratings of repairs that apply to multiple rules.

9.2.4 Examples

We demonstrate generation of the repair collection for rule ϕ_1 from its augmented S-DAG shown in Fig. 9.3 (pg. 131). Repairs can be found in Fig. 9.4 (pg. 132). Consider the right hand leaf at the following assignment and mode set (we follow the right hand path in the S-DAG):

$$\eta = \begin{Bmatrix} t \mapsto \mathsf{4}, x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \mathtt{dState} = \mathsf{3}\}, k \mapsto \mathsf{kaA2}, \\ d \mapsto \{\mathtt{key} = \mathsf{kaA3}, \mathtt{kId} = \mathsf{man1.xml}, \mathtt{kKind} = \ldots\}, \\ m \mapsto \{\mathtt{dId} = \mathsf{man1.xml}, \mathtt{dState} = \mathsf{2}, \mathtt{kind} = \mathsf{field\ M.}\} \end{Bmatrix}$$
$$m = \{(t, \mathsf{KEEP}), (x, \mathsf{Del}), (k, \mathsf{Chg}), (d, \mathsf{Chg}), (m, \mathsf{Chg})\}$$

From the leaf's suggestion collection $\{\{m.\mathtt{kind}\ [\mathsf{field\ M.} \rightsquigarrow \mathsf{technical\ M.}]\ 2\}\}$, we derive the repair collection $\{\{rep_{\mathsf{Chgman1}}\}\}$, which contains a repair to change the kind of the manual man1.xml, because the mode of the variable m is Chg. We obtain the rating for rep_{Chgman1} by applying it to ϕ_1's S-DAG in Fig. 9.3, which removes two inconsistencies (the resulting S-DAG is shown in Fig. 8.23, pg. 124). Above the leaf, the conjunction node and the existential node for m do not change the repair collection.

For the left hand leaf, which carries the predicate suggestion collection $\{\{k\ [\mathsf{kaA2} \rightsquigarrow \mathsf{kaA3}]\ 1\}, \{d.\mathtt{key}\ [\mathsf{kaA3} \rightsquigarrow \mathsf{kaA2}]\ 5\}\}$, we derive the repair collection $\{\{rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgkeys}}\}\}$. Each of these repairs resolves one inconsistency.

We now process the conjunction node located above this leaf and the existential node for m. For the conjunction node, we compute the hitting collection of both repair collections above, resulting in

$$\{\ \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\}\ \}\ .$$

The existential node for d does not change this collection. The universal node for k adds a repair set proposing to delete the key $\mathsf{kaA2}$ from the document doc2.txt, thus:

$$\{\ \{rep_{\mathsf{DelkaA2}}\}, \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\}\ \}$$

Proceeding to the universal node for x we have to consider two edges. For the right hand edge, we derive the above repair collection. For the left hand edge, we derive the collection $\{\{rep_{\mathsf{DelkaA3}}\}, \{rep_{\mathsf{Chgman1}}\}\}$. To the collection for the left hand edge, a repair set is added proposing to delete the document doc1.txt; to the collection for the right hand edge, a repair set is added proposing to delete the document doc2.txt. Then we compute the hitting collection of:

$$\{\{rep_{\mathsf{Deldoc1}}\}, \{rep_{\mathsf{DelkaA3}}\}, \{rep_{\mathsf{Chgman1}}\}\} \qquad \begin{matrix}\text{left hand edge} \\ x \mapsto \{\mathtt{dId} = \mathsf{doc1.txt}, \ldots\}\end{matrix}$$

$$\begin{Bmatrix} \{rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{DelkaA2}}\}, \\ \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\} \end{Bmatrix} \qquad \begin{matrix}\text{right hand edge} \\ x \mapsto \{\mathtt{dId} = \mathsf{doc2.txt}, \ldots\}\end{matrix}$$

This results in the final repair collection for ϕ_1 (the universal node for t does not change the below collection, because t is annotated by KEEP):

$$rss_{\phi_1} = \left\{ \begin{array}{l} \{rep_{\mathsf{Deldoc1}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{Deldoc1}}, rep_{\mathsf{DelkaA2}}\}, \\ \{rep_{\mathsf{DelkaA3}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{DelkaA3}}, rep_{\mathsf{DelkaA2}}\}, \\ \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{DelkaA2}}\}, \\ \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\} \end{array} \right\}$$

The repair set $\{rep_{\mathsf{Deldoc1}}, rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}$ is not a member of the hitting collection, because it subsumes the repair set $\{rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}$. For similar reasons, the hitting collection is lacking the sets $\{rep_{\mathsf{Deldoc1}}, rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\}$, $\{rep_{\mathsf{DelkaA3}}, rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgdoc2}}\}$, and $\{rep_{\mathsf{DelkaA3}}, rep_{\mathsf{Chgman1}}, rep_{\mathsf{Chgkeys}}\}$.

For rule ϕ_2, we compute the following repair collection from the augmented S-DAG shown in Fig. 9.3 (pg. 131).

$$rss_{\phi_2} = \{ \ \{rep_{\mathsf{Chgman1'}}\} \ \}$$

In order to obtain a repair collection for both rules ϕ_1 and ϕ_2, we compute the hitting collection of their final repair collections, thus:

$$rss_{\phi_1 \wedge \phi_2} = \left\{ \begin{array}{l} \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Deldoc2}}\}, \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{DelkaA2}}\}, \\ \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgdoc2}}\}, \{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgkeys}}\} \end{array} \right\}$$

The repair $rep_{\mathsf{Chgman1''}}$ results from merging rep_{Chgman1} and $rep_{\mathsf{Chgman1'}}$, which propose the same change to the repository:

$$\begin{array}{lll} rep_{\mathsf{Chgman1''}} = \mathsf{Rep} & \{\phi_1(m), \phi_2(m_2)\} & \mathtt{repManDs}(t) \\ & \{t \mapsto 4\} & \\ & \mathsf{Chg}\ \{\mathtt{dId} = \mathsf{man1.xml}, \ldots\}.\mathtt{kind} \rightsquigarrow \mathsf{technical\ M.} & \\ & \mathsf{Rate}\ \{2\ (\mathsf{high}), 1 (\mathsf{medium})\}\ \{\phi_1, \phi_2\}\ 2 & \end{array}$$

Next, we discuss how the rating of a repair is determined and how hitting sets are computed. In Sect. 9.3, we show how derived repairs can be employed to manage consistency in heterogeneous repositories.

9.2.5 Determining Repair Ratings

In this section, we assign ratings to repairs; i.e., we describe the auxiliary function determineRating used in Sect. 9.2.2. The rating of a repair *rep* considers (1) how many inconsistencies *rep* resolves at which priority, (2) which rules might be impacted by applying *rep*, and (3) the cost of the hint from which *rep* originates.

Recall a basic property of S-DAGs: A predicate leaf represents as many inconsistencies as there are paths from the S-DAG root to this leaf. In the S-DAG for ϕ_1 (see Fig. 9.3 pg. 131), the right hand leaf labeled $\mathtt{kind}(m) = \mathtt{kKind}(d)$ represents two inconsistencies; the left hand leaf labeled $k = \mathtt{key}(d)$ represents one inconsistency. In order to rate repairs, we apply each repair to its augmented S-DAG and count the number of inconsistencies that have disappeared. Application of repairs to augmented S-DAGs reduces to application of actions, described in Sect. 8.5. For example, changing the kind of the manual

man1.xml to technical M. removes the leaf labeled kind(m) = kKind(d) from the S-DAG. This causes the left hand side of the S-DAG in Fig. 9.3 to collapse. Thus, the repair resolves two inconsistencies.

Obviously, repairs can cause new inconsistencies. For example, changing the key definition for the key kaA3 to a key definition for a key kaA2 (as proposed by rep_{Chgkeys}) incurs new inconsistencies for the key reference kaA3. Prior to applying a repair to the documents in the repository we cannot anticipate such negative impacts. Therefore, we approximate the document changed by a repair. Then we match this document against the documents affected by a rule. Recall that we have assigned each rule with its relevant documents, in order to support rule filtering (see Sect. 6.2.1). In our example, ϕ_1 depends on all plain text documents and on all XML documents (symbolized by the regular expressions {*.txt,*.xml}), whereas ϕ_2 depends on manuals only ({man*.xml}).

We approximate the document changed by a repair as follows:

1. We inspect the repair action, which might directly apply to a value that has a document type, i.e., a subtype of Doc.
2. We inspect the repair context, which might contain variable bindings to documents.
3. We inspect metadata of function symbols used to calculate the sphere of the repair variable.

Consider, e.g., the repair rep_{Chgkeys}. In rule ϕ_1, the variable d depends on t only, which is not bound to a document. However, d is calculated with help of the function symbols concatMap, kDefs, repResDs, and repStates. The function symbol repResDs depends on the key resolvers in the repository {keys*.xml}.[5] Thus, rep_{Chgkeys} potentially violates rule ϕ_1 only.

9.2.6 Computing Hitting Collections

Hitting collections play an essential rôle in our algorithm for deriving repair collections from S-DAGs. In Sect. 9.2.2, we referred to the algorithm defined in this section by the auxiliary function hittingColl. Recall the basic properties of a hitting collection for a number of repair collections *rsss*:

- Each repair set in the hitting collection subsumes at least one repair set from each repair collection in *rsss*. This means that the hitting collection contains all repair sets necessary to remove all inconsistencies.
- No proper subcollection of the hitting collection satisfies the above property, i.e., there is no smaller collection containing repair sets that remove all inconsistencies.

[5] In Sect. 6.2.1, this information is added by the language designer with the help of static analysis.

The calculation of hitting collections[6] has received much attention in the field of diagnosis generation and constraint maintenance in databases [GL97]. For our purposes, we adapt the algorithm in [GSW89], which generates a hitting set for a set of sets of atomic items. For example, for the set $\{\{a, b\}, \{b, c\}, \{a, c\}, \{d\}\}$ the hitting set $\{\{a, c, d\}, \{a, b, d\}, \{b, c, d\}\}$ is computed. The algorithm relies on an equality relation between the atomic items. In our setting, however, we deal with repair *sets*, instead of atomic items. Contradicting and subsuming repairs, respectively, require modifications to the algorithm in [GSW89].

Assume we have already computed the repair collections below a conjunction node, or below a universal node, or for each rule in a rule system. This results in a set *rsss* of repair collections. We compute the hitting collection as follows.

1. Generate a hitting collection DAG (HC-DAG). In the HC-DAG, each node is labeled by either a repair collection from *rsss*, or $\surd$ (we have found a member of the hitting collection), or $\times$ (we have aborted generation, i.e., the node has been closed). Each edge is labeled by a repair set, chosen from the repair collection of its parent node.

2. From the HC-DAG, we derive the hitting collection by following each path from the root to a node labeled by $\surd$ and joining all repair sets in this path.

We generate the HC-DAG as follows. Differences to [GSW89] result from dealing with repair sets (instead of atomic items) and taking care of repair compatibility and repair subsumption.

1. Let D represent the growing HC-DAG. Generate the HC-DAG root, labeled by the first repair collection in *rsss*. We assume that *rsss* is ordered, e.g., by the corresponding S-DAGs. The root will be processed in Step 2 below.

2. Process the nodes in D in a breath first-order: We first process all nodes at one level *before* processing the node children. To process a node n do:

   (a) Compute $\mathsf{repairs}(n)$, defined as the union of all repair sets labeling edges on the path from the root to node n.

   (b) If in every repair collection $rss \in rsss$ there exists a repair set rs that is subsumed by $\mathsf{repairs}(n)$, then label n with $\surd$. Clearly, in this case each repair collection *rss* contains a repair set already represented by n. Otherwise, label n by the first repair collection rss' in *rsss* in which no repair set is subsumed by $\mathsf{repairs}(n)$.

   (c) If the node n is labeled by a repair collection *rss*, then for each repair set $rs \in rss$, generate a new downward edge labeled *rs*, if the repairs in *rs* are compatible to the repairs in $\mathsf{repairs}(n)$, collected before. This edge leads to a new node m with $\mathsf{repairs}(m) = \mathsf{repairs}(n) \cup rs$.

[6] In the literature, hitting collections are often called hitting sets. We use the term "hitting collection," in order to emphasize that a hitting collection is a special repair collection.

The new node m will be processed (labeled and expanded) after all nodes in the same generation as n have been processed. If each repair set $rs \in rss$ contradicts the repair set $\mathsf{repairs}(n)$, we immediately close the node n by re-labeling it with $\times$.

(d) When processing a node n, a repair *rep* can be subsumed by a repair $rep' \in \mathsf{repairs}(n)$. If *rep* and *rep'* apply to different rules, then we annotate the node n by explicit information about how the rating of *rep'* has to be adapted. Clearly, the inconsistencies resolved by *rep* contribute to the inconsistencies resolved by *rep'*. Consequently, the rating of *rep'* increases by the rating of *rep*.

In order to reduce the size of the HC-DAG, we employ the following pruning enhancements of [GSW89].

1. *Sharing Nodes*: The algorithm above refuses to create a new node m as descendant of a node n (Step 2c), if there exists a node n' in the HC-DAG D, such that $\mathsf{repairs}(n') = \mathsf{repairs}(n) \cup rs$. In that case, we let the *rs*-edge under n point to n'. This corresponds to the usual sharing in DAGs. Otherwise, a new node m is generated.

2. *Closing*: If there exists a node n', labeled by $\surd$ and $\mathsf{repairs}(n')$ is subsumed by $\mathsf{repairs}(n)$, then close the node n, i.e., label it by $\times$. We do not compute a label for the node n nor do we generate any successor nodes. Clearly, n does not contribute to the hitting collection, because the repairs represented by n' are already sufficient for resolving all inconsistencies.

3. *Pruning*: If the repair collection *rss* is to label a node and it has not been used previously, then attempt to *prune* the HC-DAG D as generated so far:

   (a) If there exists a node n', labeled by the repair collection $rss' \in rsss$, where $rss \subset rss'$, then re-label n' with *rss*. For each repair set $rs \in rss' \setminus rss$, the *rs*-edge emanating from n' is removed along with its target node and all of its descendants (except for those nodes that have another ancestor).

   (b) Delete the repair collection *rss'* from *rsss*.

The above algorithm can be used to calculate a hitting collection when we want to resolve *all* inconsistencies. For *partial* inconsistency resolution, however, we have to adapt the algorithm, in order not to loose repairs by subsumption:

- During HC-DAG generation:
  - In step 2b, we label the node n by $\surd$ only if in every repair collection $rss \in rsss$ there exists a repair set *rs* that is *a subset of* $\mathsf{repairs}(n)$.
  - In step 2d, we use equality, instead of repair subsumption.
- During HC-DAG pruning:

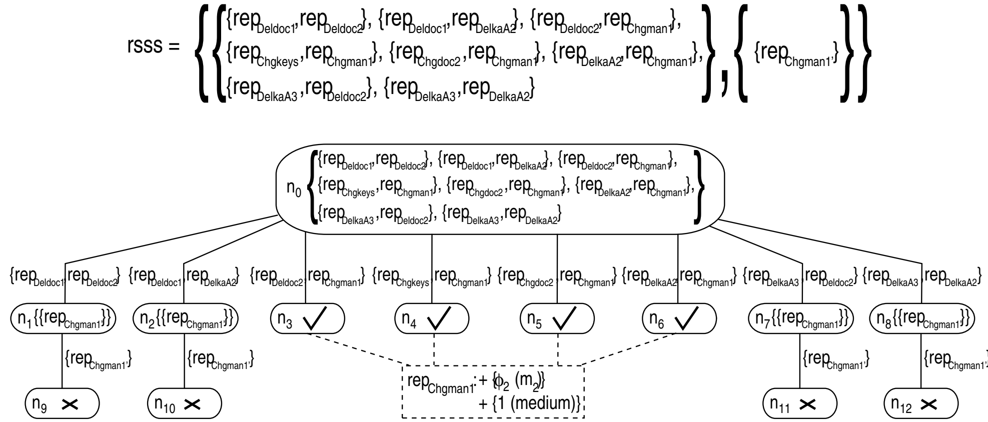


Figure 9.8: HC-DAG for our example rules ϕ_1 and ϕ_2

– In step 2, we close a node n only if there exists a node n', which is labeled by $\surd$ and $\mathsf{repairs}(n')$ is *a subset of* $\mathsf{repairs}(n)$.

In order to derive the overall repair collection for both rules ϕ_1 and ϕ_2, we generate the HC-DAG shown in Fig. 9.8. We have numbered the nodes from n_0 to n_{12}, where node numbers correspond to the order of node generation. The nodes n_3, n_4, n_5, and n_6 have no children nodes, because the repair $rep_{\mathsf{Chgman1'}}$ is essentially equal to the repair rep_{Chgman1}. We have annotated the nodes n_3 through n_6, in order to increase the ranking of the repair rep_{Chgman1}, which in turn results in the repair $rep_{\mathsf{Chgman1''}}$. The node n_9 has been closed, because the repair set $\mathsf{repairs}(n_3)$ is subsumed by $\mathsf{repairs}(n_9)$. For similar reasons, the nodes n_{10} through n_{12} have been closed. From the HC-DAG in Fig. 9.8, we derive the final hitting collection $rss_{\phi_1 \wedge \phi_2}$ by following each path from the root to a leaf labeled by $\surd$ and by adjusting the ranking of the repairs appropriately.

9.3 Repairing Repositories

Clearly, from a repair collection derived for a repository authors can choose an arbitrary repair set. Collections can, however, become huge for larger repositories, particularly for rule systems containing many rules. For example, in our case study we have derived a repair collection consisting of 430 sets each containing between 13 and 16 repairs. In this situation, one cannot reasonably expect authors to make a good choice. Therefore, we exploit repair ratings to reduce the number of reasonable choices from a repair collection. This alleviates the task of resolving inconsistencies and also supports partial inconsistency resolution, which is of vital importance when numerous inconsistencies occur while there is pressure of time and cost.

9.3.1 Using Repair Ratings

Our approach to reducing the number of repair choices is to sort the repair collection w.r.t. user-defined metrics that reflect user preferences. Formally,

Repair	rating_Σ			$\mathsf{rating}_{\max}$		
rep_{Chgman1}	$-2-2$	$=$	-4	$2\cdot5-2-2$	$=$	6
$rep_{\mathsf{Chgman1'}}$	$-2-2$	$=$	-4	$1\cdot3-2-2$	$=$	-1
$rep_{\mathsf{Chgman1''}}$	$-2-2$	$=$	-4	$2\cdot5+1\cdot3-2-2$	$=$	9
rep_{Chgdoc2}	$-1-1$	$=$	-2	$1\cdot5-1-1$	$=$	3
rep_{Chgkeys}	$-1-5$	$=$	-6	$1\cdot5-1-5$	$=$	-1
rep_{DelkaA2}	$-1-10$	$=$	-11	$2\cdot5-1-10$	$=$	-1
rep_{DelkaA3}	$-1-10$	$=$	-11	$1\cdot5-1-10$	$=$	-6
rep_{Deldoc1}	$-1-15$	$=$	-16	$1\cdot5-1-15$	$=$	-11
rep_{Deldoc2}	$-1-15$	$=$	-16	$2\cdot5-1-15$	$=$	-6

Table 9.1: Example ratings calculated by rating_Σ and $\mathsf{rating}_{\max}$, respectively, for repairs shown in Fig. 9.4 (pg. 132)

a *preference metric* is defined by a total pre-order between repair sets, i.e., a transitive and reflexive relation.[7] A user-defined metric could, e.g., regard a repair set rs preferable to a repair set rs', if the repairs in rs have a higher rating than those in rs', thus:

$$rs \geq_\Sigma rs' \quad \Leftrightarrow \quad \sum_{rep \in rs} \mathsf{rating}_\Sigma(rep) \geq \sum_{rep' \in rs'} \mathsf{rating}_\Sigma(rep')$$

The rating_Σ function assigns to each repair a number depending on its rating. For a repair, rating_Σ counts potentially violated rules (negatively) and subtracts the repair cost. In addition, we punish repairs proposing to delete document content by subtracting 10 and repairs proposing to delete whole documents by subtracting 15. These constants were determined experimentally and will have to be refined as we gain more experience in the future. Usually, the punishment of deletion should be greater than each hint cost. Based on our experiments with software specifications, we consider changing a value preferable to adding a value, which in turn is preferable to deleting a value. In fact, deleting document content or deleting a document itself is a “fall-back” solution that should be considered only if no other repair is possible. For our example repairs from Fig. 9.4 (pg. 132), we calculate the ratings shown in Tab. 9.1. Sorting the overall repair collection w.r.t. the metric $\geq_\Sigma$ results in:

1.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgdoc2}}\}$,
2.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgkeys}}\}$,
3.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{DelkaA2}}\}$,
4.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Deldoc2}}\}$

Thus, authors would probably choose to change the key reference kaA2 to kaA3 in the document doc1.txt and to change the kind of the manual man1.xml to technical M.

Even this simple metric results in noticeable improvements. Other more sophisticated metrics will certainly produce better results for large repair collections (see Sect. 12.4).

[7] A metric does not need to be antisymmetric. Consequently, some repair sets may achieve the same rank in the sorted repair collection.

9.3.2 Partial Inconsistency Resolution

In many situations, we cannot achieve full consistency in a repository. Instead, we only want to resolve the most troubling inconsistencies whose resolutions impose low costs; i.e., we live with less important inconsistencies. For this purpose, we may employ a metric that considers a repair set rs preferable to a repair set rs', if rs contains a repair whose rating is bigger than the ratings of all repairs in rs', thus:

$$rs \geq_{\max} rs' \quad \Leftrightarrow \quad \exists\, rep \in rs \bullet \mathsf{rating}_{\max}(rep) \geq \max_{rep' \in rs'} \mathsf{rating}_{\max}(rep')$$

Now we use a new rating function $\mathsf{rating}_{\max}$, which also considers the inconsistencies resolved by the individual repairs. We calculate inconsistency rankings by (1) mapping priorities to numbers ($\mathsf{high} \rightarrow 5, \mathsf{medium} \rightarrow 3, \mathsf{low} \rightarrow 1$), (2) multiplying each priority with its associated count of inconsistencies, and (3) finally adding up the results.[8] For example, adding up the inconsistencies for $\{2(\mathsf{high}), 1(\mathsf{medium})\}$ results in $2 \cdot 5 + 1 \cdot 3 = 13$. For our example repairs, we calculate ratings also shown in Tab. 9.1. Using the $\geq_{\max}$ metric we obtain the ranking

1.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgdoc2}}\}$,
1.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Chgkeys}}\}$,
1.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{DelkaA2}}\}$,
1.) $\{rep_{\mathsf{Chgman1''}}, rep_{\mathsf{Deldoc2}}\}$

which basically indicates that changing the kind of the manual man1.xml resolves most of the inconsistencies at a relatively small cost.

9.4 Summary

In this chapter, we present an approach to derive repairs for inconsistent repositories. Our primary goal is to propose to authors the most useful repairs. The novel idea is to derive repairs from augmented S-DAGs and not directly from the documents in the repository. The advantage of separating repair derivation from S-DAG generation is that the repository needs to be locked during the computationally cheap S-DAG generation only. By exploiting meta information in augmented S-DAGs about suggested repair actions, we eliminate most repairs that are probably not useful. Our repair derivation algorithm guarantees that (1) each repair set in the repair collection contains compatible repairs only and (2) the repair sets provide mutually independent alternatives. Repairs may introduce new inconsistencies. Therefore, we determine the rules that could be broken by a repair. Finally, we employ ratings and costs of repairs to rank the repair collection by means of user-defined preference metrics. These preference metrics also provide a practical means for partial inconsistency resolution.

[8] Clearly, this only makes sense for partial inconsistency resolution, because a repair set contains repairs that will resolve all inconsistencies.

We consider the generation of prioritized repairs a useful basis for managing consistency in repositories and an essential step towards consistency maintaining DMSs.

In the next part of this thesis, we illustrate by a case study the application of our consistency maintenance approach.

Part III

Case Study: Maintaining Consistency in Industrial Software Specifications

In this part, we apply our consistency maintenance approach to the development of software specifications. Typically, industrial software specifications contain many documents of heterogeneous type and content. The development process of a specification induces temporal consistency requirements. Thus, from our point of view, software specifications provide a useful test bed to validate our consistency maintenance approach.

From the software engineering point of view, developing consistent specifications is one of the most challenging tasks in software development. Often, inconsistencies cause severe costs and delays — they are major reasons for the failure of so many software projects. In particular, informal approaches to specifying software suffer from semantic inconsistencies. Up to now achieving a reasonable level of consistency has required huge manual effort. On the other hand, formal software specifications have as yet not been included into the every-day work of software engineers. Usually, formal approaches require too many changes in the development process and suffer from unjustified complexity. Thus, up to now they do not scale to an industrial setting. Our consistency maintenance approach does not require any adaptations to software engineering processes — it can be set on top of informal software development.

In our case study, we apply consistency maintenance to a software specification method developed by sd&m,[9] a well-established German software company. In Chapter 10, we describe *analysis modules* as a method for developing industrial software specifications and list some consistency requirements. In Chapter 11, we formalize 15 of these requirements. In Chapter 12, we present an example software specification and show how our approach achieves a useful degree of consistency. We present lessons learnt from our case study in Chapter 13.

[9] sd&m: software design & management AG (a company of Capgemini), see www.sdm.de and www.capgemini.com

Chapter 10
Analysis Modules

In this chapter, we describe *analysis modules* developed by sd&m. In addition, we give an overview of consistency requirements between these modules. We shall see that software specification is a quite complex scenario, which benefits from consistency management. In the following, we consider analysis modules and consistency requirements "as is," i.e., we do not discuss them in detail. For a detailed discussion see [SSKK02, SK02].

In this chapter, we proceed as follows: First, we review software specifications from the "document perspective." In Sect. 10.2, we describe analysis modules and list consistency requirements, some of which are formalized in Chapter 11. Finally in Sect. 10.3, we classify consistency requirements technically. Fig. 10.1 illustrates the context of this chapter.

10.1 Industrial Software Specifications are Heterogeneous Document Sets

A well-structured and semantically consistent requirements specification is the base for the success of a software project. Usually, practitioners concentrate on completely and correctly capturing all requirements for the specified system. At sd&m, over 22 years experience in software development confirm that industrial specifications (1) contain heterogeneous results (distributed over many documents in multiple formats), (2) are to large part informal, (3) are guided by a process model, e.g., [JCJO92, BRJ99, JBR99, Bec00, Coc01], and (4) are developed by a team (often using a DMS). Usually, consistency between specification results is achieved by time-consuming manual effort. On the other hand, many theoretical approaches aim at consistent specifications [Wir90, Rat94, RS01, ABK+02]. In practice, however, these approaches are not integrated into the every-day work of software engineers, because they require numerous adaptations to the software development process. Thus, entirely formal approaches currently do not scale to most industrial specifications.

For specifying software, we use analysis modules developed at sd&m [SSKK02, SK02], a document based approach to software specifications [Den93]. A specification consists of many documents that may be written in natural language or may contain (semi-)formal content. Temporal aspects like the

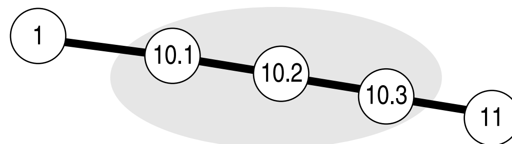


Figure 10.1: Chapter 10 in context

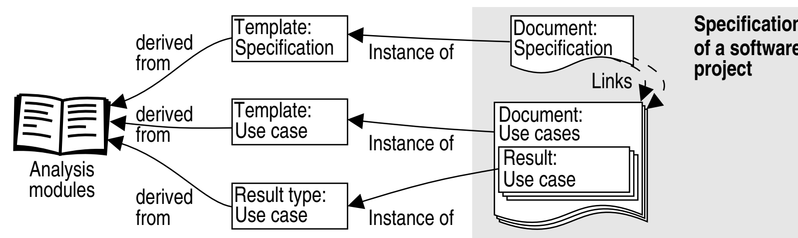


Figure 10.2: Relationships between analysis modules, document templates, result types, specifications, documents, and results

transition from a coarse specification to its fine specification cause additional complexity.

Analysis modules cover a range of properties that can be found in software specifications for large systems:

- A specification contains multiple result types that deal with different aspects of the specified system.
- Usually, for each result type, a specific document type is employed, e.g., UML models, informal graphics, structured or plain text.
- Often, specific tools are used to edit these documents.

Above properties are independent of the specification method itself, like object oriented analysis [Boo94], UML modelling [BRJ99], agile processes [Coc01], or even extreme programming [Bec00]. The number and heterogeneity of the specification results may, however, cause redundancies and inconsistencies. Since inconsistencies severely impact the further development process, huge manual effort is spent on achieving consistency. Our approach to tolerant consistency maintenance automatically pinpoints inconsistencies and proposes to software engineers repairs that resolve inconsistencies. Our idea is simple: We formalize consistency rules for sd&m's analysis modules and check concrete specifications for consistency w.r.t. these rules. We integrate consistency maintenance into the every-day work of software engineers *without* requiring any adaptations to their software engineering practices. This is a major difference to formal software engineering approaches. Therefore, we expect good user acceptance — a key feature entirely formal approaches have not achieved yet. Through applying our consistency management approach, software engineers can concentrate on their actual work: the technical correctness and completeness of a specification.

10.2 Analysis Modules and Consistency Requirements

Fig. 10.2 illustrates the relationships between analysis modules, specifications, and result types. Each analysis module describes a *result type* of a specification. In the specification of a software project, result types are instantiated to

Figure 10.3: Overview of analysis modules by sd&m

results. By the term result we mean the outcome of a requirements analysis activity; by result types we categorize results. For some result types, there may exist multiple results. For example, from the analysis module for use cases we derive the result type "use case;" a software project includes multiple instances (results), i.e., the actual use cases. Usually, specifications for large systems are partitioned into multiple documents, each of which contains results of one result type. The structure of a document follows a template that is derived from the corresponding analysis module. A template contains a header for document metadata (e.g., author and status) and a body that corresponds to a result type. A *software specification* consists of (1) documents containing the results of the specification and (2) a specification document, which references these documents. We shall see that a specification can contain multiple specification documents, in order to distinguish different reader species. All documents are stored in a repository managed by a DMS, in order to coordinate collaborative work of multiple software engineers.

An analysis module acts as a guideline for describing a result, e.g., a use case or a business process. At sd&m, 17 analysis modules have been identified, shown in Fig. 10.3. In the following, we describe the analysis modules and list consistency requirements between them. We have collected the requirements in tight cooperation with sd&m; for later reference, we assign consecutive numbers. Most requirements represent wishes to an "ideal" specification. Clearly, they will be broken during the development process. For each requirement, we list a description in natural language, its strength (weak or strong), its importance (high, medium, or low), its class (see below), whether it affects multiple files, whether it affects multiple versions, and a rough estimate about the complexity of predicates and functions needed for formalization (simple, complex, or very complex). Recall that for the application of our approach it is negligible

whether a requirement affects multiple documents or one document only. The document format is of minor importance, too.

We classify consistency requirements as follows:

- *Referential*: The requirement is essentially a referential integrity constraint: Some document (content) should exist. Referential requirements employ simple functions and predicates only, e.g., equality.
- *Unique*: We require that the name of some content is unique. Usually, these requirements use simple functions and predicates. It can, however, be complicated to collect all content of the same kind, in order to check uniqueness.
- *Linguistic*: The requirement is concerned with linguistic issues, such as readability or semantic similarity of natural-language texts. For formalizing these requirements, the language designer must define complex text analysis functions. Haskell's Foreign Function Interface [C^+03] provides access to libraries useful for this purpose.
- *Logic*: We require logical derivation, e.g., that some facts are implied by other facts, which uses a predicate *implies*. Of course, this predicate is limited to a decidable subset of logic.
- *Naming*: The name of some content should comply with its naming convention. We formalize naming conventions by regular expressions, such that compliance to a naming convention reduces to the matching of text against a regular expression. The language designer can use Haskell's regular expression library for regular expression matching.
- *Calculation*: We require that an equation holds for some content, e.g., that an entity type carries less than seven attributes.

We shall see that most of the requirements below appear quite vague and unclear in natural language. By formalizing requirements as consistency rules we get a much better idea about what consistency really means. In Chapter 11, we formalize 15 requirements, which are written in bold face. We have chosen to formalize at least one requirement from each class above. The formalized consistency rules show interesting properties from the formalization point of view. Also, the rules are useful to examine how specific inconsistencies are shown by S-DAGs. Therefore, the rules provide a good means to evaluate the usefulness of our approach, but also to determine interesting directions for future research.

In the following subsections, we concentrate on the analysis modules needed for our example specification in Chapter 12. For brevity, we omit a description of the modules batch, printed output, overview of the technical architecture, interfaces to up/downstream systems, data migration, cut over, and non-functional requirements.

10.2.1 Essentials

The analysis module 'goals, scope, constraints' includes major goals and the purpose of the specified system. In addition, some global constraints and the most important requirements are identified. In particular, the goals, scope, constraints document defines actors. Once determined, the global conditions of a software project should be invariant over time.

Rule 1 The goals, scope, constraints document, included in the study, should not change significantly afterwards.

strength	importance	class	files	versions	complexity	
weak	high	linguistic	1	multiple	very complex	□

Rule 2 There exists at most one valid goals, scope, constraints document.

strength	importance	class	files	versions	complexity	
strong	medium	referential	multiple	1	simple	□

Rule 3 The name of an actor is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	□

10.2.2 Non-Functional Properties

Non-functional properties, such as performance aspects, naming conventions, or style guides, apply to all result types of a specification. Non-functional requirements list simple properties. For more complex global requirements, cross-cutting concerns are used. For example, cross-cutting concerns include a naming convention for each result type of the specification and a style guide containing layout descriptions for the user interface. For large specifications, with many similar dialogs, the style guide contains dialog types.

Rule 4 The cross-cutting concerns of the fine specification should contain a style guide.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	□

Rule 5 The cross-cutting concerns should contain a naming convention for each result of a specification.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 6 The style guide should not change significantly over time.

strength	importance	class	files	versions	complexity	
weak	high	linguistic	1	multiple	very complex	□

Rule 7 There exists at most one cross-cutting concerns document per topic.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 8 The name of a dialog type is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	naming, unique	multiple	1	complex	□

10.2.3 Functional Requirements

Often, functional requirements are considered the heart of a specification. The analysis modules include business processes, use cases, and analysis functions.

A business process describes central activities in the customer business. Therefore, business processes are necessary for a basic understanding of the new system and the communication between software engineers and customers. The specified system supports some activities of a business process. At sd&m, business processes are categorized by organizational units. In addition, all business processes are listed in a business process overview. Each activity of a business process is associated with an actor and may reference in- and output data, respectively. Activities supported by the system are described by use cases in detail.

Rule 9 The name of a business process is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	naming, unique	multiple	1	complex	☐

Rule 10 The name of an organizational unit is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	naming, unique	multiple	1	complex	☐

Rule 11 Each business process listed in the business process overview should be defined.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	☐

Rule 12 Each business process should exist in the overview.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	☐

Rule 13 An activity within a business process is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	naming, unique	multiple	1	complex	☐

Rule 14 Each activity is associated with an actor.

strength	importance	class	files	versions	complexity	
weak	medium	referential	1	1	simple	☐

Rule 15 It should be clear which activities are supported by which computer systems (new / adjacent / none).

strength	importance	class	files	versions	complexity	
weak	medium	referential	1	1	simple	☐

Rule 16 Data and documents needed for or produced by an activity should be marked.

strength	importance	class	files	versions	complexity	
weak	medium	referential	1	1	simple	☐

Rule 17 A business process should contain a few branches only.

strength	importance	class	files	versions	complexity	
weak	low	calculation	1	1	simple	☐

A use case describes the functional requirements of the specified system from the user perspective. From a business process, a use case can be found as part of an activity, one activity as a whole, or several activities together. For a use case, software engineers may state pre- and postconditions; the preconditions of a use case should be implied by the postconditions of all other use cases (these use cases should be performed before). For our purposes, we formalize pre- and postconditions by boolean logic formulae.

Rule 18 For each activity supported by the system, there should exist (one or more) use cases.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 19 The name of a use case is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	naming, unique	multiple	1	complex	□

Rule 20 A use case should be associated with an activity in a business process.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 21 The preconditions of a use case should be satisfiable from the postconditions of all other use cases.

strength	importance	class	files	versions	complexity	
weak	medium	logic	multiple	1	complex	□

Rule 22 Use cases referenced by another use case should exist.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 23 A use case should not both use and extend another use case.

strength	importance	class	files	versions	complexity	
weak	medium	referential	1	1	simple	□

Rule 24 There should be no cycles in the uses-relations between use cases.[1]

strength	importance	class	files	versions	complexity	
weak	medium	calculation	multiple	1	complex	□

Rule 25 Extends-relations between use cases should be used rarely.

strength	importance	class	files	versions	complexity	
weak	medium	calculation	1	1	simple	□

Rule 26 Entity types needed by a use case should be defined in the data model.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

[1] Rule 24 requires to compute the transitive closure of the uses-relation.

Complex processes and computations without user interaction may be described by analysis functions, which are referred to by use cases. For example, analysis functions are used to describe checking procedures for the credit assessment of a customer. By describing such details through analysis functions, use cases remain small and comprehensible. The description of an analysis function contains pre- and postconditions, effects, and messages. The pre- and postconditions of an analysis function should comply with the pre- and postconditions of its using use cases. In the fine specification, analysis functions may reference entity types in the data model or data types in the data types document.

Rule 27 The name of an analysis function is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity
weak	high	unique, naming	multiple	1	complex

□

Rule 28 An analysis function used by a use case should exist.

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	1	simple

□

Rule 29 The preconditions of an analysis function should be implied by the preconditions of each referencing use case.

strength	importance	class	files	versions	complexity
weak	medium	logic	multiple	1	complex

□

Rule 30 The conjunction of the postconditions of all analysis functions referenced by a use case u should imply the postconditions of u.

strength	importance	class	files	versions	complexity
weak	medium	logic	multiple	1	complex

□

Rule 31 Analysis functions used by another analysis function should exist.

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	1	simple

□

Rule 32 The preconditions of an analysis function should be implied by the preconditions of each using analysis function.

strength	importance	class	files	versions	complexity
weak	medium	logic	multiple	1	complex

□

Rule 33 The conjunction of the postconditions of all analysis functions used by an analysis function f should imply the postconditions of f.

strength	importance	class	files	versions	complexity
weak	medium	logic	multiple	1	complex

□

Rule 34 Entity types (data types) used by analysis functions should be defined in the data model (data types document).

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	1	simple

□

10.2.4 Data

The data model defines business data; i.e., it provides a technical view on the static persistent data of the system. Usually, entity relationship diagrams (in UML notation) are used for this purpose. Attributes of entity types reference data types from the data types document. Depending on the type system used (e.g., if polymorphic data types are supported), these data types may not be found directly; they can be derived only. Entity types within the data model may be referenced by use cases, analysis functions, or dialogs. The consistency requirements below also cover demands to a "useful" data model.

Rule 35 The name of an entity type is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	☐

Rule 36 The name of an attribute should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	naming	multiple	1	complex	☐

Rule 37 Within an entity type, the names of its attributes should be unique.

strength	importance	class	files	versions	complexity	
weak	high	unique	multiple	1	simple	☐

Rule 38 The name of an attribute is not the name of an entity type.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	☐

Rule 39 An attribute has an associated type, which should be derivable from the data types.

strength	importance	class	files	versions	complexity	
weak	high	logic	multiple	1	complex	☐

Rule 40 Entity types referenced by a relation should be defined in the data model.

strength	importance	class	files	versions	complexity	
weak	high	referential	1	1	simple	☐

Rule 41 In the finished fine specification, each relation should have cardinalities associated with its entity types.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	☐

Rule 42 The UML notation must use entity types, attributes, and associations only (no methods).

strength	importance	class	files	versions	complexity	
strong	medium	referential	1	1	simple	☐

Rule 43 Each entity type in the data model should be used by a use case, an analysis function, or a dialog.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	☐

Rule 44 A key attribute for an entity type should be visible, i.e., there exists a dialog showing this attribute.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	□

Rule 45 There exists at most one valid data model.

strength	importance	class	files	versions	complexity	
strong	high	referential	1	1	simple	□

Rule 46 An entity type should have less than seven attributes.

strength	importance	class	files	versions	complexity	
weak	low	calculation	1	1	simple	□

The data types document defines technical data types. These types may be used for attribute types. We distinguish between elementary types, structure types (records), enumeration types (variants), and range types.

Rule 47 The name of a data type is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	□

Rule 48 The lower bound of a range data type should be smaller than its upper bound.

strength	importance	class	files	versions	complexity	
weak	medium	calculation	1	1	simple	□

Rule 49 Field names of a structure type are unique and should comply with their naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	medium	unique, naming	multiple	1	complex	□

Rule 50 Field types of a structure type should be derivable from the data types.

strength	importance	class	files	versions	complexity	
weak	high	logic	1	1	complex	□

Rule 51 The types of enumeration values should be derivable from the data types.

strength	importance	class	files	versions	complexity	
weak	high	logic	1	1	complex	□

Rule 52 There exists at most one data types document.

strength	importance	class	files	versions	complexity	
strong	high	referential	1	1	simple	□

Rule 53 Values and fields referenced by a data type format should exist in the same data type definition.

strength	importance	class	files	versions	complexity	
weak	high	referential	1	1	simple	□

Rule 54 The type of a range should be either elementary or enumerable.

strength	importance	class	files	versions	complexity	
weak	high	referential	1	1	simple	□

10.2.5 User Interface

Users can interact with the specified system in different ways. The analysis module 'dialog' specifies the look and feel of dialogs (by dialog screens) as well as their functionality (by interactivity diagrams [Den91]). For a dialog, software engineers define data (entity types in the data model) presented to or modified by the user. For large systems, containing many similar dialogs, the style guide in the cross-cutting concerns should define dialog types. Dialogs are associated to use cases. For complex computations, dialogs may reference analysis functions.

Rule 55 The name of a dialog is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	□

Rule 56 The name of a screen is unique within its dialog and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	□

Rule 57 A dialog type referenced by a dialog should exist in the style guide of the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	very complex	□

Rule 58 A persistent input/output field in a dialog screen should be associated with a valid attribute from the data model.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	□

Rule 59 The data type of an attribute shown by a persistent input/output field should be associated with a format in the data types.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 60 A dialog screen referenced by a state in an interactivity diagram (IAD) should be defined in the same dialog.

strength	importance	class	files	versions	complexity	
weak	high	referential	1	1	simple	□

Rule 61 Each dialog screen should occur in the dialog's IAD.

strength	importance	class	files	versions	complexity	
weak	high	referential	1	1	simple	□

Rule 62 Dialogs referenced by a dialog should be defined.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 63 Analysis functions used by a dialog should be defined in an analysis functions document.

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

Rule 64 **Each analysis function used by a dialog should also be used by each use case associated with this dialog.**

strength	importance	class	files	versions	complexity	
weak	high	referential	multiple	1	simple	□

10.2.6 Miscellaneous

Besides the modules above, a specification should contain reading instructions and a glossary. Reading instructions describe the structure of the specification and the specification method, e.g., object oriented analysis. For each analysis module, used in the specification, the reading instructions determine which reader species should read instances of this module. We distinguish between users, software designers, the test team, and the project management. For example, analysis functions should be read by the test team; they do not need to be read by the user.

Rule 65 The name of a reader species is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	☐

Rule 66 The reading instructions should address the following reader species: user, software designer, test team, and project management.

strength	importance	class	files	versions	complexity	
weak	medium	referential	1	1	simple	☐

Rule 67 Each analysis module described by the reading instructions should be part of the specification.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	☐

Rule 68 There must exist at most one valid reading instructions document.

strength	importance	class	files	versions	complexity	
strong	medium	referential	1	1	simple	☐

The glossary explains domain-specific vocabulary and technical terms. Once agreed on, the definition of a term should be invariant over time.

Rule 69 There must exist at most one valid glossary.

strength	importance	class	files	versions	complexity	
strong	medium	referential	1	1	simple	☐

Rule 70 The definition of a term in the glossary does not change significantly over time.

strength	importance	class	files	versions	complexity	
weak	high	linguistic	1	multiple	very complex	☐

Rule 71 The glossary should define a term only once.

strength	importance	class	files	versions	complexity	
weak	high	unique	1	1	simple	☐

10.2.7 Components

Large software systems are partitioned into technical components, which provide a view orthogonal to the analysis modules. At sd&m, components are applied to the analysis modules use cases, analysis functions, data model, dialogs, batch processes, and printed outputs. Components appear as sections in documents. Useful components fulfill the following requirements.

Rule 72 The name of a component is unique and should comply with its naming convention in the cross-cutting concerns.

strength	importance	class	files	versions	complexity	
weak	high	unique, naming	multiple	1	complex	□

Rule 73 A use case should use analysis functions from its own component only.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	□

Rule 74 In the data model, an entity type has more relations to entity types within the same component than to entity types within a different component.

strength	importance	class	files	versions	complexity	
weak	medium	calculation	multiple	1	simple	□

10.2.8 Specification Documents

A specification document references documents containing the results of a specification. Every reader species from the reading instructions needs a specific specification document. For a specification, we distinguish between study, coarse specification, and fine specification. Transitions between these phases call for temporal consistency requirements. For example, business processes agreed on in the coarse specification should be refined in the fine specification. Thus, we require that activities from the coarse specification are part of the fine specification. On the other hand, all business processes in the fine specification should be known in the coarse specification already.

Rule 75 The study should contain goals, scope, constraints and a glossary.

strength	importance	class	files	versions	complexity	
weak	medium	referential	multiple	1	simple	□

Rule 76 Each document referenced by a specification document exists.

strength	importance	class	files	versions	complexity	
strong	high	referential	multiple	1	simple	□

Rule 77 Each document referenced by a specification document should pass linguistic analysis, e.g., the check for whole sentences, passive constructions, and the avoidance of 'unwanted' words.

strength	importance	class	files	versions	complexity	
weak	medium	linguistic	multiple	1	very complex	□

Rule 78 In a specification, each document delivered to the customer should be easy to read.

strength	importance	class	files	versions	complexity	
weak	high	linguistic	multiple	1	very complex	□

Rule 79 Business processes of the coarse specification should not change significantly in the fine specification.

strength	importance	class	files	versions	complexity
weak	medium	linguistic	multiple	multiple	very complex

□

Rule 80 A system-supported activity in a business process of the coarse specification is also included in the fine specification and is also supported by the system.

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	multiple	simple

□

Rule 81 The fine specification should not introduce new business processes.

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	multiple	simple

□

Rule 82 A specification intended for a species of readers only contains documents intended for this reader species.

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	1	simple

□

10.2.9 Document Life Cycle

Each document follows a simple life cycle, which distinguishes the status values 'in progress,' 'quality assurance,' and 'finished.' When the work on a document in status 'in progress' has ended it gets the status 'quality assurance,' which in turn becomes 'finished' after the document has passed quality control successfully. We model restrictions of the sd&m document life cycle by the following consistency requirements.

Rule 83 If a document in finished status is modified, it must get the status 'in progress.'

strength	importance	class	files	versions	complexity
strong	medium	referential	1	2	simple

□

Rule 84 A document in status 'in progress' should change its status to 'quality assurance' only.

strength	importance	class	files	versions	complexity
weak	medium	referential	1	2	simple

□

Rule 85 The coarse specification includes documents in status 'finished' only.

strength	importance	class	files	versions	complexity
weak	medium	referential	multiple	1	simple

□

Rule 86 The fine specification contains all analysis modules of the finished coarse specification; the modules are in finished status.

strength	importance	class	files	versions	complexity
weak	high	referential	multiple	multiple	simple

□

10.2.10 Requirements not Formalizable by Our Approach

In this section, we list some requirements that are, as yet, not formalizable by our approach. Mostly, this is because these requirements call for precise semantic analysis of natural language or pictures. If, however, such means were provided, e.g., by sophisticated metadata extraction approaches [WS99, MPG01, FGLM02], the below requirements become formalizable.

- Business processes should not describe technical requirements.
- An activity that is already completely modelled by a use case or a set of use cases should not be modelled by another use case.
- Graphical and textual descriptions of business processes, use cases, and the data model, respectively, should be consistent.
- The graphical layout of a dialog conforms to the layout specified in the dialog type.
- A use case should not describe the graphical user interface.
- A specification should be neither too detailed nor too coarse.
- The style of a specification should not be too schematic.

10.3 Summary

Tab. 10.1 summarizes consistency requirements from the previous section. Most requirements are referential. Some of these referential requirements can be expressed by the document model (e.g., by XML Schema constraints) or checked by a link checker. Most referential requirements rely, however, on additional information that link checkers are not aware of. Notice that link checkers and restrictions by the document model implement a *strict* view of consistency, which would severely hinder the development of a specification. Another large group of requirements is formed by naming and uniqueness requirements. Most uniqueness requirements are also naming requirements. The large number of requirements in these categories is justified by sd&m's naming policy: Naming conventions are considered useful for industrial software specifications.

Only a few requirements are strong. At sd&m, software engineers expect most requirements to be violated by a specification. This supports our thesis that full consistency w.r.t. user-defined consistency requirements cannot be achieved in a real life environment. Instead, it is necessary to tolerate inconsistencies.

Strength	Impor-tance	Referential	Lin-guistic	Logic	Naming	Unique	Calc-ulation
weak	high	5 7 11 12 18 20 22 26 28 31 34 38 40 53 54 *57* 59 60 61 62 63 **64** **80** 81 **82** 86	*1 6* ***70*** *78*	*39 50 51*	***3*** *8 9 10 13* ***19*** *27 35 36 47 49 55 56 65 72*	**3** 8 9 10 13 **19** 27 35 37 47 49 55 56 65 71 72	
	medium	**4** 14 15 16 23 **41** 43 44 58 66 67 **73** 75 **84** 85	*77 79*	***21*** *29* ***30*** *32 33*			*24* 25 **48** **74**
	low						17 46
strong	high	45 52 69 76					
	medium	**2** 42 68 69 83					
	low						

Table 10.1: Summary of consistency requirements for analysis modules (requirements in italic face include complex or very complex predicates; requirements in bold face are formalized in the next chapter)

Chapter 11
Formalizing Consistency Rules

In this chapter, we formalize 15 of the consistency requirements found in Chapter 10. We will see that the most laborious work lies in the definition of document types. We need to define only a few additional function symbols and predicate symbols, because the basic language Prelude already contains many useful symbols. Since we use XML as document format, the implementations of parser functions can be derived by the Haskell XML parser HaXml [WR99]. In Sect. 11.1, we define the domain-specific language SDM. In Sect. 11.2, we formalize consistency requirements. Fig. 11.1 illustrates the context of this chapter.

11.1 The Language Designer's Work: Developing the Language SDM

In this section, we define the domain-specific language SDM. For brevity, we only present those parts of SDM that are necessary to formalize our example rules. The following sections contain definitions of types, predicate symbols, and function symbols. Many type definitions are given partially; we mark omissions by three dots (...). Most of the types and function symbols are needed to process sd&m specific documents. Sect. 11.1.4 contains an excerpt from the basic language Prelude. For a full documentation of types, predicate symbols, and function symbols see the project WWW site

www2-data.informatik.unibw-muenchen.de/cde.html

11.1.1 Type Definitions

An sd&m specific document carries a validity attribute and a status. The type `DocSDM` is supertype of all sd&m specific documents. Recall that record subtypes inherit all labels from their supertypes; thus `DocSDM` also includes the labels `dId` and `dState`.[1] A specification document contains its kind and the

[1]For brevity, we omit record constructors — they are considered equal to their corresponding record type constructors. E.g., the definition of `DocSDM` is actually:
$\texttt{DocSDM} < \{\texttt{Doc}\} \quad = \quad \texttt{DocSDM}\ \{\texttt{valid} : \texttt{Bool}, \texttt{status} : \texttt{Status}, \ldots\}.$

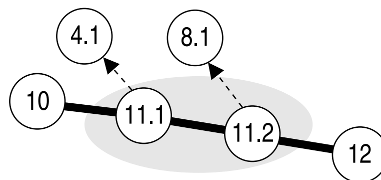


Figure 11.1: Chapter 11 in context

reader species expected to read this specification. In addition, the type `DocSpec` includes labels for references to documents that are part of a specification. We define the type `Result` as a common supertype of all specification results; it carries a name and a plain text description.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{sd\&m documents}} \\
\mathtt{DocSDM} < \{\mathtt{Doc}\} & = & \{\mathtt{valid} : \mathtt{Bool}, \mathtt{status} : \mathtt{Status}, \ldots\} \\
\multicolumn{3}{l}{\text{Document status values}} \\
\mathtt{Status} & = & \mathtt{InProgress} \mid \mathtt{QualAssurance} \mid \mathtt{Finished} \\
\multicolumn{3}{l}{\text{Specification documents}} \\
\mathtt{DocSpec} < \{\mathtt{DocSDM}\} & = & \left\{\begin{array}{l}\mathtt{specKind} : \mathtt{SpecKind}, \mathtt{readers} : \mathtt{String}, \\ \mathtt{doc_crosscut} : [\mathtt{String}], \mathtt{doc_reading} : \mathtt{String}, \\ \mathtt{doc_busproc} : [\mathtt{String}], \mathtt{doc_datamod} : \mathtt{Maybe\ String}, \ldots \end{array}\right\} \\
\multicolumn{3}{l}{\text{Specification kinds}} \\
\mathtt{SpecKind} & = & \mathtt{Study} \mid \mathtt{Coarse} \mid \mathtt{Fine} \\
\multicolumn{3}{l}{\text{Supertype of all result types}} \\
\mathtt{Result} & = & \{\mathtt{name} : \mathtt{String}, \mathtt{desc} : \mathtt{String}\}
\end{array}
$$

A goals, scope, constraints document contains names and descriptions for actors.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{Goals, scope, constraints documents}} \\
\mathtt{DocGoalsScope} < \{\mathtt{DocSDM}\} & = & \mathtt{existingactors} : [\mathtt{Result}], \ldots
\end{array}
$$

A cross-cutting concerns document contains naming conventions and a style guide. The type `NamingConv` carries an extra label for the naming convention of each specification result. A naming convention is expressed by a `String` denoting a regular expression. The style guide contains types for screens (in a dialog) and buttons (in a screen).

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{Cross-cutting concerns documents}} \\
\mathtt{DocCrossCut} < \{\mathtt{DocSDM}\} & = & \mathtt{conventions} : \mathtt{NamingConv}, \mathtt{styleguide} : \mathtt{Styleguide}, \ldots \\
\multicolumn{3}{l}{\text{Naming conventions}} \\
\mathtt{NamingConv} & = & \mathtt{nc_actor} : \mathtt{Maybe\ String}, \mathtt{nc_uc} : \mathtt{Maybe\ String}, \ldots \\
\multicolumn{3}{l}{\text{Style guides}} \\
\mathtt{Styleguide} & = & \mathtt{screenTs} : [\mathtt{SGType}], \mathtt{buttonTs} : [\mathtt{SGType}], \ldots \\
\multicolumn{3}{l}{\text{Style guide types}} \\
\mathtt{SGType} < \{\mathtt{Result}\} & = & \{\mathtt{ref_layout} : \mathtt{String}\}
\end{array}
$$

A business process document contains a list of business processes. For a business process, we restrict ourselves to labels needed to model activities. For an activity, we model its input and output data only. A reference to an ingredient of a business process (e.g., an activity) carries the business process name and the name of the ingredient. In order to define a process, an ingredient carries pointers to next and previous ingredients, respectively. These pointers may consist of boolean logic expressions. The parametric type `Logic` α is also used for pre- and postconditions of analysis functions and use cases, respectively.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{Business process documents}} \\
\mathtt{DocBusProc} < \{\mathtt{DocSDM}\} & = & \{\mathtt{processes} : [\mathtt{BusProc}]\} \\
\multicolumn{3}{l}{\text{Supertypes of results having an actor}} \\
\mathtt{Actor} & = & \{\mathtt{actors} : [\mathtt{String}]\} \\
\multicolumn{3}{l}{\text{Business processes}} \\
\mathtt{BusProc} < \{\mathtt{Result}, \mathtt{Actor}\} & = & \mathtt{activities} : [\mathtt{Activity}], \mathtt{usesBP} : [\mathtt{RefBP}], \ldots
\end{array}
$$

References to business processes
$\texttt{RefBP} = \{\texttt{bpName} : \texttt{String}, \texttt{bpInt} : \texttt{RefBPInt}\}$
References to business process ingredients
$\texttt{RefBPInt} = \texttt{RefActivity String} \mid \texttt{RefProc RefBP} \mid \ldots$
Ingredients of business processes
$\texttt{BPInt} < \{\texttt{Result}\} = \texttt{prevI} : \texttt{Maybe (Logic RefBPInt)}, \texttt{nextI} : \texttt{Maybe (Logic RefBPInt)}$
Boolean logic expressions
$\texttt{Logic}\ \alpha = \texttt{LTrue} \mid \texttt{Con}\ \alpha \mid \texttt{Not (Logic}\ \alpha\texttt{)} \mid \texttt{And [Logic}\ \alpha\texttt{]} \mid \texttt{Or [Logic}\ \alpha\texttt{]}$
Activities
$\texttt{Activity} < \{\texttt{BPInt}, \texttt{Actor}\} = \texttt{input} : [\texttt{Data}], \texttt{output} : [\texttt{Data}], \ldots$
Data references
$\texttt{Data} = \texttt{Document String} \mid \texttt{Data String} \mid \ldots$

A use case document is subdivided by components, which contain use cases or other subcomponents. Since a component is a recursive data structure, we need to flatten it, in order to iterate over all elements. For this purpose, we define the function `flattenComp`, see Sect. 11.1.3. A use case contains references to analysis functions and to its associated activities in business processes. The pre- and postconditions of a use case are boolean logic expressions over atomic conditions, denoted by `String`s.

Use case documents
$\texttt{DocUsecase} < \{\texttt{DocSDM}\} = \{\texttt{ucComp} : [\texttt{Comp Usecase}]\}$
Components
$\texttt{Comp}\ \alpha < \{\texttt{Result}\} = \{\texttt{subcomps} : [\texttt{Comp}\ \alpha], \texttt{elements} : [\alpha]\}$
Use cases
$\texttt{Usecase} < \{\texttt{Result}, \texttt{Prepost}, \texttt{Actor}\}$
$= \texttt{for_activity} : [\texttt{RefBP}], \texttt{ref_fun} : [\texttt{String}], \ldots$
Pre- and postconditions
$\texttt{PrePost} = \texttt{precons} : \texttt{Logic String}, \texttt{postcons} : \texttt{Logic String}$

Documents for analysis functions follow the shape of use case documents. An analysis function carries pre- and postconditions, its parameters, and its result. Parameters may be given by plain text descriptions or by references to entity types.

Analysis function documents
$\texttt{DocAnaFun} < \{\texttt{DocSDM}\} = \{\texttt{afComp} : [\texttt{Comp AnaFun}]\}$
Analysis functions
$\texttt{AnaFun} < \{\texttt{Result}, \texttt{PrePost}\} = \texttt{params} : [\texttt{Param}], \texttt{result} : \texttt{Param}, \ldots$
Parameters of analysis functions
$\texttt{Param} = \texttt{ParamDesc String} \mid \texttt{ParamRef RefET} \mid \ldots$
References to entity types
$\texttt{RefET} = \{\texttt{etName} : \texttt{String}, \texttt{etAttrs} : [\texttt{String}]\}$

Components of a data model contain entity type definitions. An entity type may have relations with other entity types (for brevity, we omit a record label for attributes).

Data model documents
$\texttt{DocDataMod} < \{\texttt{DocSDM}\} = \{\texttt{compsDM} : [\texttt{Comp ET}]\}$
Entity types
$\texttt{ET} < \{\texttt{Result}\} = \texttt{relatesWith} : [\texttt{RelatesWith}], \ldots$
Relations with other entity types
$\texttt{RelatesWith} < \{\texttt{Result}\} = \texttt{targetcard} : \texttt{String}, \texttt{ref_entitytype} : \texttt{RefET}, \ldots$

A technical data type may be a basic type, a range type, or a structure type. Range types also carry explicit information about the type of the range.

Data types documents
$\mathtt{DocDataTs} < \{\mathtt{DocSDM}\} \;=\; \{\mathtt{dataTs} : [\mathtt{DataType}]\}$
Technical data types
$\mathtt{DataType} < \{\mathtt{Result}\} \;=\; \{\mathtt{kind} : \mathtt{DTKind}, \ldots\}$
Kinds of technical data types
$\mathtt{DTKind} \;=\; \mathtt{Basic} \mid \mathtt{Range\ Range} \mid \mathtt{Struct}\ [\mathtt{Field}] \mid \ldots$
Range data types
$\mathtt{Range} \;=\; \mathtt{rangeType} : \mathtt{String}, \mathtt{rangeFrom} : \mathtt{String}, \mathtt{rangeTo} : \mathtt{String}$
Fields of structure data types
$\mathtt{Field} < \{\mathtt{Result}\} \;=\; \{\mathtt{fieldType} : \mathtt{String}, \ldots\}$

Dialog documents are subdivided by components, too. A dialog consists of screens, actions, and an interactivity diagram. For brevity, we only concentrate on actions, which carry references to analysis functions.

Dialog documents
$\mathtt{DocDialog} < \{\mathtt{DocSDM}\} \;=\; \{\mathtt{diaComp} : [\mathtt{Comp\ Dialog}]\}$
Dialogs
$\mathtt{Dialog} < \{\mathtt{Result}\} \;=\; \mathtt{usecases} : [\mathtt{String}], \mathtt{actions} : [\mathtt{Action}], \ldots$
Actions
$\mathtt{Action} < \{\mathtt{Result}\} \;=\; \{\mathtt{actFun} : [\mathtt{String}], \ldots\}$

A reading instructions document contains definitions of existing reader species and for each analysis module a reading instruction (for brevity, we concentrate on reading instructions for analysis functions and cross-cutting concerns only). A reading instruction specifies which reader species should read which part of an analysis module.

Reading instructions documents
$\mathtt{DocReading} < \{\mathtt{DocSDM}\}$
$\quad = \; \mathtt{existingreaders} : [\mathtt{Result}], \mathtt{ri_anafun} : \mathtt{Maybe}\ [\mathtt{RI}], \mathtt{ri_crosscut} : \mathtt{Maybe}\ [\mathtt{RI}], \ldots$
Reading instructions
$\mathtt{RI} \;=\; \{\mathtt{shouldRead} : [\mathtt{String}], \mathtt{whichPart} : \mathtt{String}\}$

A glossary contains term definitions, which may reference other terms.

Glossary documents
$\mathtt{DocGlossary} < \{\mathtt{DocSDM}\} \;=\; \{\mathtt{terms} : [\mathtt{Term}]\}$
Glossary terms
$\mathtt{Term} < \{\mathtt{Result}\} \;=\; \{\mathtt{alternatives} : [\mathtt{String}], \mathtt{seealso} : [\mathtt{String}]\}$

11.1.2 Predicate Symbol Definitions

We need to define new predicate symbols that are not already included in the language Prelude. $\equiv$ determines whether two `String`s are similar; we regard two `String`s as similar, if the first `String` is a substring of the second `String`, where case is neglected. We match a `String` against a regular expression by `match`, which uses Haskell's regular expression library. The symbol $|\!\Rightarrow$ is used to determine logical implication between boolean logic expressions.

$\equiv$	:	String×String → Bool	Similarity between Strings
match	:	String×String → Bool	Regular expression matching, third argument is a regular expression
$\models\!\Rightarrow$	:	Logic α×Logic α → Bool	Logical implication, first formula implies second formula

11.1.3 Function Symbol Definitions

First, we define some auxiliary functions for convenience. The constant `todo` returns the `String` "TODO" and serves as a marker. We flatten recursive components by `flattenComp`; each returned component does not contain any subcomponents. The total functions `rngFrom` and `rngTo` return the lower bound and upper bound of a data type, respectively. Applied to a non-range data type, `rngFrom` returns 0 and `rngTo` returns 1. The function symbol `addRI` adds a reading instruction for a specific reader species to a list of reading instructions.

todo	:	String	String constant "TODO"
flattenComp	:	Comp α → [Comp α]	Flatten component
rngFrom	:	DTKind → Float	Lower bound of a range data type
rngTo	:	DTKind → Float	Upper bound of a range data type
addRI	:	Maybe [RI]×String → Maybe [RI]	Add a reading instruction for reader species given as third argument

We also need functions that access documents in the repository and parse their content to a data structure from the language SDM. The implementations of the repository access functions below are derived from the DTDs of the analysis modules. The final column below denotes the documents accessed.

repDocSDM	:	State → [DocSDM]	sd&m documents	{*.xml}
repSP	:	State → [DocSpec]	Specifications	{Spec*.xml}
repGoalsScope	:	State → [DocGoalsScope]	Goals, scope, constraints	{Goals*.xml}
repCrosscut	:	State → [DocCrosscut]	Cross-cutting concerns	{Cross*.xml}
repBusProc	:	State → [DocBusProc]	Business processes	{BP*.xml}
repUsecase	:	State → [DocUsecase]	Use cases	{UC*.xml}
repAnaFun	:	State → [DocAnaFun]	Analysis functions	{AnaFun*.xml}
repDataMod	:	State → [DocDataMod]	Data models	{DataM*.xml}
repDataTs	:	State → [DocDataTs]	Data types	{DataT*.xml}
repDialog	:	State → [DocDialog]	Dialogs	{Dialog*.xml}
repReadingInst	:	State → [DocReading]	Reading instructions	{Reading*.xml}
repGlossary	:	State → [DocGlossary]	Glossaries	{Gloss*.xml}

11.1.4 Relevant Excerpt from the Language Prelude

For brevity, we omit a detailed description of the basic language Prelude. Below, we only define types and symbols needed for our case study. Their meaning should be known already from the preceding chapters. Most implementations carry over from the Haskell Prelude [PJ03]. Noticeable exceptions are the functions `filter`$\in$ and `filter`$\notin$: `filter`$\in$ (xs, f, ys) returns all elements y from the list ys for which $f(y) \in xs$ holds; `filter`$\notin$ (xs, f, ys) returns those $y \in ys$ for which $f(y) \in xs$ fails. Prelude includes these functions for convenience, because we prohibit partial application. The predicate symbols $=$, $\neq$, $<$, $\leq$, and $\in$ are fully polymorphic, which is safe since in our syntax we permit first-order type

variables only. Recall that first-order type variables α do not match function types (we use canonical equality and ordering relations, which are derived automatically for all types). In contrast to `prev`, the function symbol `prevState` is total, i.e., `prevState`$(t) = t$ if $t =$ `repInit`.

`Doc`	=	`dId : String, dState : State`	Documents
`Bool`	=	`True` | `False`	Booleans
$[\alpha]$	=	$[] \mid (:)\ \alpha \times [\alpha]$	Haskell like lists
`Maybe` α	=	`Nothing` | `Just` α	Haskell like `Maybe`
`Int`			Integer numbers
`Float`			Floating point numbers
`String`			Strings

$=$	:	$\alpha \times \alpha \rightarrow \mathsf{Bool}$	Equality
$\neq$	:	$\alpha \times \alpha \rightarrow \mathsf{Bool}$	In-equality
$<$	:	$\alpha \times \alpha \rightarrow \mathsf{Bool}$	Less than
$\leq$	:	$\alpha \times \alpha \rightarrow \mathsf{Bool}$	Less than or equal
$\in$	:	$\alpha \times [\alpha] \rightarrow \mathsf{Bool}$	Element of relation
`null`	:	$[\alpha] \rightarrow \mathsf{Bool}$	Is a list empty?

$1, 2, \ldots$	:	`Int`	Integer constants
_	:	`String`	Empty string
$/_{\mathtt{Int}}$	:	`Int`$\times$`Int` $\rightarrow$ `Int`	Integer division
$+\!\!+$	:	`String`$\times$`String` $\rightarrow$ `String`	String concatenation
`id`	:	$\alpha \rightarrow \alpha$	Identity function
`length`	:	$[\alpha] \rightarrow$ `Int`	Length of a list
$\backslash$	:	$[\alpha] \times [\alpha] \rightarrow [\alpha]$	Set like difference for lists
`catMaybes`	:	[`Maybe` α] $\rightarrow [\alpha]$	Haskell like `catMaybes`
`map`	:	$(\alpha \rightarrow \beta) \times [\alpha] \rightarrow [\beta]$	Haskell like `map`
`concatMap`	:	$(\alpha \rightarrow [\beta]) \times [\alpha] \rightarrow [\beta]$	Haskell like `concatMap`
`filter`	:	$(\alpha \rightarrow$ `Bool`$) \times [\alpha] \rightarrow [\alpha]$	Haskell like `filter`
`filter`$\in$	:	$[\beta] \times (\alpha \rightarrow \beta) \times [\alpha] \rightarrow [\alpha]$	Advanced `filter`
`filter`$\notin$	:	$[\beta] \times (\alpha \rightarrow \beta) \times [\alpha] \rightarrow [\alpha]$	Advanced negative `filter`
`repStates`	:	$[\mathsf{State}]$	All repository states
`repInit`	:	State	Initial repository state
`next`	:	$\mathsf{State} \rightarrow \mathsf{State}$	Next repository state
`prev`	:	$\mathsf{State} \rightarrow \mathsf{State}$	Previous repository state
`prevState`	:	$\mathsf{State} \rightarrow \mathsf{State}$	Previous repository state (total)

11.2 The Rule Designer's Work: Formalizing Consistency Rules

In this section, we formalize 15 consistency requirements from Sect. 10.2. Except for Rule 84 each of the following rules is applicable to rule filtering (as described in Sect. 6.2.1).

Rule 2: There exists at most one valid goals, scope, constraints document. We first quantify over all repository states t. For all valid goals, scope, constraints documents gs and gs_1, we require that they have the same name. If there exist two or more valid goals, scope, constraints documents, we propose to invalidate the document gs_1. The hint collection $\{\emptyset\}$ avoids generation of any repairs for the atomic formula `dId`$(gs) =$ `dId`(gs_1). Our system associates the documents {`Goals*.xml`}.

$$\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, gs \in \texttt{repGoalsscope}(t) \bullet$$
$$\texttt{valid}(gs) = \texttt{True} \Rightarrow \begin{pmatrix} \forall\, gs_1 \in \texttt{repGoalsscope}(t) \bullet \\ \texttt{valid}(gs_1) = \texttt{True} \quad \{\{gs_1.\texttt{valid} \leadsto \mathsf{False\ True\ new}\ 5\}\} \Rightarrow \\ \texttt{dId}(gs) = \texttt{dId}(gs_1) \quad \{\emptyset\} \end{pmatrix}$$

Rule 3: The name of an actor is unique and should comply with its naming convention in the cross-cutting concerns. At all repository states, we require that all actors a_1 and a_2 with the same name also have the same description. In addition, the name of an actor should match the naming convention *nc* for actors. We employ a case statement here, because in the record type `NamingConv` the label `nc_actor` has type `Maybe String`. If two actors have the same name but a different description, we propose to change the name of the actor that has been changed. If the name of an author does not match its naming convention, we propose to change the name to the naming convention. Our system associates the documents {`Goals*.xml`,`Cross*.xml`}.

$$\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, a_1 \in \texttt{concatMap}(\texttt{existingactors}, \texttt{repGoalsScope}(t)) \bullet$$
$$\forall\, a_2 \in \texttt{concatMap}(\texttt{existingactors}, \texttt{repGoalsScope}(t)) \bullet$$
$$\begin{pmatrix} \texttt{name}(a_1) = \texttt{name}(a_2) \quad \begin{matrix}\{a_1.\texttt{name} \leadsto \texttt{name}(a_1) +\!\!+ \texttt{todo}\ \mathsf{True\ new}\ 10\}, \\ \{a_2.\texttt{name} \leadsto \texttt{name}(a_2) +\!\!+ \texttt{todo}\ \mathsf{True\ new}\ 10\}\end{matrix} \Rightarrow \\ \texttt{desc}(a_1) = \texttt{desc}(a_2) \quad \{\emptyset\} \end{pmatrix} \wedge$$
$$\begin{pmatrix} \exists\, nc \in \texttt{map}(\texttt{nc_actor}, \texttt{map}(\texttt{conventions}, \texttt{repCrosscut}(t))) \bullet \\ \texttt{match}\,(\texttt{name}(a_1), \texttt{case}(nc, \{\texttt{Just} \to \texttt{id}, \texttt{Nothing} \to _\})) \\ \{\{a_1.\texttt{name} \leadsto \texttt{case}(nc, \{\texttt{Just} \to \texttt{id}, \texttt{Nothing} \to _\})\ \mathsf{False}\ 10\ \}\} \end{pmatrix}$$

Rule 4: The cross-cutting concerns of the fine specification should contain a style guide. For all fine specifications *spec*, we require that there exists a cross-cutting concerns document *cc* that, if part of *spec*, has a non-empty style guide *stG*. A non-empty style guide contains button types or screen types. Notice that we access the cross-cutting concerns documents at the last modification state of the specification *spec*. Inconsistencies can be repaired by "downgrading" the specification to a coarse specification, or removing the document *cc* from the specification *spec*, or adding part of a style guide (we propose to add a dummy button type or screen type). Our system associates the documents {`Spec*.xml`,`Cross*.xml`}.

$$\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, spec^{\mathsf{CHG}} \in \texttt{repSp}(t) \bullet$$
$$\texttt{specKind}(spec) = \texttt{Fine} \quad \{\{spec.\texttt{specKind} \leadsto \mathsf{Coarse\ True\ new}\ 50\}\} \quad \Rightarrow$$
$$\exists\, cc \in \texttt{repCrosscut}(\texttt{dState}(spec)) \bullet$$
$$\texttt{dId}(cc) \in \texttt{doc_crosscut}(spec) \quad \{\{spec.\texttt{doc_crosscut} \leadsto \texttt{doc_crosscut}(spec) \setminus [\texttt{dId}(cc)]\ \mathsf{True}\ \ 15\}\} \quad \Rightarrow$$
$$\forall\, stG^{\mathsf{CHG}} \in [\texttt{styleguide}(cc)] \bullet$$
$$\begin{pmatrix} \texttt{buttonTs}(stG) \neq [] \\ stG.\texttt{buttonTs} \leadsto \texttt{SGType} \begin{matrix}\texttt{name} = \texttt{todo}, \texttt{desc} = \texttt{todo}, \\ \texttt{ref_layout} = \texttt{todo}\end{matrix} \quad \mathsf{False}\ 1 \end{pmatrix} \vee$$
$$\begin{pmatrix} \texttt{screenTs}(stG) \neq [] \\ stG.\texttt{screenTs} \leadsto \texttt{SGType} \begin{matrix}\texttt{name} = \texttt{todo}, \texttt{desc} = \texttt{todo}, \\ \texttt{ref_layout} = \texttt{todo}\end{matrix} \quad \mathsf{False}\ 1 \end{pmatrix}$$

Rule 19: The name of a use case is unique and should comply with its naming convention in the cross-cutting concerns. Essentially, this rule follows the shape of Rule 3. It is, however, more complicated to collect

all use cases, because they are spread over multiple components. We explicitly quantify over components, in order to better identify use cases. Without explicit quantification, repairs would contain use cases only but not their components. We use the function symbol `flattenComp` to flatten components; that way we can deal with recursive components as if they were first-order types (i.e., non-recursive). Our system associates the documents {`UC*.xml`,`Cross*.xml`}.

$$\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \\
\forall\, c_1^{\mathsf{KEEP}} \in \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{concatMap}(\mathtt{ucComp}, \mathtt{repUsecase}(t))) \bullet \\
\forall\, c_2^{\mathsf{KEEP}} \in \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{concatMap}(\mathtt{ucComp}, \mathtt{repUsecase}(t))) \bullet \\
\forall\, u_1 \in \mathtt{elements}(c_1) \bullet \forall\, u_2 \in \mathtt{elements}(c_2) \bullet \\
\left(\begin{array}{ll}
\mathtt{name}(u_1) = \mathtt{name}(u_2) & \begin{array}{l}\{u_1.\mathtt{name} \rightsquigarrow \mathtt{name}(u_1) +\!\!+ \mathtt{todo}\ \mathsf{True\ new}\ 10\}, \\ \{u_2.\mathtt{name} \rightsquigarrow \mathtt{name}(u_2) +\!\!+ \mathtt{todo}\ \mathsf{True\ new}\ 10\}\end{array} \quad \Rightarrow \\
u_1 = u_2 & \{\emptyset\}
\end{array} \right) \wedge \\
\left(\begin{array}{l}
\exists\, nc \in \mathtt{map}(\mathtt{nc_uc}, \mathtt{map}(\mathtt{conventions}, \mathtt{repCrosscut}(t))) \bullet \\
\mathtt{match}\,(\mathtt{name}(u_1), \mathsf{case}(nc, \{\mathtt{Just} \rightarrow \mathtt{id}, \mathtt{Nothing} \rightarrow _\})) \\
\qquad \{\{u_1.\mathtt{name} \rightsquigarrow \mathsf{case}(nc, \{\mathtt{Just} \rightarrow \mathtt{id}, \mathtt{Nothing} \rightarrow _\})\ \mathsf{False}\ 10\ \}\}
\end{array} \right)
\end{array}$$

Rule 21: The preconditions of a use case should be satisfiable from the postconditions of all other use cases. We require that, if a use case u has preconditions, these preconditions are implied by the conjunction of the postconditions of all other use cases. The problem is to collect the postconditions from all other use cases in a list, to which we can apply the variant constructor `And`. We collect all use cases by $\mathtt{concatMap}(\mathtt{elements}, \mathtt{concatMap}(\ldots))$. This list is filtered, such that only those use cases remain that have a name different from u's name. From these use cases, we take the postconditions. We must not employ a quantifier here, because all other postconditions *together* should imply the preconditions of u. Clearly, our system cannot resolve inconsistencies for this rule; repairs will propose to change the use case u, such that its preconditions are implied by the postconditions of all other use cases. Our system associates the documents {`UC*.xml`}.

$$\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \\
\forall\, c^{\mathsf{KEEP}} \in \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{concatMap}(\mathtt{ucComp}, \mathtt{repUsecase}(t))) \bullet \\
\forall\, u \in \mathtt{elements}(c) \bullet \\
\mathtt{And} \left(\begin{array}{l}
\mathtt{catMaybes}(\mathtt{map}(\mathtt{postcons}, \\
\qquad \mathtt{filter}{\notin}\,([\mathtt{name}(u)], \mathtt{name}, \\
\qquad\qquad \mathtt{concatMap}(\mathtt{elements}, \mathtt{concatMap}(\mathtt{flattenComp}, \\
\qquad\qquad\qquad \mathtt{concatMap}(\mathtt{ucComp}, \mathtt{repUsecase}(t)))))))
\end{array} \right) \quad |\!\Rightarrow \\
\mathsf{case}(\mathtt{precons}(u), \{\mathtt{Just} \rightarrow \mathtt{id}, \mathtt{Nothing} \rightarrow \mathtt{LTrue}\})
\end{array}$$

Rule 30: The conjunction of the postconditions of all analysis functions used by a use case u should imply the postconditions of u. Essentially, the rule follows the shape of Rule 21. Use cases that do not use analysis functions are neglected. Our system associates the documents {`UC*.xml`,`AnaFun*.xml`}.

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \\
\forall\, u^{\mathsf{CHG}} \in \texttt{concatMap}(\texttt{elements}, \texttt{concatMap}(\texttt{flattenComp}, \quad \bullet \\
\qquad \texttt{concatMap}(\texttt{ucComp}, \texttt{repUsecase}(t)))) \\
\texttt{null}(\texttt{ref_fun}(u)) \quad \{\emptyset\} \qquad \vee \\
\texttt{And} \left(\begin{array}{l} \texttt{catMaybes}(\texttt{map}(\texttt{postcons}, \\ \qquad \texttt{filter}\in(\texttt{ref_fun}(u), \texttt{name}, \\ \qquad\qquad \texttt{concatMap}(\texttt{elements}, \texttt{concatMap}(\texttt{flattenComp}, \\ \qquad\qquad\qquad \texttt{concatMap}(\texttt{afComp}, \texttt{repAnaFun}(t))))))) \end{array} \right) \quad \models\!\Rightarrow \\
\texttt{case}(\texttt{precons}(u), \{\texttt{Just} \rightarrow \texttt{id}, \texttt{Nothing} \rightarrow \texttt{LTrue}\})
\end{array}
$$

Rule 41: In the finished fine specification, each relation should have cardinalities associated with its entity types. For each finished fine specification *spec*, we require for all data models *dm* part of *spec* the following: All relations *rel* of each entity type *et* must have a non-empty target cardinality. We propose to resolve inconsistencies by either “downgrading” the specification or adding a `todo` marker to the target cardinality. Our system associates the documents {`Spec*.xml`,`DataM*.xml`}.

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, spec^{\mathsf{CHG}} \in \texttt{repSp}(t) \bullet \\
\quad \begin{array}{ll} \texttt{specKind}(spec) = \texttt{Fine} & \{\{spec.\texttt{specKind} \leadsto \mathsf{Coarse\ True}\ 50\}\} \quad \wedge \\ \texttt{status}(spec) = \texttt{Finished} & \{\{spec.\texttt{status} \leadsto \mathsf{InProgress\ True}\ 40\}\} \end{array} \quad \Rightarrow \\
\quad \forall\, dm^{\mathsf{KEEP}} \in \texttt{repDataMod}(\texttt{dState}(spec)) \bullet \\
\quad\; \texttt{dId}(dm) = \texttt{case}(\texttt{doc_datamod}(spec), \{\texttt{Just} \rightarrow \texttt{id}, \texttt{Nothing} \rightarrow _\}) \quad \{\emptyset\} \qquad \Rightarrow \\
\quad\; \forall\, et^{\mathsf{KEEP}} \in \texttt{concatMap}(\texttt{elements}, \texttt{concatMap}(\texttt{flattenComp}, \texttt{compsDM}(dm))) \bullet \\
\quad\; \forall\, rel^{\mathsf{CHG}} \in \texttt{relatesWith}(et) \bullet \\
\quad\; \texttt{targetcard}(rel) \neq _ \qquad \{\{rel.\texttt{targetcard} \leadsto \mathsf{todo\ False}\ 1\}\}
\end{array}
$$

Rule 48: The lower bound of a range data type should be smaller than its upper bound. We require that for each data type *dt* its lower bound is strictly smaller than its upper bound. Recall that `rngFrom` and `rngTo` are total functions. Our system associates the documents {`DataT*.xml`}.

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, dt \in \texttt{concatMap}(\texttt{dataTs}, \texttt{repDataTs}(t)) \bullet \\
\texttt{rngFrom}(\texttt{kind}(dt)) < \texttt{rngTo}(\texttt{kind}(dt))
\end{array}
$$

Rule 64: Each analysis function used by a dialog should also be used by each use case associated with this dialog. For each analysis function *fun*, referenced by a dialog *dlg*, we require that *fun* is also referenced by all use cases u called by *dlg*. We can resolve inconsistencies by either removing from the dialog *dlg* the use case reference to u or by adding to u a reference to the analysis function *fun*. Our system associates the documents {`Dialog*.xml`,`UC*.xml`}.

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \\
\forall\, dlg^{\mathsf{CHG}} \in \texttt{concatMap}(\texttt{elements}, \texttt{concatMap}(\texttt{flattenComp}, \bullet \\
\qquad \texttt{concatMap}(\texttt{diaComp}, \texttt{repDialog}(t)))) \\
\forall\, fun \in \texttt{concatMap}(\texttt{actFun}, \texttt{actions}(dlg)) \bullet \\
\forall\, u^{\mathsf{CHG}} \in \texttt{concatMap}(\texttt{elements}, \texttt{concatMap}(\texttt{flattenComp}, \quad \bullet \\
\qquad \texttt{concatMap}(\texttt{ucComp}, \texttt{repUsecase}(t)))) \\
\begin{array}{ll} \texttt{name}(u) \in \texttt{usecases}(dlg) & \{\{dlg.\texttt{usecases} \leadsto \texttt{usecases}(dlg) \setminus [\texttt{name}(u)]\ \mathsf{True}\ 3\ \}\} \quad \Rightarrow \\ fun \in \texttt{ref_fun}(u) & \{\{u.\texttt{ref_fun} \leadsto fun : (\texttt{ref_fun}(u))\ \mathsf{False}\ 1\ \}\} \end{array}
\end{array}
$$

Rule 70: The definition of a term in the glossary does not change significantly over time. We require that for all defined terms $term_o$ in an

old glossary there exists in a newer glossary a term $term_n$ that has the same name as $term_o$ and the description of which is similar to that of $term_o$. If this rule is violated, we propose to change the name and the description of $term_n$ as given by $term_o$. Our system associates the documents {`Gloss*.xml`}.

$$\forall\, t_o^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, t_n^{\mathsf{KEEP}} \in \texttt{repStates} \bullet$$
$$t_o < t_n \Rightarrow \begin{pmatrix} \forall\, g_o^{\mathsf{KEEP}} \in \texttt{repGlossary}(t_o) \bullet \forall\, term_o^{\mathsf{KEEP}} \in \texttt{terms}(g_o) \bullet \\ \exists\, g_n \in \texttt{repGlossary}(t_n) \bullet \exists\, term_n \in \texttt{terms}(g_n) \bullet \\ \texttt{name}(term_o) = \texttt{name}(term_n) \quad \{\{term_n.\texttt{name} \leadsto \texttt{name}(term_o)\ \mathsf{False}\ 5\ \}\} \quad \wedge \\ \texttt{desc}(term_o) \equiv \texttt{desc}(term_n) \quad \{\{term_n.\texttt{desc} \leadsto \texttt{desc}(term_o)\ \mathsf{False}\ 2\ \}\} \end{pmatrix}$$

Rule 73: A use case should use analysis functions from its own component only. For all use case components c, and for all analysis functions f_{ref}, referenced by a use case, we require the following: There should exist an analysis function component c_{fun} with the same name as c, which includes an analysis function with the same name as f_{ref}. Inconsistencies can be resolved by either changing the name of the analysis function's component c_{fun} or by changing the analysis function reference f_{ref}. Our system associates the documents {`UC*.xml`,`AnaFun*.xml`}.

$$\begin{aligned}
&\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, uD^{\mathsf{CHG}} \in \texttt{repUsecase}(t) \bullet \\
&\forall\, c^{\mathsf{CHG}} \in \texttt{concatMap}(\texttt{flattenComp}, \texttt{ucComp}(uD)) \bullet \forall\, u^{\mathsf{CHG}} \in \texttt{elements}(c) \bullet \\
&\forall\, f_{ref} \in \texttt{ref_fun}(u) \bullet \\
&\exists\, c_{fun} \in \texttt{concatMap}(\texttt{flattenComp}, \texttt{concatMap}(\texttt{afComp}, \texttt{repAnaFun}(t))) \bullet \\
&\texttt{name}(c) = \texttt{name}(c_{fun}) \quad \{\{c_{fun}.\texttt{name} \leadsto \texttt{name}(c)\ \mathsf{False}\ 10\ \}\} \quad \wedge \\
&\exists\, f \in \texttt{elements}(c_{fun}) \bullet \\
&f_{ref} = \texttt{name}(f) \quad \{\{f_{ref} \leadsto \texttt{name}(f)\ \mathsf{False}\ 1\ \}\}
\end{aligned}$$

Rule 74: In the data model, an entity type has more relations to entity types within the same component than to entity types within a different component. For all data model components c and for all their entity types et, we require the following. The number of all relations from et to entity types within the component c is greater than or equal to the half of the number of all relations from et. This is equivalent to the original requirement. Our system associates the documents {`DataM*.xml`}.

$$\begin{aligned}
&\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, dm^{\mathsf{KEEP}} \in \texttt{repDataMod}(t) \bullet \\
&\forall\, c^{\mathsf{CHG}} \in \texttt{concatMap}(\texttt{flattenComp}, \texttt{compsDM}(dm)) \bullet \forall\, et^{\mathsf{CHG}} \in \texttt{elements}(c) \bullet \\
&\texttt{length}(\texttt{relatedWith}(et)) \not{/}_{\texttt{Int}}\ 2 \ \leq \\
&\texttt{length}(\texttt{filter}\in (\texttt{map}(\texttt{etName}, \texttt{map}(\texttt{ref_entitytype}, \texttt{relatesWith}(et))), \\
&\qquad \texttt{name}, \texttt{elements}(c)))
\end{aligned}$$

Rule 80: A system-supported activity in a business process of the coarse specification is also included in the fine specification and is also supported by the system. For every finished fine specification s_f, we require for all finished coarse specifications s_c that existed before the following. For all business processes bp_c that are part of the coarse specification, there exists a corresponding business process bp_f in the fine specification, such that the following holds. For each activity act_c in the business process bp_c, there exists a corresponding activity in the business process bp_f that is supported by the system in the same way as act_c is. For resolving inconsistencies, we

consider to "downgrade" the fine specification, or to adapt business processes or activities of the fine specification. Our system associates the documents {Spec*.xml,BP*.xml}.

$$
\begin{array}{l}
\forall\, t_f^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, t_c^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, s_f^{\mathsf{CHG}} \in \mathtt{repSp}(t_f) \bullet \\
\left(\begin{array}{lll}
t_c < t_f & & \wedge \\
\mathtt{specKind}(s_f) = \mathtt{Fine} & \{\{s_f.\mathtt{specKind} \rightsquigarrow \mathtt{Coarse}\ \mathsf{True}\ 50\}\} & \wedge \\
\mathtt{status}(s_f) = \mathtt{Finished} & \{\{s_f.\mathtt{status} \rightsquigarrow \mathtt{InProgress}\ \mathsf{True}\ 40\}\} &
\end{array}\right) \Rightarrow \\
\quad \forall\, s_c^{\mathsf{KEEP}} \in \mathtt{filter}{\in}\,([\mathtt{dId}(s_f)], \mathtt{dId}, \mathtt{repSp}(t_c)) \bullet \\
\quad (\mathtt{specKind}(s_c) = \mathtt{Coarse} \;\wedge\; \mathtt{status}(s_c) = \mathtt{Finished}) \Rightarrow \\
\quad\; \forall\, bpD_c^{\mathsf{KEEP}} \in \mathtt{filter}{\in}\,(\mathtt{doc_busproc}(s_c), \mathtt{dId}, \mathtt{repBusProc}(\mathtt{dState}(s_c))) \bullet \\
\quad\; \forall\, bp_c^{\mathsf{KEEP}} \in \mathtt{processes}(bpD_c) \bullet \\
\quad\;\; \exists\, bpD_f \in \mathtt{filter}{\in}\,(\mathtt{doc_busproc}(s_f), \mathtt{dId}, \mathtt{repBusProc}(\mathtt{dState}(s_f))) \bullet \\
\quad\;\; \exists\, bp_f \in \mathtt{processes}(bpD_f) \bullet \\
\quad\;\;\; \mathtt{name}(bp_f) = \mathtt{name}(bp_c) \quad \{\{bp_f.\mathtt{name} \rightsquigarrow \mathtt{name}(bp_c)\ \mathsf{False}\ 4\}\} \quad \wedge \\
\quad\;\;\; \forall\, act_c^{\mathsf{KEEP}} \in \mathtt{activities}(bp_c) \bullet \exists\, act_f \in \mathtt{activities}(bp_f) \bullet \\
\quad\;\;\; \mathtt{name}(act_f) = \mathtt{name}(act_c) \quad \{\{act_f.\mathtt{name} \rightsquigarrow \mathtt{name}(act_c)\ \mathsf{False}\ 2\}\} \quad \wedge \\
\quad\;\;\; \mathtt{systems}(act_f) = \mathtt{systems}(act_c) \quad \{\{act_f.\mathtt{systems} \rightsquigarrow \mathtt{systems}(act_c)\ \mathsf{False}\ 1\}\}
\end{array}
$$

Formalizing the above requirement as a consistency rule results in much more precision by making implicit assumptions explicit:

- Coarse specification and fine specification are the same document at different repository states.
- A fine specification may be developed without coarse specification. Notice that this holds w.r.t. Rule 80 only. Fine specification including business processes should not be developed without coarse specification, in order to fulfill Rule 81.
- Business processes of the coarse specification should be part of the fine specification, but they may be included in another document.
- Activities within a business process of the coarse specification should exist in the *same* business process of the fine specification.
- Activities unsupported in the coarse specification are also unsupported in the fine specification.

In the above rule, we do *not* require that referenced documents really exist. This should be formalized by another rule, in order to keep diagnoses and S-DAGs comprehensible. In fact, in an earlier version of Rule 80 we also required the validity of the document references. This resulted, however, in incomprehensible S-DAGs.

Rule 82: A specification intended for a species of readers only contains documents intended for this reader species. We require that for each specification document *sp* its referenced reading instruction document exists. If there exists no matching reading instructions document, we propose either to adapt the reference in the specification or to change the name of the reading instructions document. For a matching reading instructions document *ri*, we require the following. If the specification *sp* contains business processes,

then the readers of *sp* should be included in the reading instructions for business processes. For resolving inconsistencies, we propose to either drop the business processes from *sp* or to add the readers of *sp* to the reading instructions for business processes. Other parts of the specification (analysis functions, cross-cutting concerns, data models, data types, use cases) are processed similarly. Since the data type `DocReading` carries an extra label for each reading instruction, the below rule is actually larger; for brevity, we show the formalization for business processes and cross-cutting concerns only. Our system associates the documents {`Spec*.xml`,`Reading*.xml`}.

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, sp^{\mathsf{CHG}} \in \texttt{repSp}(t) \bullet \\
\exists\, ri \in \texttt{repReadingInst}(\texttt{dState}(sp)) \bullet \\
\texttt{doc_reading}(sp) = \texttt{dId}(ri) \quad \begin{array}{l} \{sp.\texttt{doc_reading} \rightsquigarrow \texttt{dId}(ri)\ \mathsf{False\ new}\ 5\}, \\ \quad ri.\texttt{dId} \rightsquigarrow \texttt{doc_reading}(sp)\ \mathsf{False\ new}\ 10, \end{array} \qquad \wedge \\
\left(\begin{array}{l}
\neg\,(\texttt{null}(\texttt{doc_busproc}(sp)) \quad \{\{sp.\texttt{doc_busproc} \rightsquigarrow []\ \mathsf{False}\ \ 5\}\}) \qquad \Rightarrow \\
\texttt{readers}(sp) \in \texttt{concatMap}\ \ \texttt{shouldRead}, \texttt{case}\ \ \texttt{ri_busproc}(ri), \begin{array}{l} \texttt{Just} \to \texttt{id}, \\ \texttt{Nothing} \to [] \end{array} \\
\qquad\qquad \{\{ri.\texttt{ri_busproc} \rightsquigarrow \texttt{addRI}(\texttt{ri_busproc}(ri), \texttt{readers}(sp))\ \mathsf{False}\ \ 2\}\}
\end{array}\right) \wedge \\
\left(\begin{array}{l}
\neg\,(\texttt{null}(\texttt{doc_crosscut}(sp)) \quad \{\{sp.\texttt{doc_crosscut} \rightsquigarrow []\ \mathsf{False}\ \ 5\}\}) \qquad \Rightarrow \\
\texttt{readers}(sp) \in \texttt{concatMap}\ \ \texttt{shouldRead}, \texttt{case}\ \ \texttt{ri_crosscut}(ri), \begin{array}{l} \texttt{Just} \to \texttt{id}, \\ \texttt{Nothing} \to [] \end{array} \\
\qquad\qquad \{\{ri.\texttt{ri_crosscut} \rightsquigarrow \texttt{addRI}(\texttt{ri_crosscut}(ri), \texttt{readers}(sp))\ \mathsf{False}\ \ 2\}\}
\end{array}\right)
\end{array}
$$

Rule 84: A document in status 'in progress' should change its status to 'quality assurance' only. At all repository states, we require for all documents d the following. If d already existed in the previous state, then its status should not have changed from 'in progress' to 'finished'. Our system associates the documents {`*.xml`}. The rule is *not* filtered due to the use of `prevState`.

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\, d^{\mathsf{CHG}} \in \texttt{repDocSDM}(t) \bullet \\
\texttt{dState}(d) = t \quad \{\emptyset\} \qquad \Rightarrow \\
\ \forall\, d_{prev}^{\mathsf{KEEP}} \in \texttt{repDocSDM}(\texttt{prevState}(t)) \bullet \\
\ (\texttt{dId}(d) = \texttt{dId}(d_{prev}) \quad \{\emptyset\} \qquad \wedge \qquad \texttt{status}(d_{prev}) = \texttt{InProgress}) \qquad \Rightarrow \\
\quad \neg\,(\texttt{status}(d) = \texttt{Finished} \quad \{\{d.\texttt{status} \rightsquigarrow \texttt{InProgress}\ \mathsf{True}\ \ 40\}\})
\end{array}
$$

Chapter 12

The Ski School

In this chapter, we apply our consistency rules to a concrete software specification. We shall see that consistency maintenance is necessary and our approach is useful in practice. First, we introduce an example specification. In Sect. 12.2, we show how this specification might be developed with the help of a revision control system. The careful reader will notice that the specification includes some inconsistencies w.r.t. the rules from Chapter 11. Sect. 12.3 contains augmented S-DAGs showing these inconsistencies and possible repair actions for individual rules. In Sect. 12.4, we present repair collections derived by our system. A performance summary is given in Sect. 12.5. In Chapter 13, we discuss lessons learnt from our case study. Fig. 12.1 illustrates the context of this chapter.

12.1 The Ski School Specification

The ski school is a sample sd&m software specification used in-house for teaching the application of analysis modules. With kind permission from sd&m, we use part of this specification to illustrate the application of our consistency maintenance approach. In the following subsections, we give an overview of the finished fine specification. For brevity, we only describe parts relevant for our consistency rules. The specification consists of 18 documents: four specification documents (we have four reader species), one goals, scope, constraints document, one glossary, one reading instruction, one cross-cutting concerns document, three business processes (in three business process documents), ten use cases (in three use case documents), twelve analysis functions (in one analysis functions document), two dialogs (in one dialogs document), one data model, and one data types document. The specified system is subdivided into three components. We assume that the specification is developed by four software engineers, who perform 24 check-ins to the repository.

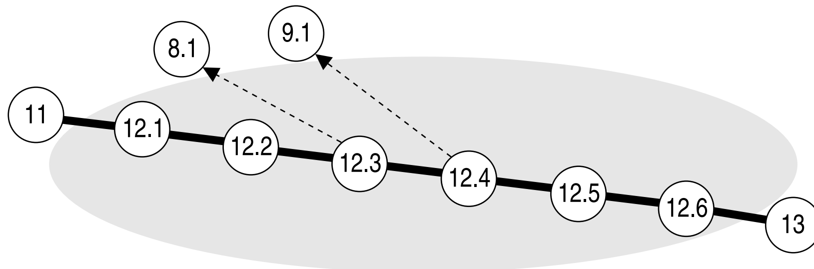


Figure 12.1: Chapter 12 in context

Reader Species	Business Processes	Use Cases	Analysis Functions	Data Model	Data Types	Dialogs
user	×	×				×
designer	×	×	×	×	×	×
test team		×	×	×		×
management						

Table 12.1: Contents of the ski school specification for different reader species

Reader Species	Description
user	Our customer, the staff of the ski school.
designer	The software designers who develop the system.
test team	Those people test our software.
management	The project management. They coordinate the whole work.

Table 12.2: Reader species listed in the reading instructions

12.1.1 Specification Documents

We distinguish four reader species: users, software designers, test team, and management. Consequently, we develop four different specifications. Tab. 12.1 lists the contents of each specification. Besides these modules, each specification contains goals, scope, constraints, cross-cutting concerns, the glossary, and reading instructions.

12.1.2 Goals, Scope, Constraints

We define two actors: 'Planning staff ski school' and 'Office staff ski school.'

12.1.3 Glossary

The glossary defines the following terms:
Course: A course in the ski school consists of some classes. Usually, it is held by one teacher who should not change during the course. We have different courses for beginners, children, seniors, and advanced skiers. A course follows a fixed schedule. (see also Teacher, Class)
Class: A class is part of a course. In a class, a teacher gives a closed part of a course. There should be less than eleven students in a class. (see also Course, Student)
Teacher: He is the one who should transfer the knowledge about skiing to the students. (see also Course, Class)
Student: The participants of a course — they should learn skiing. We call them "students," as students at school.

12.1.4 Reading Instructions

The reading instructions include reader species for our specification (Tab. 12.2) and determine the parts of a specification a reader species should read (Tab. 12.3).

Module	User	Management	Designer	Test Team
business processes	×		×	
use cases	×		×	×
analysis functions			×	×
data model			×	×
data types			×	
dialogs	×		×	×
cross-cutting concerns	×	×	×	×
glossary	×	×	×	×
goals, scope, constraints	×	×	×	×
reading instructions	×	×	×	×

Table 12.3: Who should read which part of a specification?

Result Type	Naming Convention
actor	*
business process	BP_[*]
activity in a business process	Act_[*]
component	Com_[*]
use case	UC_[*]
analysis function	AF_[*]
entity type	ET_[*]
attribute for an entity type	Att_[*]
technical data type	DT_[*]
dialog	Dia_[*]
screen of a dialog	Scr_[*]
action of a dialog	A_[*]
reader species	*

Table 12.4: Naming conventions in the cross-cutting concerns

12.1.5 Cross-Cutting Concerns

For each result type, we define a naming convention, given by a regular expression (see Tab. 12.4).

12.1.6 Business Processes

For the ski school, we define three business processes, shown in Fig. 12.2, 12.3, and 12.4. According to the naming conventions, the name of each business process begins with the upper case letters "BP."

Every week the business process BP_[Plan classes] is used to plan the classes for a course. It requires that the courses are planned for the season and that the availabilities of the teachers are clear. First, we determine the class utilization. If we have too many requests for a course, we try to either create a new course and classes (when there are many similar requests) or to move some students to other classes. Creating a new course requires to assign a teacher to that course using the business process BP_[Assign teacher to classes]. If there are too few requests for a class, we check whether some classes can be merged, in order to achieve a good class utilization. If no classes can be merged, we delete the class with too few requests.

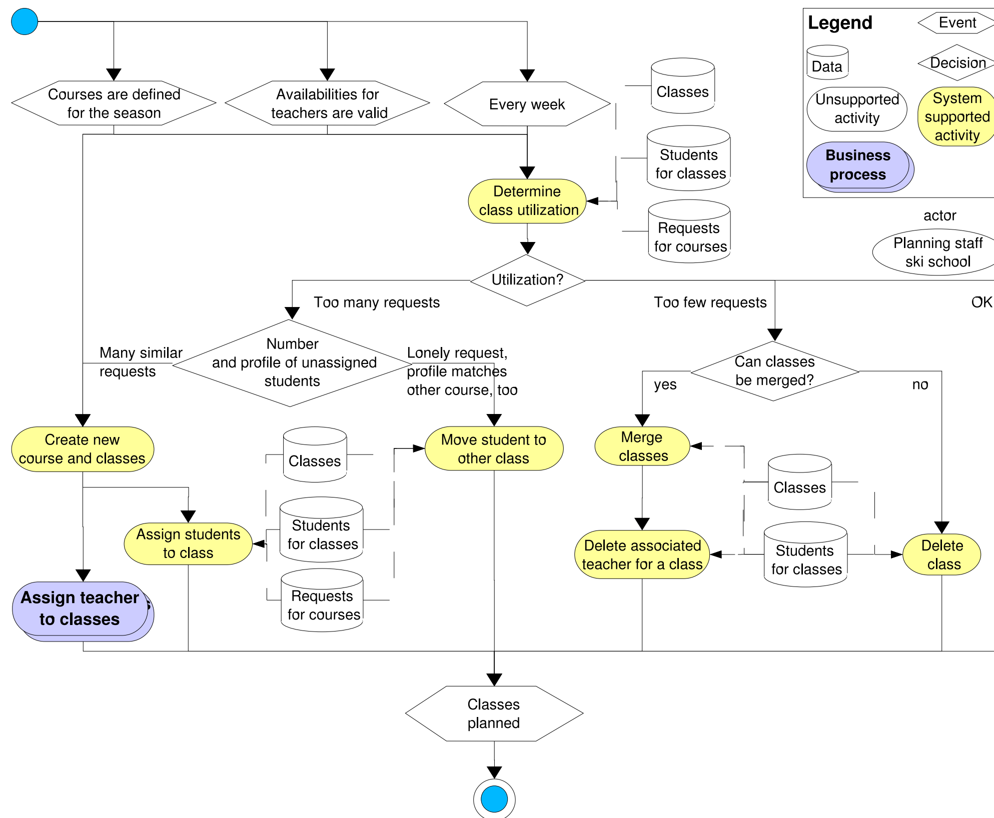


Figure 12.2: Business process BP_[Plan classes]

The business process BP_[Assign teacher to classes] requires that all classes of a course are planned. Then our system suggests a teacher for this course. If a teacher is available for all classes in the course or if he can be disengaged from another course, we assign this teacher. Otherwise, we try to split the given course and to assign the teacher at least to some classes of the given course.

The business process BP_[Register students] registers a student and assigns him to a course. In order to perform this business process, the classes for a course must be planned and the student's request must arrive before 0900 a. m. First, our system creates a course summary from which we get courses that match the criteria given by the student. If a matching course is found, we register the student and assign him to the course. Otherwise, we try to find a course for the student at another date, assign the student to part of a course only, split the registration, or (in the end) find another course type. If all that fails, we set the student to a wish list. During the development of the specification, we will drop support for wish lists.

12.1.7 Use Cases

In this section, we summarize the use cases defined for activities in the business processes above. For each use case, we give a short description and determine pre- and postconditions. We also list inputs and references to analysis functions.

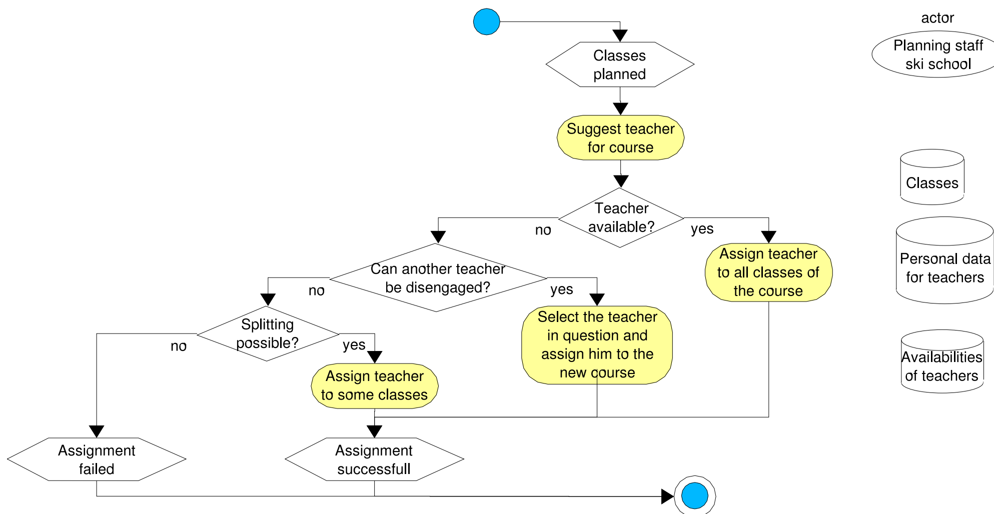


Figure 12.3: Business process BP_[Assign teacher to classes]

Fig. 12.5 gives an overview of all use cases and their actors. According to the naming conventions, the name of each use case begins with the upper case letters "UC." We subdivide the specified system by three components: Com_[Course planning], Com_[Internal data], and Com_[Course attendees].

Use Cases for the Business Process BP_[Plan classes]

All use cases below belong to the component Com_[Course planning].
Name: UC_[Create new course and classes]
Activities: Act_[Create new course and classes] (BP_[Plan classes])
Procedure: The user enters the course's type, start date, end date, and cadence (daily, weekly). If weekly is selected, then the day of week is needed, too. The system creates the course and sets its status to "planned." In addition, the system creates all classes for this course.
Preconditions: course type exists
Postconditions: course created, classes created, no assignment of teachers to course, no assignment of students to course
Inputs: course type, start date, end date, cadence
Analysis functions: AF_[Create course], AF_[Create classes for a course]

Name: UC_[Delete course]
Activities: Act_[Delete class], Act_[Delete associated teacher for a class] (BP_[Plan classes])
Procedure: The user enters the course he wants to delete. For each class, the system checks that there is no student assigned to this class. Then the system removes any assignments of teachers to the classes and makes the teachers available for the class dates. The system deletes all classes for the course. If the course is not part of the wish list, the system deletes the course.
Preconditions: course created
Postconditions: course removed

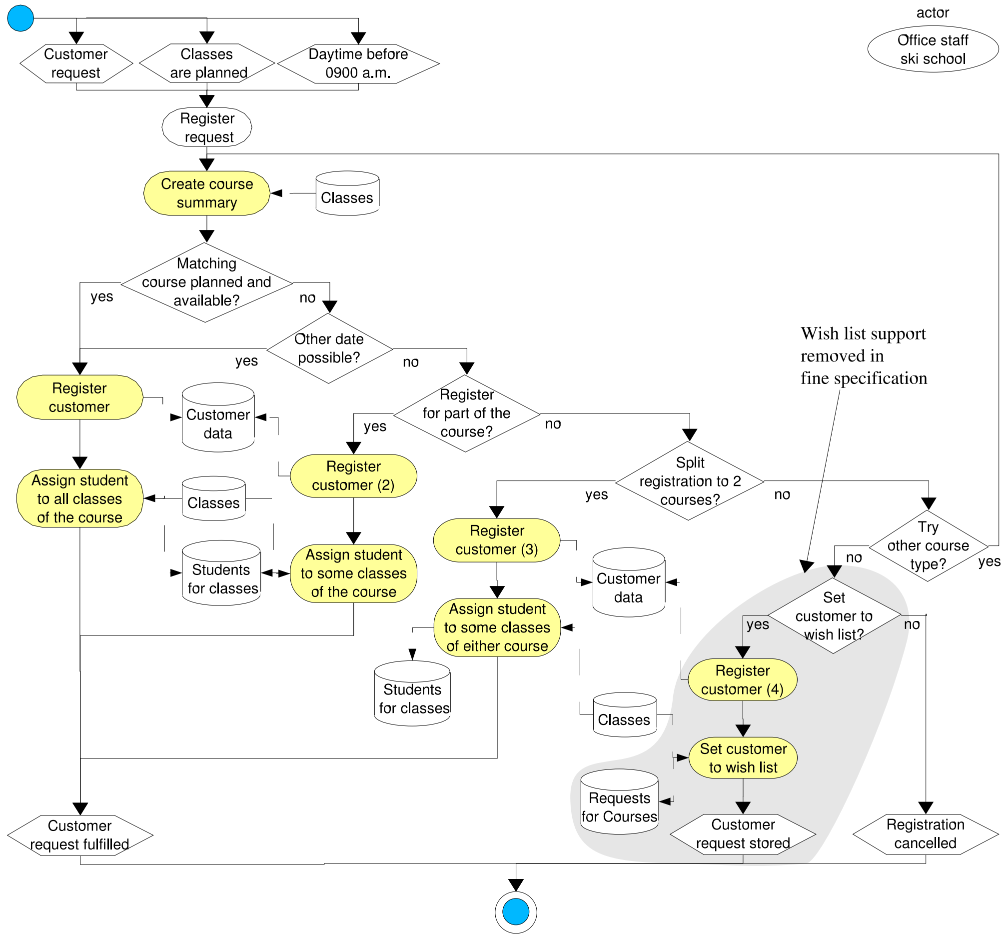


Figure 12.4: Business process BP_[Register students] (gray area removed in fine specification)

Inputs: course
Analysis functions: AF_[Remove assignment of teacher to class], AF_[Remove class], AF_[Remove course]

Name: UC_[Change assignments of students to classes]
Activities: Act_[Move student to other class], Act_[Merge classes] (BP_[Plan classes])
Procedure: The user selects some students attending a class c_1. Then he chooses the target class c_2 to which the students should be moved. The system assigns the selected students to c_2 (via the use case UC_[Assign student to class]) and removes their assignment to c_1.
Preconditions: classes created
Postconditions:
Inputs: class, student
Analysis functions:
Uses: UC_[Assign student to class]

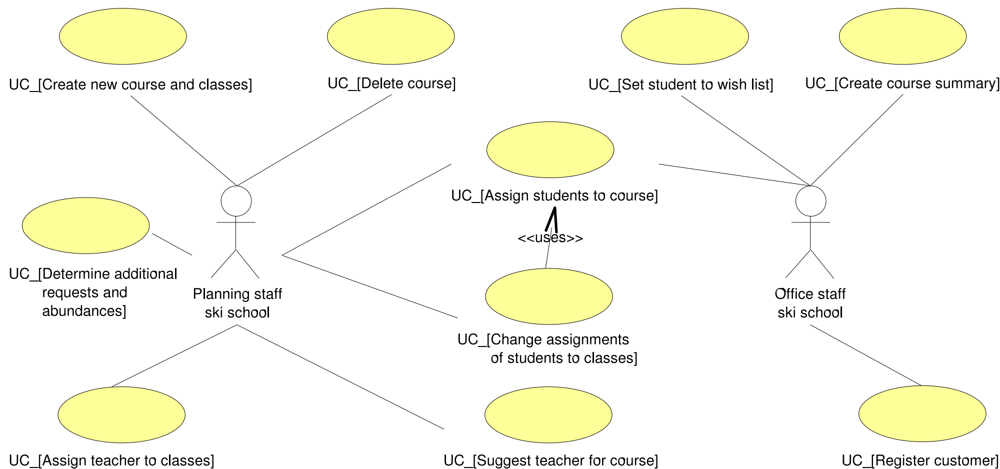


Figure 12.5: Overview of use cases

Name: UC_[Determine additional requests and abundances]
Activities: Act_[Determine class utilization] (BP_[Plan classes])
Procedure: The user enters begin date and end date of the time period to plan. The system shows planned courses and classes, their proposed number of students, the number of actually assigned students, and the number of requests in the wish list for that class. In addition, the system shows courses that are not planned but requested by the wish list.
Preconditions: classes created
Postconditions: planning requests known
Inputs: begin date, end date
Analysis functions:

Use Cases for the Business Process BP_[Assign teacher to classes]

All use cases below belong to the component Com_[Internal data].

Name: UC_[Suggest teacher for course]
Activities: Act_[Suggest teacher for course] (BP_[Assign teacher to classes])
Procedure: The user enters the course and criteria to search for a teacher. The system presents a list of all teachers that match the search criteria and that are available at the course dates.
Preconditions: course created
Postconditions: teacher suggestions for a course
Inputs: course, search criteria
Analysis functions: AF_[Filter teachers], AF_[Check availability]

Name: UC_[Assign teacher to classes]
Activities: Act_[Assign teacher to all classes of the course], Act_[Select the teacher in question and assign him to the new course], Act_[Assign teacher to some classes] (BP_[Assign teacher to classes])
Procedure: The user chooses some classes and the teacher to assign. The system checks availability of the teacher for the class dates and ensures that every

class has enough teachers assigned (for children classes we need two teachers). Finally, the system assigns the teacher to the classes and updates the teacher's schedule.
Preconditions: classes created
Postconditions: teacher assigned to class
Inputs: class, teacher
Analysis functions: AF_[Check availability], AF_[Assign teacher to class]

Use Cases for the Business Process BP_[Register students]

All use cases below belong to the component Com_[Course attendees].

Name: UC_[Assign students to course]
Activities: Act_[Assign student to all classes of the course], Act_[Assign student to some classes of the course], Act_[Assign student to some classes of either course] (BP_[Register students]),
Act_[Assign students to class] (BP_[Plan classes])
Procedure: The user enters the student name and selects the requested course and classes. For each class, the system checks whether the maximum number of students is still respected. Also, the system determines conflicting classes to which the student is already assigned.
Preconditions: course created, classes created, customer registered
Postconditions: student assigned to class, no conflicting assignments to other classes
Inputs: student, course, class
Analysis functions: AF_[Check maximum students for class], AF_[Find classes for student by date]

Name: UC_[Create course summary]
Activities: Act_[Create course summary] (BP_[Register students])
Procedure: The user enters course type and begin date. The system creates a summary about the courses planned.
Preconditions: course created
Postconditions:
Inputs: course type, begin date
Analysis functions: AF_[Filter courses]

Name: UC_[Register customer]
Activities: Act_[Register customer], Act_[Register customer (2)], Act_[Register customer (3)], Act_[Register customer (4)] (BP_[Register students])
Procedure: The user enters customer data: name, age, gender, language, and address. The system adds the customer to the internal database.
Preconditions:
Postconditions: customer registered
Inputs: name, age, gender, language, address
Analysis functions: AF_[Add customer]

The following use case will be removed in the fine specification (at state 21). It is, however, part of the coarse specification.

Name: UC_[Set student to wish list]
Activities: Act_[Set customer to wish list] (BP_[Register students])
Procedure: The user enters the requested course type, begin date, end date, and schedule. The system checks whether there exists a course matching the given criteria. We distinguish between: (1) the course is planned but there are no places available (then we set the customer to the wish list), or (2) the course is not planned, or (3) there does not exist a matching course.
Preconditions: customer registered
Postconditions: request in wish list
Inputs: course type, begin date, end date, schedule
Analysis functions: AF_[Filter courses]

12.1.8 Analysis Functions

Analysis functions below describe specific functionality without user interaction. For each analysis function, we list its name, a short description, its parameters, its result, and pre- and postconditions. According to the naming conventions, the name of each analysis function begins with the upper case letters "AF."

Analysis Functions in the Component Com_[Course planning]

Name: AF_[Create course]
Description: Creates a new course and adds it to the database with status "planned."
Parameters: course type, start date, end date, cadence
Result: course
Preconditions: course type exists
Postconditions: course created, no assignment of teachers to course, no assignment of students to course

Name: AF_[Create classes for a course]
Description: Creates new classes for a course with status "planned."
Parameters: course, cadence
Result: set of classes
Preconditions: course created
Postconditions: classes created

Name: AF_[Remove assignment of teacher to class]
Description: Remove the assignment of a teacher to a class.
Parameters: teacher, class
Result: teacher
Preconditions: classes created
Postconditions: assignment of teacher to class removed

Name: AF_[Remove class]
Description: Remove a class of a course.
Parameters: course, class
Result: course

Preconditions: course created, assignment of teacher to class removed
Postconditions: class removed

Name: AF_[Remove course]
Description: Remove a course from the database.
Parameters: course
Result: -
Preconditions: class removed, course created
Postconditions: course removed

Analysis Functions in the Component Com_[Course attendees]

Name: AF_[Check maximum students for class]
Description: Checks whether the number of students assigned to a class is below the maximum.
Parameters: class
Result: Bool
Preconditions: classes created
Postconditions:

Name: AF_[Find classes for student by date]
Description: For a specific date, find classes that a student is assigned to.
Parameters: student, date
Result: set of classes
Preconditions: customer registered
Postconditions:

Name: AF_[Filter courses]
Description: Filter courses that match some criteria.
Parameters: course type, begin date, end date, schedule
Result: set of courses
Preconditions:
Postconditions:

Name: AF_[Add customer]
Description: Adds a customer to the database.
Parameters: name, age, gender, language, address
Result: student
Preconditions:
Postconditions: customer registered

Analysis Functions in the Component Com_[Internal data]

Name: AF_[Filter teachers]
Description: Determine teachers that match some criteria.
Parameters: search criteria
Result: set of teachers
Preconditions:
Postconditions:

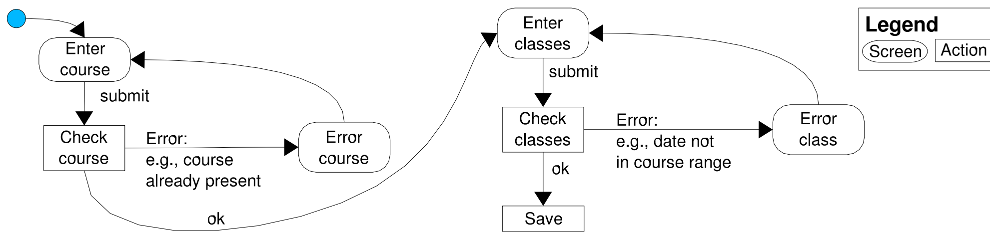


Figure 12.6: Interactivity diagram for dialog Dia_[New course and classes]

Name: AF_[Check availability]
Description: Check whether a teacher is available at a given period of time.
Parameters: begin date, end date, teacher
Result: Bool
Preconditions:
Postconditions:

Name: AF_[Assign teacher to class]
Description: Assign a teacher to a class.
Parameters: teacher, class
Result: teacher
Preconditions: classes created
Postconditions: teacher assigned to class

12.1.9 Dialogs

For our ski school, we define two dialogs, in order to input data necessary for the use cases UC_[Assign students to course] and UC_[Create new course and classes], respectively. Each dialog specification includes a description, an interactivity diagram (IAD), and definitions of screens and actions used in the IAD. Screens carry in- and output fields, which reference entity types defined in the data model (see Fig. 12.8)

Dialogs in the Component Com_[Course planning]

Name: Dia_[New course and classes]
Description: The dialog is used to enter data needed for a new course and its classes. Finally, the new course is stored in the database.
Use case: UC_[Create new course and classes]
Screens:

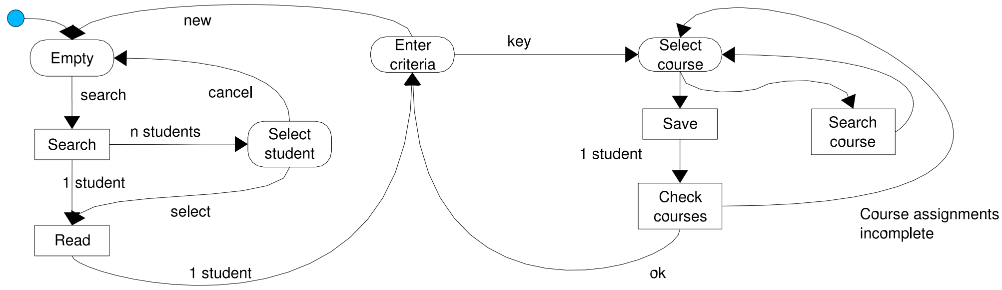


Figure 12.7: Interactivity diagram for dialog Dia_[Assign student to course]

Screen	Description	Fields
Scr_[Enter course]	enter a new course and its characteristics	input: name (ET_[Course], Att_[Name]), begin (ET_[Course], Att_[Begin]), end (ET_[Course], Att_[End]), cadence (ET_[Course], Att_[Cadence]), dow (ET_[Course], Att_[DayOfWeek])
Scr_[Enter classes]	enter new classes for a course	input: date (ET_[Class], Att_[Date]) input: begin (ET_[Class], Att_[Begin]) input: end (ET_[Class], Att_[End])
Scr_[Error course]	show error message	output: error
Scr_[Error class]	show error message	output: error

Actions:

Action	Description	Analysis Functions
A_[Check course]	check whether course already in database	-
A_[Check classes]	check whether each class date is in range	-
A_[Save]	save new course	-

IAD: see Fig. 12.6.

Dialogs in the Component Com_[Course attendees]

Name: Dia_[Assign student to course]
Description: The dialog is used to assign a student to a course.
Use case: UC_[Assign students to course]
Screens:

Screen	Description	Fields
Scr_[Empty]	enter search criteria for students	input: name (ET_[Student], Att_[Name])
Scr_[Select student]	select a student from a list	combo box: students (ET_[Student], Att_[Name], Att_[Age], Att_[Address])
Scr_[Enter criteria]	enter search criteria for course	input: begin (ET_[Course], Att_[Begin]), end (ET_[Course], Att_[End])
Scr_[Select course]	select a course for the student	combo box: courses (ET_[Course])

Actions:

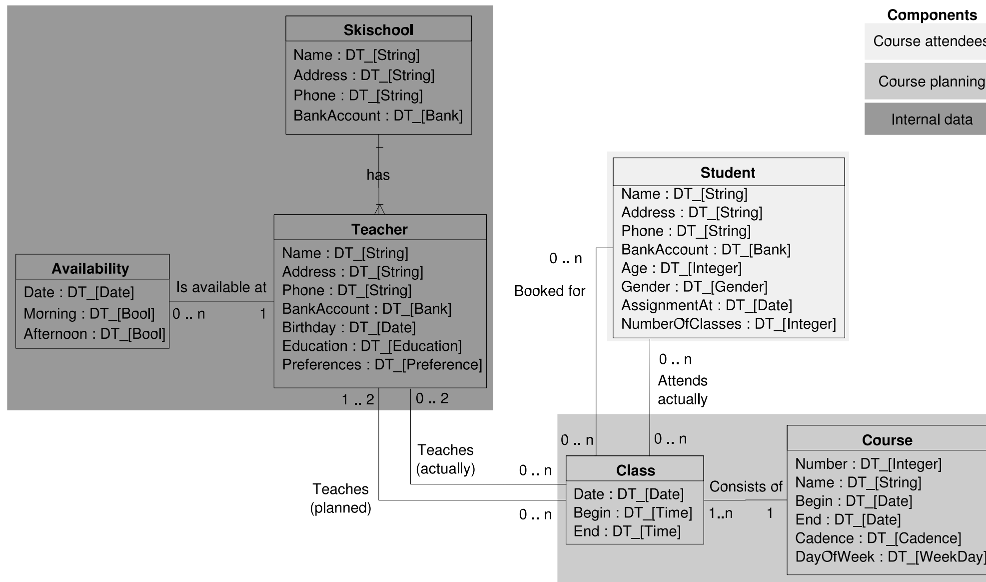


Figure 12.8: Data model

Action	Description	Analysis Functions
A_[Search]	search students	-
A_[Read]	read student record	-
A_[Search course]	search course	-
A_[Save]	save course assignment	-
A_[Check courses]	check whether assignment ok	AF_[Check maximum students for class]

IAD: see Fig. 12.7.

12.1.10 Data Model

The data model shown in Fig. 12.8 is in a preliminary state; it only models the ski school, its teachers, and the assignments of students to classes and courses.

12.1.11 Data Types

The data types document defines technical data types used in the data model (see Tab. 12.5). Basic types require no further definition — we assume they are already available in the programming language used in the construction. For a variant (or enumeration) data type, we list all alternatives, separated by the symbol |. For a record (or structure) data type, we list labels and their types. For a range data type, we list the lower bound and the upper bound, separated by two dots.

12.2 Developing the Specification

For experimenting with the ski school specification, we assume a team of four developers: the project manager Maggie and three software engineers Bob, Elli,

Data Type	Definition	Description
DT_[String]	basic	sequence of characters
DT_[Integer]	basic	integer numbers
DT_[Bool]	True | False	booleans
DT_[Bank]	account : DT_[Integer];	bank account: account number,
	bank : DT_[Integer];	bank code,
	name : DT_[String]	bank name
DT_[Date]	day : DT_[Day]	date: $1 \leq$ day ≤ 31,
	month : DT_[Month]	$1 \leq$ month ≤ 12,
	year : DT_[Year]	$1900 \leq$ year ≤ 3000
DT_[Time]	hour : DT_[Hour]	time: $0 \leq$ hour ≤ 23
	minute : DT_[Minute]	$0 \leq$ minute ≤ 59
DT_[Day]	1 .. 31	day of month
DT_[Month]	1 .. 12	month of year
DT_[Year]	1900 .. 3000	year
DT_[Hour]	0 .. 23	hour of day
DT_[Minute]	0 .. 59	minute of hour
DT_[Preference]	Children | Adults	preferences of a teacher
DT_[Education]	Graduated | None	education of a teacher
DT_[Cadence]	Daily | Weekly	cadence of a course
DT_[WeekDay]	Mo | Tu | We | Th | Fr | Sa | Su	day of week
DT_[Gender]	Male | Female	gender

Table 12.5: Data types

and Peter. We use the revision control system DARCS [Rou04], where each developer is responsible for some part of the specification (see Tab. 12.6). Notice that Bob, Elli, and Peter share the documents AnaFun.xml and Dialogs.xml. DARCS takes care that changes to these documents are mergeable. Below, we summarize the change log of the DARCS repository. In Sect. 12.3 and 12.4, we discuss some of the inconsistencies occurred during the development.

1. Check-in by Maggie:
   new files: ReadingInst.xml, GoalsScope.xml
   comments: ReadingInst.xml: added reader species user, designer, test team, and management
   GoalsScope.xml: added actors 'Planning staff ski school' and 'Office staff ski school'

2. Check-in by Maggie:
   new files: Glossary.xml, CrossCutting.xml
   changed files: ReadingInst.xml
   comments: Glossary.xml: added definitions for course, class, teacher, and student
   CrossCutting.xml: added naming conventions for actors, business processes, activities, components, use cases, and analysis functions
   ReadingInst.xml: added reading instructions for business processes, use cases, and the glossary

3. Check-in by Bob:
   new files: BP_Plan_Classes.xml, UC_Plan_Classes.xml

Person	Responsibilities	Owned Documents
Maggie	specification documents	SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml
	goals, scope, constraints	GoalsScope.xml
	glossary	Glossary.xml
	reading instructions	ReadingInst.xml
	cross-cutting Concerns	CrossCutting.xml
	data model	DataModel.xml
	data types	DataTypes.xml
Bob	business process BP_[Plan classes]	BP_Plan_Classes.xml
	associated use cases	UC_Plan_Classes.xml
	associated analysis functions	AnaFun.xml
	associated dialogs	Dialogs.xml
Elli	business process BP_[Assign teacher to class]	BP_Assign_Teacher.xml
	associated use cases	UC_Assign_Teacher.xml
	associated analysis functions	AnaFun.xml
Peter	business process BP_[Register students]	BP_Register_Students.xml
	associated use cases	UC_Register_Students.xml
	associated analysis functions	AnaFun.xml
	associated dialogs	Dialogs.xml

Table 12.6: Responsibilities of the developers

comments: BP_Plan_Classes.xml: added business process BP_[Plan classes] including activities Act_[Determine class utilization] and Act_[Create new course and classes]
UC_Plan_Classes.xml: added use cases 'Create new course and classes' and 'Determine additional requests and abundances' (without component)

4. Check-in by Elli:
new files: BP_Assign_Teacher.xml, UC_Assign_Teacher.xml
comments: BP_Assign_Teacher.xml: added business process BP_[Assign teacher to classes] including activities Act_[Suggest teacher for course], Act_[Assign teacher to all classes of the course], and Act_[Select the teacher in question and assign him to the new course]
UC_Assign_Teacher.xml: added use cases UC_[Suggest teacher for course] and UC_[Assign teacher to classes] (without component)

5. Check-in by Peter:
new files: BP_Register_Students.xml, UC_Register_Students.xml
comments: BP_Register_Students.xml: added business process BP_ _[Register students] including activities Act_[Create course summary], Act_[Assign student to all classes of the course], Act_[Assign student to some classes of the course], and Act_[Assign student to some classes of either course]

UC_Register_Students.xml: added use cases UC_[Assign students to course], UC_[Create course summary], UC_[Register customer] (without component)

6. Check-in by Maggie:
   new files: SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml
   changed files: Glossary.xml, ReadingInst.xml
   comments: initial coarse specification documents (in progress)
   Glossary.xml: give more precise descriptions for course, class, and teacher; set status to 'finished'
   ReadingInst.xml: added reading instructions for analysis functions, data model, data types, dialogs, and cross-cutting concerns; set status to 'finished'

7. Check-in by Bob:
   changed files: BP_Plan_Classes.xml, UC_Plan_Classes.xml
   comments: BP_Plan_Classes.xml: added activities Act_[Assign students to class] and Act_[Move students to other class]
   UC_Plan_Classes.xml: changed name of use case 'Create new course and classes' to UC_[Create new course and classes], changed name of use case 'Determine additional requests and abundances' to UC_[Determine additional requests and abundances], added use cases UC_[Change assignments of students to classes] and UC_[Assign students to course]

8. Check-in by Peter:
   changed files: BP_Register_Students.xml, UC_Register_Students.xml
   comments: BP_Register_Students.xml: added activities Act_[Register customer (4)] and Act_[Set customer to wish list]
   UC_Register_Students.xml: added use case UC_[Set student to wish list]

9. Check-in by Maggie:
   changed files: SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml, ReadingInst.xml
   comments: set status of specification documents to 'quality assurance,' reference documents for business processes and use cases
   ReadingInst.xml: added reader species 'management' to the cross-cutting concerns

10. Check-in by Bob:
    changed files: BP_Plan_Classes.xml, UC_Plan_Classes.xml
    comments: BP_Plan_Classes.xml: added input/output to all activities, added activities Act_[Merge classes], Act_[Delete associated teacher for a class], and Act_[Delete class]
    UC_Plan_Classes.xml: added use case UC_[Delete course], deleted use case UC_[Assign students to course] (use case was duplicated)
    @Peter: add a reference to Act_[Assign students to class] (BP_[Plan classes]) to your UC_[Assign students to course]

11. Check-in by Peter:
    changed files: BP_Register_Students.xml, UC_Register_Students.xml
    comments: BP_Register_Students.xml: added input/output to all activities
    UC_Register_Students.xml: added reference to Act_[Assign students to class] (BP_[Plan classes]) to use case UC_[Assign students to course]

12. Check-in by Elli:
    changed files: BP_Assign_Teacher.xml
    comments: added input/output to all activities

13. Check-in by Maggie:
    changed files: SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml
    comments: all specification documents in status 'finished;' coarse specification finished (milestone 1)

14. Check-in by Maggie:
    new files: DataModel.xml, DataTypes.xml
    changed files: CrossCutting.xml
    comments: CrossCutting.xml: added naming conventions for entity types, attributes, relations, data types, fields in types, dialogs, and screens in dialogs
    DataModel.xml: added component Com_[Internal data] containing entity types ET_[Skischool], ET_[Teacher], and ET_[Availability]; added component Com_[Course planning] containing entity types ET_[Class] and ET_[Course]; added component Com_[Course attendees] containing entity type ET_[Student]
    DataTypes.xml: added data types DT_[String], DT_[Integer], DT_[Bool], and DT_[Year]
    @all: please use my components

15. Check-in by Bob:
    new files: AnaFun.xml, Dialogs.xml
    changed files: UC_Plan_Classes.xml
    comments: AnaFun.xml: added component Com_[Course planning] including analysis functions AF_[Create course], AF_[Create classes for a course], AF_[Remove assignment of teacher to class], AF_[Remove class], and AF_[Remove course]
    Dialogs.xml: added component Com_[Course planning] containing the dialog Dia_[New course and classes]
    UC_Plan_Classes.xml: moved use cases to component Com_[Course planning], added pre- and postconditions to each use case, reference analysis functions

16. Check-in by Elli:
    changed files: AnaFun.xml, UC_Assign_Teacher.xml

comments: AnaFun.xml: added component Com_[Internal data] including analysis functions AF_[Filter teachers], AF_[Check availability], and AF_[Assign teacher to class]
UC_Assign_Teacher.xml: moved use cases to component Com_[Internal data], added pre- and postconditions to each use case, reference analysis functions

17. Check-in by Peter:
    changed files: AnaFun.xml, Dialogs.xml, UC_Register_Students.xml
    comments: AnaFun.xml: added component Com_[Course attendees] including analysis functions AF_[Check maximum students for class], AF_[Find classes for student by date], AF_[Filter courses], and AF_[Add customer]
    Dialogs.xml: added component Com_[Course attendees] including the dialog Dia_[Assign student to course]
    UC_Register_Students.xml: moved use case UC_[Create course summary] to component Com_[Internal data], moved other use cases to component Com_[Course attendees], added pre- and postconditions to each use case, reference analysis functions

18. Check-in by Maggie:
    changed files: DataModel.xml, DataTypes.xml, SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml
    comments: DataModel.xml: added some attributes to entity types
    DataTypes.xml: corrected upper bound in data type DT_[Year], added data types DT_[Bank], DT_[Date], DT_[Time], DT_[Day], DT_[Month], DT_[Hour], DT_[Minute], DT_[Preference], DT_[Education], DT_[Cadence], DT_[Gender], and DT_[Weekday]
    specification documents: changed kind to 'fine,' set status to 'in progress,' reference new documents

19. Check-in by Bob:
    changed files: BP_Plan_Classes.xml, UC_Plan_Classes.xml, AnaFun.xml, Dialogs.xml
    comments: BP_Plan_Classes.xml: reference entity types (instead of using natural language), set status to 'quality assurance'
    UC_Plan_Classes.xml: reference entity types for inputs and outputs, set status to 'quality assurance'
    AnaFun.xml: in component Com_[Course planning] make references to entity types in parameters
    Dialogs.xml: in dialog Dia_[New course and classes]: make references to attributes not only to entities, add actions, add IAD

20. Check-in by Elli:
    changed files: UC_Assign_Teacher.xml, BP_Assign_Teacher.xml, AnaFun.xml
    comments: AnaFun.xml: in component Com_[Internal data] make references to entity types in parameters

UC_Assign_Teacher.xml: reference entity types for inputs and outputs, set status to 'quality assurance'
BP_Assign_Teacher.xml: reference entity types (instead of using natural language), set status to 'quality assurance'

21. Check-in by Peter:
changed files: BP_Register_Students.xml, UC_Register_Students.xml, AnaFun.xml, Dialogs.xml
comments: BP_Register_Students.xml: reference entity types (instead of using natural language), remove activities Act_[Set customer to wish list] and Act_[Register customer (4)] (wish list support dropped because too expensive), set status to 'quality assurance'
UC_Register_Students.xml: reference entity types for inputs and outputs, in use case UC_[Assign students to course] reference analysis function AF _[Check maximum students for class] (because the dialog Dia_[Assign students to course] does), remove use case UC_[Set student to wish list], move use case UC_[Create course summary] to component Com_[Course attendees], set status to 'quality assurance'
AnaFun.xml: in component Com_[Course attendees] make references to entity types in parameters, set status to 'quality assurance'
Dialogs.xml: in dialog Dia_[Assign students to course]: make references to attributes not only to entity types, add actions, add IAD, set status to 'quality assurance'

22. Check-in by Maggie:
changed files: DataModel.xml, SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml
comments: DataModel.xml: add cardinalities to relations
specification documents: set status to 'quality assurance'

23. Check-in by Maggie:
changed files: SpecUser.xml, SpecDesigner.xml, SpecManagement.xml, SpecTest.xml
comments: set status to 'finished'; fine specification finished (milestone 2)

24. Check-in by Maggie:
changed files: DataModel.xml
comments: Added target cardinality for relation 'Is available at' (which I forgot earlier), set status to 'finished'.

12.3 Sample S-DAGs

The careful reader may have noticed that our example specification is inherently inconsistent. Our system reports an overall number of 493 inconsistencies. Tab. 12.7 summarizes inconsistencies for the individual rules. Rule 2 cannot be violated, because it is strong; all other rules are weak. Next, we present

Rule	Violated at States	Number of Inconsistencies
Rule 2	-	0
Rule 3	1	2
Rule 4	18 . . . 24	140
Rule 19	3 . . . 9	14
Rule 21	15 . . . 24	10
Rule 30	16 . . . 24	42
Rule 41	23	6
Rule 48	14 . . . 17	4
Rule 64	17 . . . 20	8
Rule 70	6 . . . 24	152
Rule 73	17 . . . 20	4
Rule 74	14 . . . 24	22
Rule 80	23, 24	80
Rule 82	6 . . . 8	6
Rule 84	6, 14	3

Table 12.7: Inconsistency summary

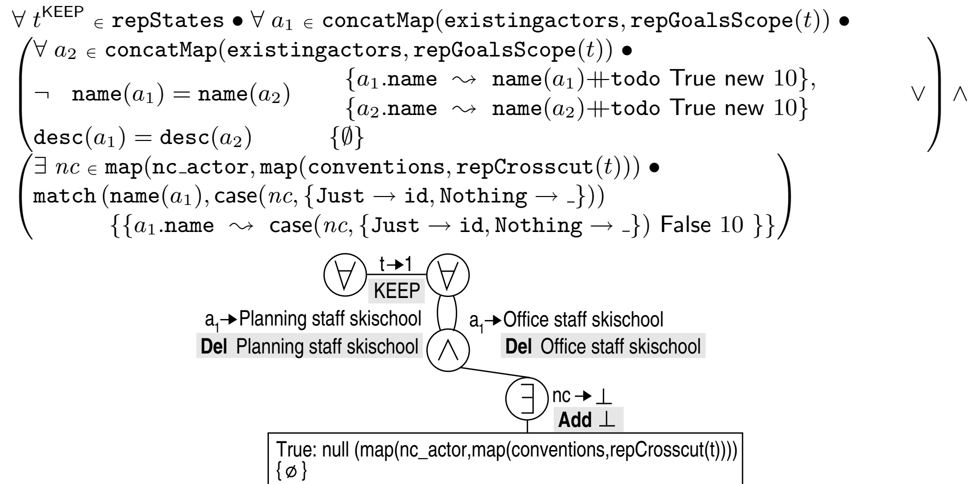


Figure 12.9: Augmented S-DAG for Rule 3 at state 1

some augmented S-DAGs. For convenience, we also show the corresponding consistency rules in their miniscoped form — they resemble the structure of S-DAGs. Recall that miniscoping replaces implications by disjunctions and pushes negations and quantifiers into formulae.

At state 1, we generate the S-DAG shown in Fig. 12.9 for Rule 3. The S-DAG shows that there does not yet exist a naming convention for authors in the cross-cutting concerns. We can resolve inconsistencies by either deleting the actors or adding a naming convention for actors. It is, however, unknown how this naming convention should look like, denoted by $\perp$. Maggie adds a naming convention at state 2. The predicate leaf is generated, because the sphere of the existential quantifier for the variable *nc* is empty.

At state 23, we generate the S-DAG shown in Fig. 12.10 for Rule 4. The S-DAG indicates that each fine specification is lacking a style guide. We can

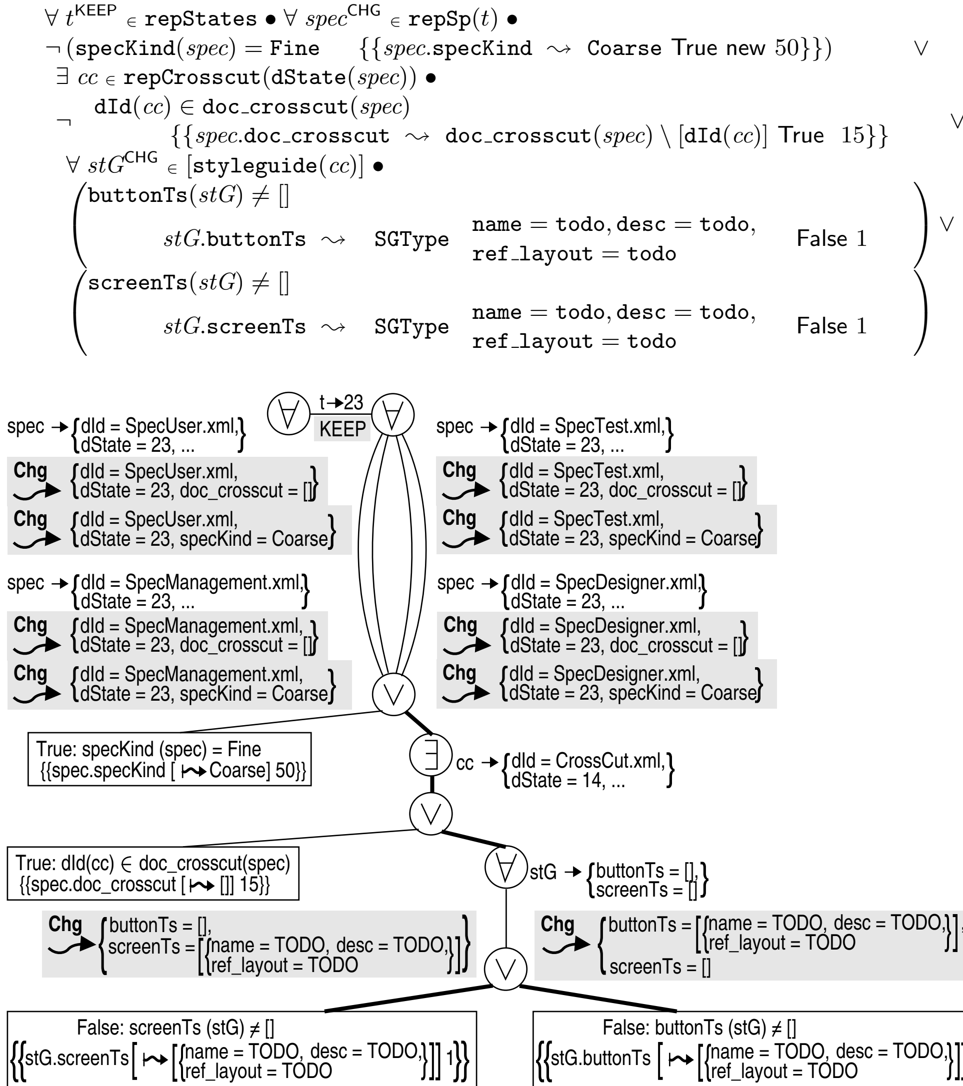


Figure 12.10: Augmented S-DAG for Rule 4 at state 23

resolve these inconsistencies by either adding a style guide, or "downgrading" the specifications, or removing the cross-cutting concerns. Adding the style guide is considered the cheapest solution. Below disjunction and existential nodes, respectively, we mark edges bold that lead to cheap repair actions.

At state 6, we generate the S-DAG shown in Fig. 12.11 for Rule 19. From the S-DAG, we see that the names of the use cases 'Create new course and classes' and 'Determine additional requests and abundances' do not match their naming convention, determined by the regular expression UC_[*]. Our system proposes to change the names to the naming convention. At state 7, Bob corrects the use case names.

At state 17, we generate the S-DAG shown in Fig. 12.12 for Rule 21. The S-DAG indicates that the preconditions of the use case UC_[Create new course and classes] in the component Com_[Course planning] cannot be fulfilled. In

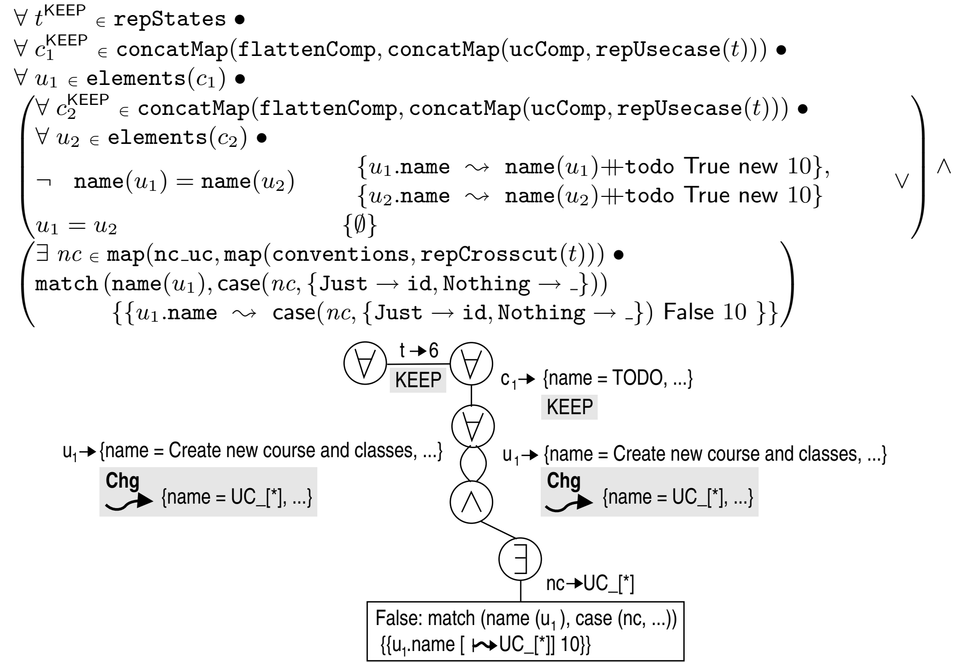


Figure 12.11: Augmented S-DAG for Rule 19 at state 6

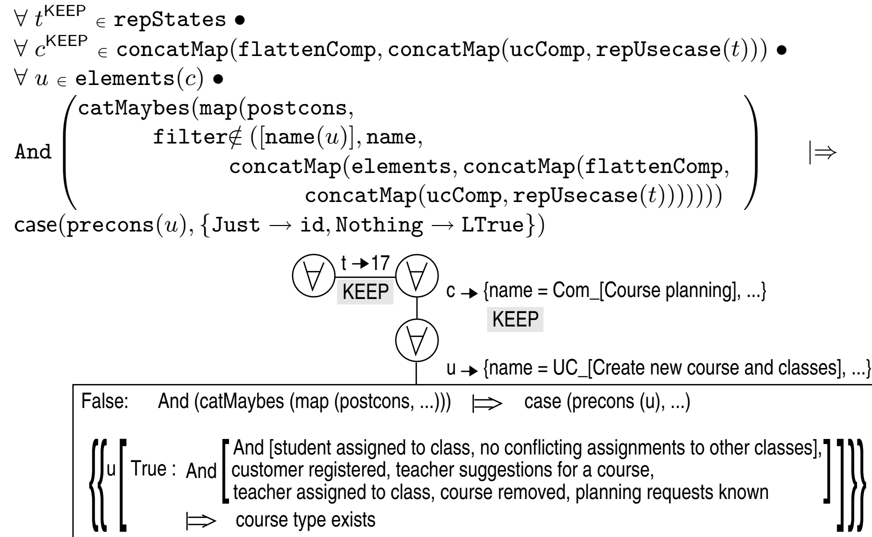


Figure 12.12: Augmented S-DAG for Rule 21 at state 17

the predicate leaf, we find the postconditions of all other use cases as first argument of the predicate symbol $|\Rightarrow$.

At state 17, we generate the S-DAG shown in Fig. 12.13 for Rule 30. From the S-DAG, we see that Rule 30 is violated for the use cases UC_[Set student to wish list], UC_[Suggest teacher for course], and UC_[Assign students to course]. The reasons for these inconsistencies are that the above use cases are annotated

$$\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \\
\forall\, u^{\mathsf{CHG}} \in \mathtt{concatMap(elements, concatMap(flattenComp,} \bullet \\
\qquad \mathtt{concatMap(ucComp, repUsecase(}t\mathtt{))))} \\
\mathtt{null(ref_fun(}u\mathtt{))} \quad \{\emptyset\} \qquad \vee \\
\mathtt{And} \left(\begin{array}{l} \mathtt{catMaybes(map(postcons,} \\ \qquad \mathtt{filter}\in \mathtt{(ref_fun(}u\mathtt{), name,} \\ \qquad\qquad \mathtt{concatMap(elements, concatMap(flattenComp,} \\ \qquad\qquad\qquad \mathtt{concatMap(afComp, repAnaFun(}t\mathtt{)))))))} \end{array} \right) \quad \models\Rightarrow \\
\mathtt{case(precons(}u\mathtt{), \{Just \rightarrow id, Nothing \rightarrow LTrue\})}
\end{array}$$

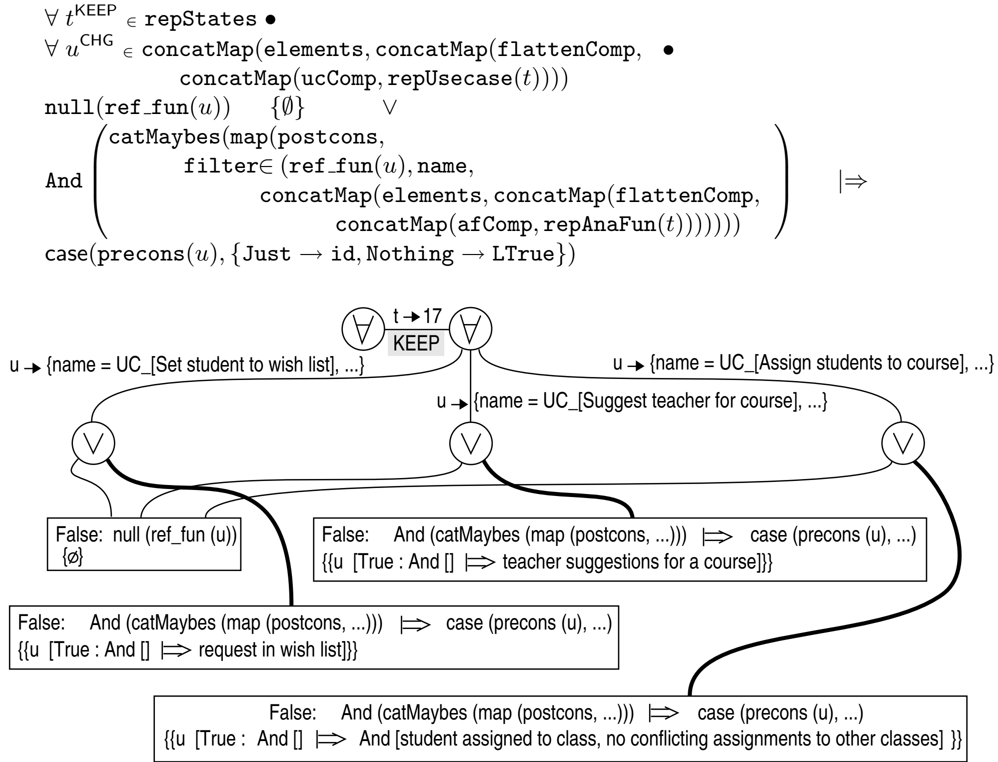


Figure 12.13: Augmented S-DAG for Rule 30 at state 17

by postconditions but they only use analysis functions lacking postconditions. Obviously, it remains unclear how we can repair these inconsistencies.

At state 23, we generate the S-DAG shown in Fig. 12.14 for Rule 41. Basically, the S-DAG indicates that the association 'is available at' of the entity type ET_[Availability] lacks a target cardinality. The entity type is defined in the document DataModel.xml. The cheapest way to resolve this inconsistency is to add a target cardinality to this association. Alternatively, one could "downgrade" the specification to status 'in progress' or to a coarse specification. Notice that Rule 41 applies to finished fine specifications only.

At state 17, we generate the S-DAG shown in Fig. 12.15 for Rule 48. The S-DAG indicates that the lower bound of the range data type DT_[Year] is not smaller than its upper bound. Maggie corrects this typo at state 18.

At state 17, we generate the S-DAG shown in Fig. 12.16 for Rule 64. From the S-DAG, we see that the use case UC_[Assign students to course] and the dialog Dia_[Assign student to course] violate the rule, because the analysis function AF_[Check maximum students for class] is used by the dialog but not by its associated use case. We can repair this inconsistency by deleting the analysis function from the dialog, cutting the association between dialog and use case, or adding the analysis function AF_[Check maximum students for class] to the use case UC_[Assign students to course]. At state 21, Peter chooses the last alternative. Notice that Rule 64 is changed by miniscoping to great extent. First, the implication is replaced by a disjunction. Then the

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, spec^{\mathsf{CHG}} \in \mathtt{repSp}(t) \bullet \\
\neg\,(\mathtt{specKind}(spec) = \mathtt{Fine} \quad \{\{spec.\mathtt{specKind} \rightsquigarrow \mathtt{Coarse}\ \mathsf{True}\ 50\}\}) \quad \vee \\
\neg\,(\mathtt{status}(spec) = \mathtt{Finished} \quad \{\{spec.\mathtt{status} \rightsquigarrow \mathtt{InProgress}\ \mathsf{True}\ 40\}\}) \quad \vee \\
\quad \forall\, dm^{\mathsf{KEEP}} \in \mathtt{repDataMod}(\mathtt{dState}(spec)) \bullet \\
\quad \neg\ \ \mathtt{dId}(dm) = \mathtt{case}(\mathtt{doc_datamod}(spec), \{\mathtt{Just} \to \mathtt{id}, \mathtt{Nothing} \to _\}) \quad \{\emptyset\} \quad \vee \\
\quad\quad \forall\, et^{\mathsf{KEEP}} \in \mathtt{concatMap}(\mathtt{elements}, \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{compsDM}(dm))) \bullet \\
\quad\quad \forall\, rel^{\mathsf{CHG}} \in \mathtt{relatesWith}(et) \bullet \\
\quad\quad \mathtt{targetcard}(rel) \neq _ \quad \{\{rel.\mathtt{targetcard} \rightsquigarrow \mathtt{todo}\ \mathsf{False}\ 1\}\}
\end{array}
$$

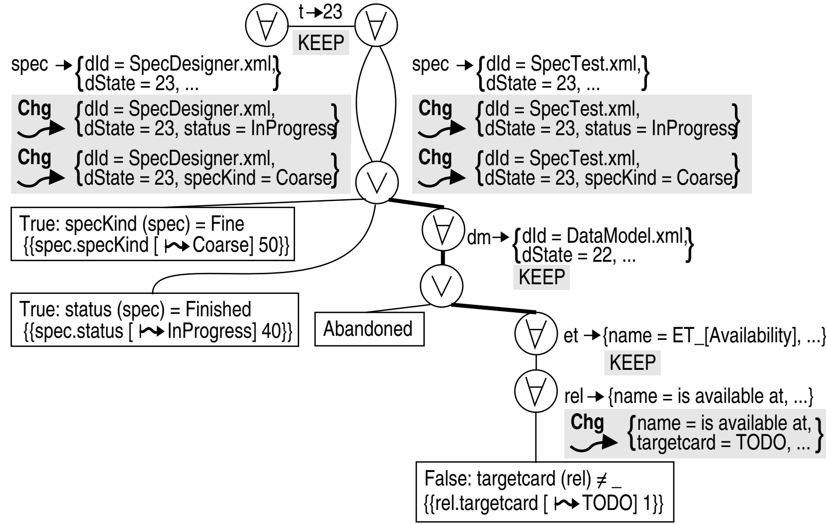


Figure 12.14: Augmented S-DAG for Rule 41 at state 23

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, dt \in \mathtt{concatMap}(\mathtt{dataTs}, \mathtt{repDataTs}(t)) \bullet \\
\mathtt{rngFrom}(\mathtt{kind}(dt)) < \mathtt{rngTo}(\mathtt{kind}(dt))
\end{array}
$$

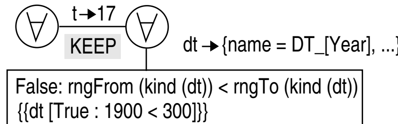


Figure 12.15: Augmented S-DAG for Rule 48 at state 17

universal quantifier for the variable *fun* is pushed into the second subformula of this disjunction. This requires to exchange the universal quantifiers for *fun* and u. Our miniscoping algorithm also has exchanged the universal quantifiers for the variables *dlg* and u in the "hope" to push the quantifier for *dlg* deeper into the formula.

At state 6, our system generates the S-DAG shown in Fig. 12.17 for Rule 70. The glossary terms Teacher and Class have been re-defined. These definitions are not similar to the previous definitions present at the states 2 through 5. Our system proposes to roll-back these changes.

$$\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \\
\forall\, u^{\mathsf{CHG}} \in \mathtt{concatMap}(\mathtt{elements}, \mathtt{concatMap}(\mathtt{flattenComp}, \bullet \\
\qquad \mathtt{concatMap}(\mathtt{ucComp}, \mathtt{repUsecase}(t)))) \\
\forall\, dlg^{\mathsf{CHG}} \in \mathtt{concatMap}(\mathtt{elements}, \mathtt{concatMap}(\mathtt{flattenComp}, \bullet \\
\qquad \mathtt{concatMap}(\mathtt{diaComp}, \mathtt{repDialog}(t)))) \\
\neg\ \mathtt{name}(u) \in \mathtt{usecases}(dlg)\ \ \{\{dlg.\mathtt{usecases} \leadsto \mathtt{usecases}(dlg) \setminus [\mathtt{name}(u)]\ \mathsf{True}\ 3\ \}\} \quad \vee \\
\forall\, fun \in \mathtt{concatMap}(\mathtt{actFun}, \mathtt{actions}(dlg)) \bullet \\
fun \in \mathtt{ref_fun}(u) \qquad \{\{u.\mathtt{ref_fun} \leadsto fun : (\mathtt{ref_fun}(u))\ \mathsf{False}\ 1\ \}\}
\end{array}$$

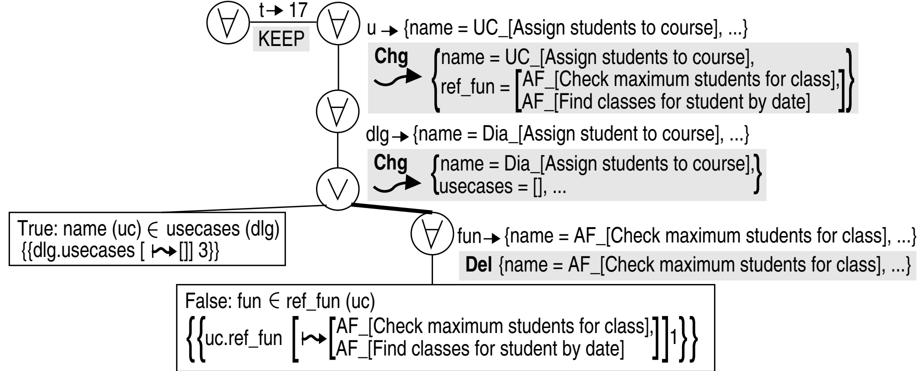


Figure 12.16: Augmented S-DAG for Rule 64 at state 17

$$\begin{array}{l}
\forall\, t_n^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, t_o^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \\
\neg(t_o < t_n) \vee \left(\begin{array}{l}
\forall\, g_o^{\mathsf{KEEP}} \in \mathtt{repGlossary}(t_o) \bullet \forall\, term_o^{\mathsf{KEEP}} \in \mathtt{terms}(g_o) \bullet \\
\exists\, g_n \in \mathtt{repGlossary}(t_n) \bullet \exists\, term_n \in \mathtt{terms}(g_n) \bullet \\
\mathtt{name}(term_o) = \mathtt{name}(term_n)\ \ \{\{term_n.\mathtt{name} \leadsto \mathtt{name}(term_o)\ \mathsf{False}\ 5\ \}\} \quad \wedge \\
\mathtt{desc}(term_o) \equiv \mathtt{desc}(term_n)\ \ \{\{term_n.\mathtt{desc} \leadsto \mathtt{desc}(term_o)\ \mathsf{False}\ 2\ \}\}
\end{array}\right)
\end{array}$$

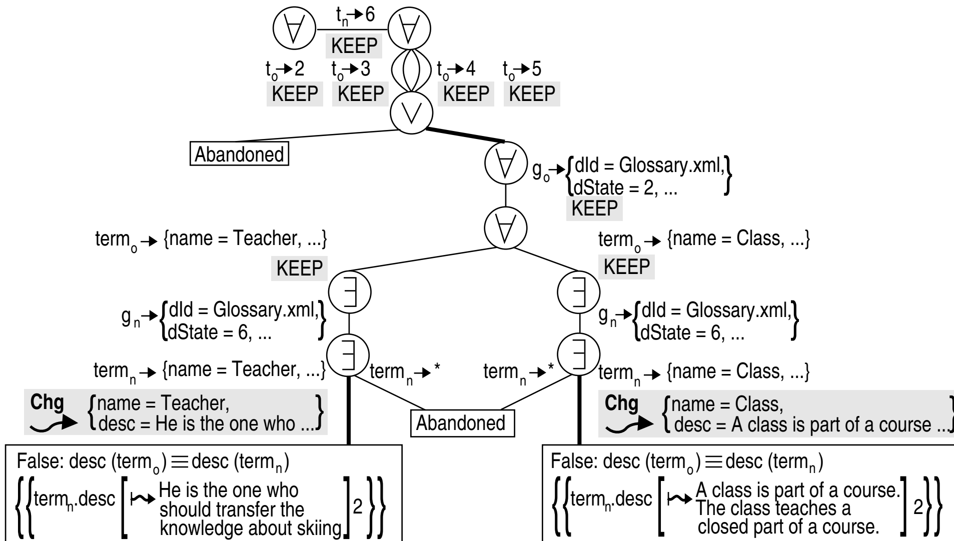


Figure 12.17: Augmented S-DAG for Rule 70 at state 6

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, uD^{\mathsf{CHG}} \in \mathtt{repUsecase}(t) \bullet \\
\forall\, c^{\mathsf{CHG}} \in \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{ucComp}(uD)) \bullet \forall\, u^{\mathsf{CHG}} \in \mathtt{elements}(c) \bullet \\
\forall\, f_{ref} \in \mathtt{ref_fun}(u) \bullet \\
\exists\, c_{fun} \in \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{concatMap}(\mathtt{afComp}, \mathtt{repAnaFun}(t))) \bullet \\
\mathtt{name}(c) = \mathtt{name}(c_{fun}) \quad \{\{c_{fun}.\mathtt{name} \leadsto \mathtt{name}(c)\ \mathsf{False}\ 10\}\} \quad \wedge \\
\exists\, f \in \mathtt{elements}(c_{fun}) \bullet \\
f_{ref} = \mathtt{name}(f) \quad \{\{f_{ref} \leadsto \mathtt{name}(f)\ \mathsf{False}\ 1\}\}
\end{array}
$$

t → 17
KEEP
uD → {dId = UC_Register_Students.xml, dState = 17, ...}
c → {name = Com_[Internal data], ...}
u → {name = UC_[Create course summary], ...}
f_{ref} → {name = AF_[Filter courses], ...}
Chg {name = AF_[Check availability], ...}
Chg {name = AF_[Filter teachers], ...}
Chg {name = AF_[Assign teacher to class], ...}
c_{fun} → {name = Com_[Course attendees], ...}
Add {name = Com_[Internal data], ...}
c_{fun} → *
Abandoned
c_{fun} → {name = Com_[Internal data], ...}
False: name (c) = name (c_{fun})
{{c_{fun}.name [↦ Com_[Internal data]] 10}}
f → {name = AF_[Assign teacher to class], ...}
f → {name = AF_[Check availability], ...}
f → {name = AF_[Filter teachers], ...}
False: f_{ref} = name (f)
{{f_{ref} [↦ AF_[Assign teacher to class]] 1}}
False: f_{ref} = name (f)
{{f_{ref} [↦ AF_[Check availability]] 1}}
False: f_{ref} = name (f)
{{f_{ref} [↦ AF_[Filter teachers]] 1}}

Figure 12.18: Augmented S-DAG for Rule 73 at state 17

$$
\begin{array}{l}
\forall\, t^{\mathsf{KEEP}} \in \mathtt{repStates} \bullet \forall\, dm^{\mathsf{KEEP}} \in \mathtt{repDataMod}(t) \bullet \\
\forall\, c^{\mathsf{CHG}} \in \mathtt{concatMap}(\mathtt{flattenComp}, \mathtt{compsDM}(dm)) \bullet \forall\, et^{\mathsf{CHG}} \in \mathtt{elements}(c) \bullet \\
\mathtt{length}(\mathtt{relatedWith}(et)) \not>_{\mathtt{Int}} 2 \quad \leq \\
\mathtt{length}(\mathtt{filter}{\in}\ (\mathtt{map}(\mathtt{etName}, \mathtt{map}(\mathtt{ref_entitytype}, \mathtt{relatesWith}(et))), \\
\qquad\qquad \mathtt{name}, \mathtt{elements}(c)))
\end{array}
$$

t → 17
KEEP
dm → {dId = DataModel.xml, dState = 14, ...}
KEEP
c → {name = Com_[Course attendees], ...}
c → {name = Com_[Course planning], ...}
et → {name = ET_[Student], ...}
et → {name = ET_[Class], ...}
False: length (relatedWith (et)) $\not>_{Int}$ 2 ⩽ length (filter∈ (...))
{{et [True : 2 ⩽ 1], c [True : 2 ⩽ 1]}}
False: length (relatedWith (et)) $\not>_{Int}$ 2 ⩽ length (filter∈ (...))
{{et [True : 1 ⩽ 0], c [True : 1 ⩽ 0]}}

Figure 12.19: Augmented S-DAG for Rule 74 at state 17

At state 17, our system generates the S-DAG shown in Fig. 12.18 for Rule 73. The S-DAG shows that the use case UC_[Create course summary] violates Rule 73, because the used analysis function AF_[Filter courses] is not a member of the use case's component Com_[Internal data]. The S-DAG also indicates that the analysis function AF_[Filter courses] is defined in the component Com_[Course attendees]. Our system proposes to add a component Com_[Internal data] or to change the analysis function reference to one of the

analysis functions in the component Com_[Internal data]. Any of these suggestions would resolve the inconsistency. At state 21, Peter moves the use case UC_[Create course summary] to the component Com_[Course attendees]. The actions in the S-DAG do not include this possibility, because the rule designer did not propose to change the use case component.

At state 17, our system generates the S-DAG shown in Fig. 12.19 for Rule 74. From the S-DAG, we see that the entity types ET_[Student] and ET_[Class] have more relations to entity types included in a different component than to entity types within their own component. Of course, these inconsistencies cannot be resolved automatically. In order to fulfill Rule 74, a complete re-design of the components seems appropriate. But since this rule only reflects a property of a good data model, these minor inconsistencies are tolerated throughout.

At state 23, our system generates the S-DAG shown in Fig. 12.20 for Rule 80. The S-DAG shows that the finished fine specifications for the user and the designer, respectively, violate the rule. For both fine specifications, we have a matching coarse specification in states between 13 and 17. The rule is violated, because wish list support has been dropped in the fine specification: At state 23, the business process BP_[Register students] lacks the activities Act_[Set customer to wish list] and Act_[Register customer (4)]. Our system proposes to add the missing activities, or downgrade the specifications to status 'in progress' or to coarse specifications. Clearly, dropping wish list support is an intentional design decision that shows up as an inconsistency. By the S-DAG in Fig. 12.20 our approach documents this design decision.

At state 6, our system generates the S-DAG shown in Fig. 12.21 for Rule 82. The S-DAG indicates that the specification for the management contains cross-cutting concerns; the reading instructions, however, fail to indicate that the management should read cross-cutting concerns. Our system proposes either to add an appropriate reading instruction or to remove the cross-cutting concerns from the specification. At state 9, Maggie adds a reading instruction for the management.

At state 6, our system generates the S-DAG shown in Fig. 12.22 for Rule 84. The rule is violated by the documents ReadingInst.xml and Glossary.xml, each of which changed its status from 'in progress' to 'finished.' Our system proposes to roll back their status values to 'in progress.'

$$
\begin{array}{l}
\forall\ t_c^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \forall\ t_f^{\mathsf{KEEP}} \in \texttt{repStates} \bullet \neg(t_c < t_f) \quad \vee \\
\forall\ s_f^{\mathsf{CHG}} \in \texttt{repSp}(t_f) \bullet \\
\neg\,(\texttt{specKind}(s_f) = \texttt{Fine} \quad \{\{s_f.\texttt{specKind} \rightsquigarrow \texttt{Coarse}\ \mathsf{True}\ 50\}\}) \quad \vee \\
\neg\,(\texttt{status}(s_f) = \texttt{Finished} \quad \{\{s_f.\texttt{status} \rightsquigarrow \texttt{InProgress}\ \mathsf{True}\ 40\}\}) \quad \vee \\
\quad \forall\ s_c^{\mathsf{KEEP}} \in \texttt{filter}\in([\texttt{dId}(s_f)], \texttt{dId}, \texttt{repSp}(t_c)) \bullet \\
\quad \neg(\texttt{specKind}(s_c) = \texttt{Coarse}) \quad \vee \quad \neg(\texttt{status}(s_c) = \texttt{Finished}) \quad \vee \\
\quad\quad \forall\ bpD_c^{\mathsf{KEEP}} \in \texttt{filter}\in(\texttt{doc_busproc}(s_c), \texttt{dId}, \texttt{repBusProc}(\texttt{dState}(s_c))) \bullet \\
\quad\quad \forall\ bp_c^{\mathsf{KEEP}} \in \texttt{processes}(bpD_c) \bullet \\
\quad\quad\quad \exists\ bpD_f \in \texttt{filter}\in(\texttt{doc_busproc}(s_f), \texttt{dId}, \texttt{repBusProc}(\texttt{dState}(s_f))) \bullet \\
\quad\quad\quad \exists\ bp_f \in \texttt{processes}(bpD_f) \bullet \\
\quad\quad\quad\quad \texttt{name}(bp_f) = \texttt{name}(bp_c) \quad \{\{bp_f.\texttt{name} \rightsquigarrow \texttt{name}(bp_c)\ \mathsf{False}\ 4\ \}\} \quad \wedge \\
\quad\quad\quad\quad\quad \forall\ act_c^{\mathsf{KEEP}} \in \texttt{activities}(bp_c) \bullet \exists\ act_f \in \texttt{activities}(bp_f) \bullet \\
\quad\quad\quad\quad\quad \texttt{name}(act_f) = \texttt{name}(act_c) \quad \{\{act_f.\texttt{name} \rightsquigarrow \texttt{name}(act_c)\ \mathsf{False}\ 2\ \}\} \quad \wedge \\
\quad\quad\quad\quad\quad \texttt{systems}(act_f) = \texttt{systems}(act_c) \quad \{\{act_f.\texttt{systems} \rightsquigarrow \texttt{systems}(act_c)\ \mathsf{False}\ 1\ \}\}
\end{array}
$$

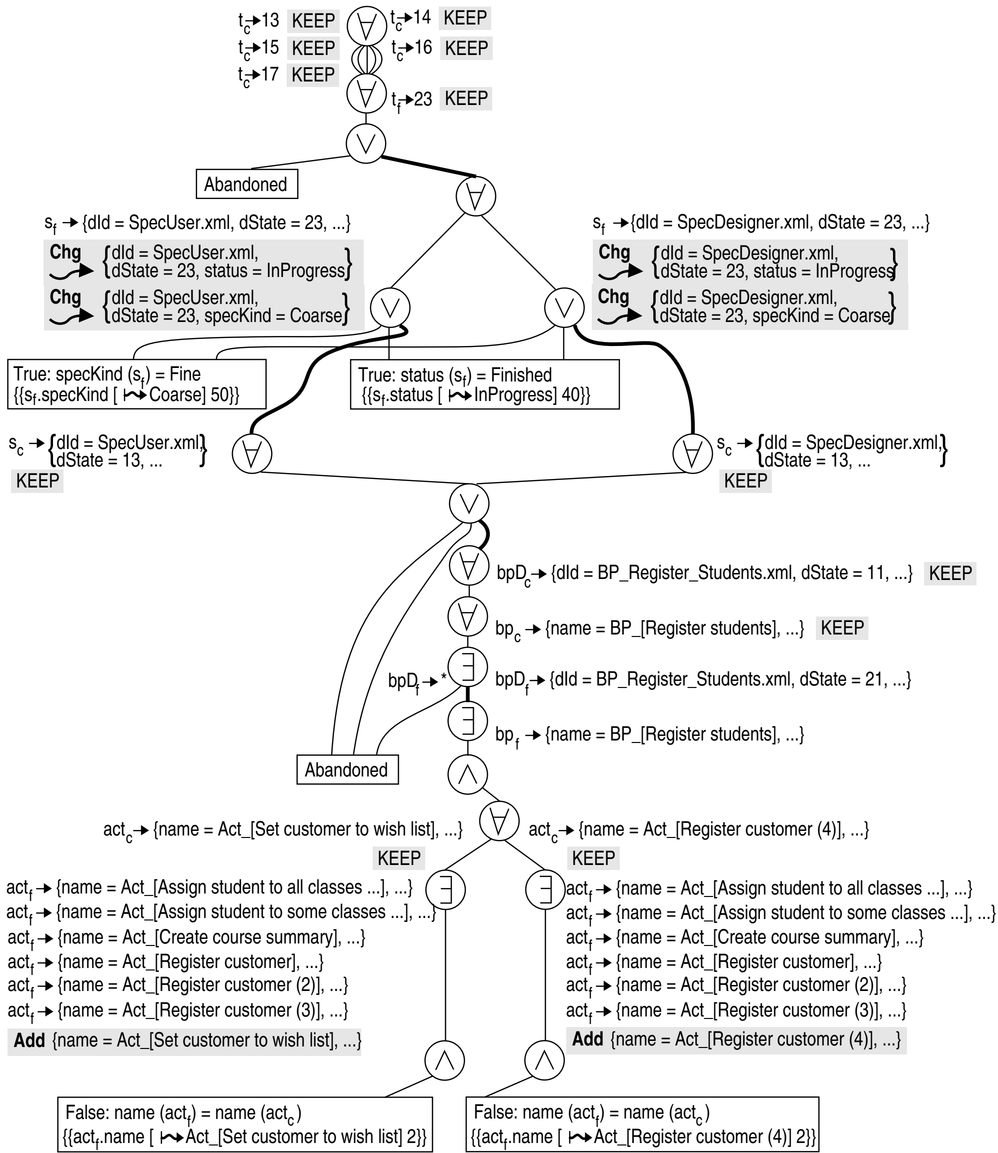


Figure 12.20: Augmented S-DAG for Rule 80 at state 23

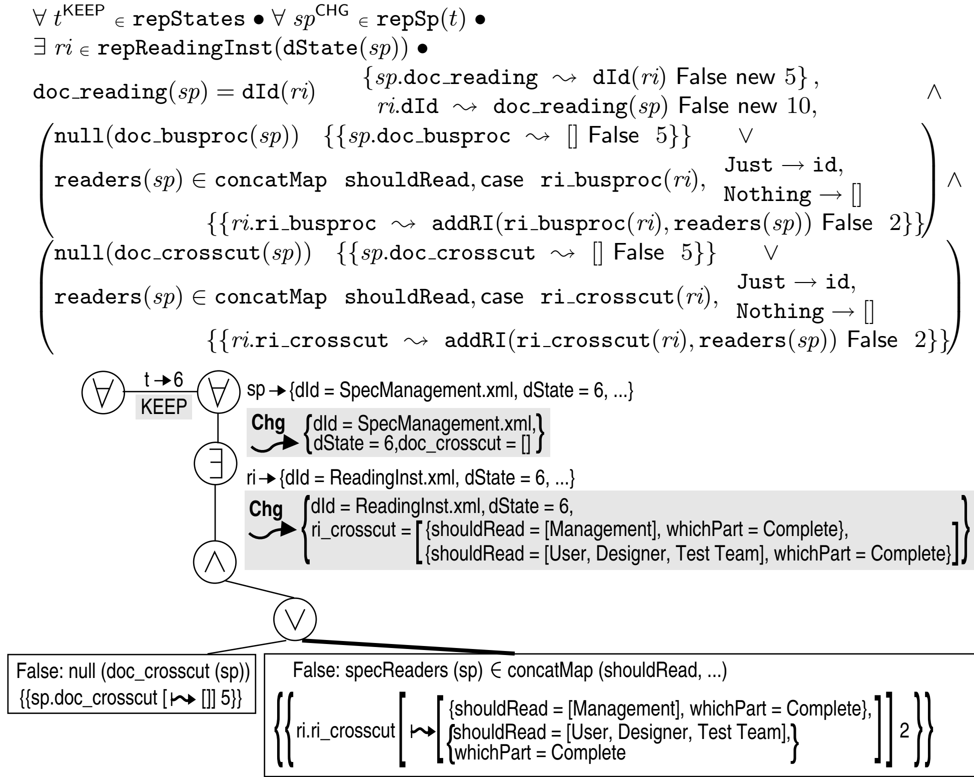


Figure 12.21: Augmented S-DAG for Rule 82 at state 6

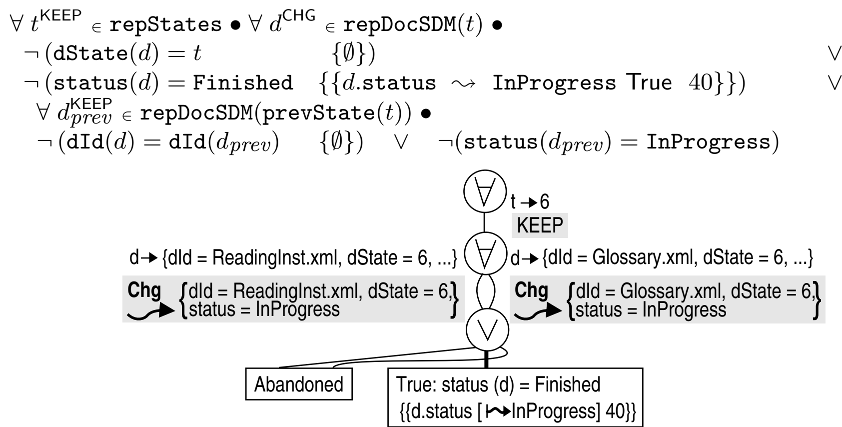


Figure 12.22: Augmented S-DAG for Rule 84 at state 6

Rep $\{19(u_1)\}$ elements(c_1) $\{t \mapsto 6, c_1 \mapsto \{$name = TODO, ...$\}\}$
Chg {name = Create new course and classes, ...}.name $\rightsquigarrow$ UC_[*]
Rate {1 (high)} $\{19, 21, 30, 64, 73, 84\}$ 10

Rep $\{19(u_1)\}$ elements(c_1) $\{t \mapsto 6, c_1 \mapsto \{$name = TODO, ...$\}\}$
Chg {name = Determine additional requests and abundances, ...}.name $\rightsquigarrow$ UC_[*]
Rate {1 (high)} $\{19, 21, 30, 64, 73, 84\}$ 10

Rep $\{70(term_n)\}$ terms(g_n) $\{t \mapsto 6, g_n \mapsto \{$dId = Glossary.xml, dState = 6, ...$\}\}$
Chg {name = Class, ...}.desc $\rightsquigarrow$ A class is part of a course. The class teaches ...
Rate {8 (high)} $\{70, 84\}$ 2

Rep $\{70(term_n)\}$ terms(g_n) $\{t \mapsto 6, g_n \mapsto \{$dId = Glossary.xml, dState = 6, ...$\}\}$
Chg {name = Teacher, ...}.desc $\rightsquigarrow$ He is the one who should transfer the ...
Rate {8 (high)} $\{70, 84\}$ 2

Rep $\{82(ri)\}$ repReadingInst(dState(sp))
$\{t \mapsto 6, sp \mapsto \{$dId = SpecManagement.xml, dState = 6, ...$\}\}$
Chg {dId = ReadingInst.xml, dState = 6, ...}.ri_crosscut $\rightsquigarrow$
{shouldRead = [Management], whichPart = Complete},
{shouldRead = [User, Designer, Test Team], whichPart = Complete}
Rate {2 (high)} $\{82, 84\}$ 2

Rep $\{84(doc)\}$ repDocSDM(t) $\{t \mapsto 6\}$
Chg {dId = Glossary.xml, dState = 6, ...}.status $\rightsquigarrow$ InProgress
Rate {1 (medium)} $\{70, 84\}$ 40

Rep $\{84(doc)\}$ repDocSDM(t) $\{t \mapsto 6\}$
Chg {dId = ReadingInst.xml, dState = 6, ...}.status $\rightsquigarrow$ InProgress
Rate {1 (medium)} $\{82, 84\}$ 40

Figure 12.23: Top-ranked repair set at state 6 (violated rules: 19, 70, 82, 84)

12.4 Sample Repair Collections

In this section, we present the repair collections for the repository states 6, 17, and 23, respectively. All collections are sorted by the following metric:

$$
\begin{aligned}
rs \geq_{\mathrm{SDM}} rs' \iff\ & c > c' \ \vee\ (c = c' \ \wedge\ |rs| \leq |rs'|) \\
\text{where } c\ &= \sum_{rep \in rs} \mathtt{rating}_{\mathrm{SDM}}(rep) \\
c'\ &= \sum_{rep' \in rs'} \mathtt{rating}_{\mathrm{SDM}}(rep') \\
& \mathtt{rating}_{\mathrm{SDM}}(\mathsf{Rep}\ _\ _\ _\ act\ (\mathsf{Rate}\ _\ violated\ cost)) \\
&= \begin{cases} -|violated| - cost - 60 & \text{if } act \text{ deletes a document} \\ -|violated| - cost - 50 & \text{if } act \text{ deletes content} \\ -|violated| - cost & \text{otherwise} \end{cases}
\end{aligned}
$$

We punish deletion by at least 50, because the maximum cost of a hint is 50 in our example rules. For brevity, we only show the top-ranked repair set for each state.

At state 6, the following rules are violated: 19, 70, 82, and 84. Our system derives 48 alternative repair sets, each of which includes seven repairs. Fig. 12.23 shows the repairs of the top-ranked repair set, where we use the rule numbers as references. Our system proposes the following repairs:

- Change the names of the use cases ‘Create new course and classes’ and ‘Determine additional requests and abundances,’ in order to resolve inconsistencies for Rule 19.

- In the glossary, roll back the descriptions of the terms Class and Teacher, in order to resolve inconsistencies for Rule 70.

- In the reading instructions, add the reader species management to the readers of the cross-cutting concerns, in order to resolve inconsistencies for Rule 82.

- Roll back the status of the glossary and the reading instructions to ‘in progress,’ in order to resolve inconsistencies for Rule 84.

At state 17, the following rules are violated: 21, 30, 48, 64, 70, 73, and 74. Our system derives 60 alternative repair sets, each of which includes 13 repairs. Fig. 12.24 shows the repairs of the top-ranked repair set including the following repairs:

- Change the use case UC_[Create new course and classes], such that the precondition ‘course type exists’ is implied by the postconditions of all other use cases.

- Change the use cases UC_[Assign students to course], UC_[set student to wish list], and UC_[Suggest teacher for course], such that their postconditions are implied by the postconditions of their used analysis functions.

- Change the data type DT_[Year], such that its lower bound 1900 is smaller than its upper bound 300.

- From the dialog Dia_[Assign student to course], remove the called use case. One might wonder that our system does not propose to add the analysis function AF_[Check maximum students for class] to the use case UC_[Assign students to course], which costs 1 only (see Fig. 12.16, pg. 203). This repair might, however, violate the rules 19, 21, 30, 64, 73, and 84. Consequently, it gets a lower rating. A repair set containing this repair is at best ranked fourth place.

- Change the glossary terms Class and Teacher as in state 6.

- In the use case UC_[Create course summary], change the used analysis function AF_[Filter courses] to AF_[Assign teacher to class].

- In the data model, change the entity type ET_[Class] and the component Com_[Course planning], such that ET_[Class] has less relations to entity types outside Com_[Course planning] than to entity types within Com_[Course planning]. Each of these repairs does not resolve any inconsistency — they must be applied together.

```
Rep {21(u)}    elements(c)
    t ↦ 17, c ↦ {name = Com_[Course planning], ...},
    u ↦ {name = UC_[Create new course and classes], ...}
  ?? [True : And [And [student assigned to class, ...], ...] |⇒ course type exists]
  Rate {1 (medium)} {19, 21, 30, 64, 73, 84}
Rep {30(u)}
  concatMap(elements, concatMap(flattenComp, concatMap(ucComp, ...)))
  {t ↦ 17, u ↦ {name = UC_[Assign students to course], ...}}
  ?? [True : And [] |⇒ And [student assigned to class, no conflicting assignments to ...]]
  Rate {2 (medium)} {19, 21, 30, 64, 73, 84}
Rep {30(u)}
  concatMap(elements, concatMap(flattenComp, concatMap(ucComp, ...)))
  {t ↦ 17, u ↦ {name = UC_[Set student to wish list], ...}}
  ?? [True : And [] |⇒ request in wish list]    Rate {2 (medium)} {19, 21, 30, 64, 73, 84}
Rep {30(u)}
  concatMap(elements, concatMap(flattenComp, concatMap(ucComp, ...)))
  {t ↦ 17, u ↦ {name = UC_[Suggest teacher for course], ...}}
  ?? [True : And [] |⇒ teacher suggestions for a course]
  Rate {2 (medium)} {19, 21, 30, 64, 73, 84}
Rep {48(dt)}    concatMap(datatypes, repDataTypes(t))
  {t ↦ 17, dt ↦ {name = DT_[Year], ...}}
  ?? [True : 1900 < 300]                    Rate {1 (medium)} {48, 84}
Rep {64(dlg)}
  concatMap(elements, concatMap(flattenComp, concatMap(diaComp, ...)))
  {t ↦ 17}
  Chg {name = Dia_[Assign student to course], ...}.usecases ⇝ []
  Rate {2 (medium)} {64, 84} 3
Rep {70(term_n)}    terms(g_n)    {t_n ↦ 17, g_n ↦ {dId = Glossary.xml, dState = 6, ...}}
  Chg {name = Class, ...}.desc ⇝ A class is part of a course. The class teaches ...
  Rate {8 (high)} {70, 84} 2
Rep {70(term_n)}    terms(g_n)    {t_n ↦ 17, g_n ↦ {dId = Glossary.xml, dState = 6, ...}}
  Chg {name = Teacher, ...}.desc ⇝ He is the one who should transfer the ...
  Rate {8 (high)} {70, 84} 2
Rep {73(f_ref)}    ref_fun(u)
    t ↦ 17, ucD ↦ {dId = UC_Register_Students.xml, dState = 17, ...},
    c ↦ {name = Com_[Internal data], ...}, u ↦ {name = UC_[Create course summary], ...}
  Chg AF_[Filter courses] ⇝ AF_[Assign teacher to class]
  Rate {1 (medium)} {19, 21, 30, 64, 73, 84} 1
Rep {74(et)}    elements(c)
    t ↦ 17, dm ↦ {dId = DataModel.xml, dState = 14, ...},
    c ↦ {name = Com_[Course planning], ...}, et ↦ {name = ET_[Class], ...}
  ?? [True : 2 ≤ 1]                         Rate {0 (medium)} {41, 74, 84}
Rep {74(c)}    concatMap(flattenComp, compsDM(dm))
    t ↦ 17, dm ↦ {dId = DataModel.xml, dState = 14, ...},
    c ↦ {name = Com_[Course planning], ...}
  ?? [True : 2 ≤ 1]                         Rate {0 (medium)} {41, 74, 84}
Rep {74(et)}    elements(c)
    t ↦ 17, dm ↦ {dId = DataModel.xml, dState = 14, ...},
    c ↦ {name = Com_[Course attendees], ...}, et ↦ {name = ET_[Student], ...}
  ?? [True : 1 ≤ 0]                         Rate {0 (medium)} {41, 74, 84}
Rep {74(c)}    concatMap(flattenComp, compsDM(dm))
    t ↦ 17, dm ↦ {dId = DataModel.xml, dState = 14, ...},
    c ↦ {name = Com_[Course attendees], ...}
  ?? [True : 1 ≤ 0]                         Rate {0 (medium)} {41, 74, 84}
```

Figure 12.24: Top-ranked repair set at state 17 (violated rules: 21, 30, 48, 64, 70, 73, 74)

- In the data model, change the entity type ET_[Student] and the component Com_[Course attendees], such that ET_[Student] has less relations to entity types outside Com_[Course attendees] than to entity types within Com_[Course attendees].

At state 23, the following rules are violated: 4, 21, 30, 41, 70, 74, and 80. Our system derives 430 alternative repair sets, which contain between 13 and 16 repairs. Fig. 12.25 shows the repairs of the top-ranked repair set, which includes 13 repairs. Our system proposes the following repairs:

- Add a screen type to the style guide. We do not know, however, how the screen type should look like.
- Change the use case UC_[Create new course and classes], such that the precondition 'course type exists' is implied by the postconditions of the other use cases.
- Change the use cases UC_[Assign students to course] and UC_[Suggest teacher for course], such that their postconditions are implied by the postconditions of their used analysis functions.
- In the data model, add a target cardinality to the relation 'is available at' emanating from the entity type ET_[Availability].
- Change the glossary terms Class and Teacher as in state 6.
- In the data model, change the entity type ET_[Class] and the component Com_[Course planning], such that ET_[Class] has less relations to entity types outside Com_[Course planning] than to entity types within Com_[Course planning]. Also, change the entity type ET_[Student] and the component Com_[Course attendees], such that ET_[Student] has less relations to entity types outside Com_[Course attendees] than to entity types within Com_[Course attendees].
- In the business process BP_[Register students], add the activities Act_[Register Customer (4)] and Act_[Set customer to wish list], where BP_[Register students] is defined in the document BP_Register_Students.xml.

12.5 Performance Summary

Tab. 12.8 summarizes the performance of our consistency checker. Column 1 shows the repository state; column 2 denotes the repository size in kilo bytes; column 3 shows the CPU time needed for generating S-DAGs using our basic S-DAG generation algorithm defined in Sect. 8.4.2; column 4 summarizes the rules evaluated during incremental S-DAG generation; and column 5 shows the CPU time needed using our incremental S-DAG generation algorithm defined in Sect. 8.4.3. All tests were performed against a DARCS [Rou04] repository

```
Rep {4(stG)}    [styleguide(cc)]
    {t ↦ 23, cc ↦ {dId = CrossCutting.xml, dState = 14, ...}}
    Chg {buttonTs = [], screenTs = []}.screenTs ⇝ [{name = TODO, ...}]
    Rate {20 (medium)} {3, 4, 19, 84} 1
Rep {21(u)}    elements(c)
      t ↦ 23, c ↦ {name = Com_[Course planning], ...},
      u ↦ {name = UC_[Create new course and classes], ...}
    ?? [True : And [course removed, planning request known, ...] |⇒ course type exists]
    Rate {1 (medium)} {19, 21, 30, 64, 73, 84}
Rep {30(u)}
    concatMap(elements, concatMap(flattenComp, concatMap(ucComp, ...)))
    {t ↦ 23, u ↦ {name = UC_[Assign students to course], ...}}
    ?? [True : And [] |⇒ And [student assigned to class, no conflicting assignments to ...]]
    Rate {2 (medium)} {19, 21, 30, 64, 73, 84}
Rep {30(u)}
    concatMap(elements, concatMap(flattenComp, concatMap(ucComp, ...)))
    {t ↦ 23, u ↦ {name = UC_[Suggest teacher for course], ...}}
    ?? [True : And [] |⇒ teacher suggestions for a course]
    Rate {2 (medium)} {19, 21, 30, 64, 73, 84}
Rep {41(rel)}    relatesWith(et)
      t ↦ 23, dm ↦ {dId = DataModel.xml, dState = 22, ...},
      et ↦ {name = ET_[Availability], ...}
    Chg {name = is available at, ...}.targetcard ⇝ TODO
    Rate {6 (medium)} {41, 74, 84} 1
Rep {70(term_n)}    terms(g_n)    {t_n ↦ 23, g_n ↦ {dId = Glossary.xml, dState = 6, ...}}
    Chg {name = Class, ...}.desc ⇝ A class is part of a course. The class teaches ...
    Rate {8 (high)} {70, 84} 2
Rep {70(term_n)}    terms(g_n)    {t_n ↦ 23, g_n ↦ {dId = Glossary.xml, dState = 6, ...}}
    Chg {name = Teacher, ...}.desc ⇝ He is the one who should transfer the ...
    Rate {8 (high)} {70, 84} 2
Rep {74(et)}    elements(c)
      t ↦ 23, dm ↦ {dId = DataModel.xml, dState = 22, ...},
      c ↦ {name = Com_[Course planning], ...}, et ↦ {name = ET_[Class], ...}
    ?? [True : 2 ≤ 1]                        Rate {0 (medium)} {41, 74, 84}
Rep {74(c)}    concatMap(flattenComp, compsDM(dm))
      t ↦ 23, dm ↦ {dId = DataModel.xml, dState = 22, ...},
      c ↦ {name = Com_[Course planning], ...}
    ?? [True : 2 ≤ 1]                        Rate {0 (medium)} {41, 74, 84}
Rep {74(et)}    elements(c)
      t ↦ 23, dm ↦ {dId = DataModel.xml, dState = 22, ...},
      c ↦ {name = Com_[Course attendees], ...}, et ↦ {name = ET_[Student], ...}
    ?? [True : 1 ≤ 0]                        Rate {0 (medium)} {41, 74, 84}
Rep {74(c)}    concatMap(flattenComp, compsDM(dm))
      t ↦ 23, dm ↦ {dId = DataModel.xml, dState = 22, ...},
      c ↦ {name = Com_[Course attendees], ...}
    ?? [True : 1 ≤ 0]                        Rate {0 (medium)} {41, 74, 84}
Rep {80(act_f)}    activities(bp_f)
      t_f ↦ 23, bpD_f ↦ {dId = BP_Register_Students.xml, dState = 21, ...},
      bp_f ↦ {name = BP_[Register students], ...}
    Add {name = Act_[Register customer (4)], ...}      Rate {10 (high)} {80, 84} 2
Rep {80(act_f)}    activities(bp_f)
      t_f ↦ 23, bpD_f ↦ {dId = BP_Register_Students.xml, dState = 21, ...},
      bp_f ↦ {name = BP_[Register students], ...}
    Add {name = Act_[Set customer to wish list], ...}  Rate {10 (high)} {80, 84} 2
```

Figure 12.25: Top-ranked repair set at state 23 (violated rules: 4, 21, 30, 41, 70, 74, 80)

State	Repository Size (KB)	Basic S-DAG Generation Time (Sec.)	Incremental S-DAG Generation evaluated Rules	Time (Sec.)
1	10	2.26	2 3 82 84	2.33
2	28	2.33	3 4 19 70 82 84	2.39
3	36	2.76	19 21 30 64 73 80 84	2.51
4	44	3.37	19 21 30 64 73 80 84	2.72
5	56	4.20	19 21 30 64 73 80 84	2.92
6	73	5.39	4 41 70 80 82 84	3.26
7	77	6.64	19 21 30 64 73 80 84	3.31
8	81	8.56	19 21 30 64 73 80 84	3.64
9	81	10.60	4 41 80 82 84	3.54
10	85	13.09	19 21 30 64 73 80 84	3.97
11	85	15.51	19 21 30 64 73 80 84	4.18
12	89	17.98	80 84	3.82
13	88	20.70	4 41 80 82 84	3.87
14	100	23.99	3 4 19 41 48 74 84	4.64
15	108	28.16	19 21 30 64 73 84	4.55
16	112	32.30	19 21 30 64 73 84	4.79
17	116	36.56	19 21 30 64 73 84	5.03
18	116	44.03	4 41 48 74 80 82 84	5.76
19	116	50.49	19 21 30 64 73 80 84	5.99
20	117	59.88	19 21 30 64 73 80 84	6.14
21	125	67.58	19 21 30 64 73 80 84	6.45
22	125	79.09	4 41 74 80 82 84	6.86
23	125	89.81	4 41 80 82 84	13.45
24	125	99.08	41 74 84	5.58

Table 12.8: Performance summary

(version 0.9.20) on a Debian Linux operating system using a Dell X200 laptop with 384 mega bytes RAM and an Intel PIII CPU running at 800 mega Hertz. We compiled our prototype system using the Glasgow Haskell Compiler version 6.2.1 [GHC04] with all compiler optimizations turned on. For both performance tests, we used the miniscoped rules, such that incremental S-DAG generation could benefit from rule filtering and incremental evaluation only. Incremental S-DAG generation shows satisfactory performance for our example. We see, however, that the time needed for incremental S-DAG generation grows to great extent at state 23. Clearly, this is due to Rule 80, which we cannot check efficiently, if the repository contains a finished fine specification: Below the topmost disjunction of Rule 80 we have to use our non-incremental S-DAG generation algorithm. Recall that disjunctions are evaluated lazily. Thus, non-incremental S-DAG generation is fast at repository states between 1 and 22, because the repository does not contain a finished fine specification at these states. Consequently, the universal quantifier for the variable s_c is never evaluated. In contrast, at state 23 the universal quantifier for s_c has to be evaluated (non-incrementally). Since the quantifier iterates over specifications in *previous* repository states (which must be rebuilt by DARCS), its evaluation is expensive. At state 24, we do not need to re-evaluate Rule 80, because it is not affected by modifications to the data model. This significantly reduces

the time needed for S-DAG generation. If, however, at a later state, a software engineer checks in a specification document or a business process document, then Rule 80 must be re-evaluated, which again will be expensive.

For analyzing the overhead introduced by incrementalization, we add up the last column, which results in an overall time of 111.71 seconds. This is equivalent to an overhead of about 13%, compared to classic S-DAG generation. Most of this overhead results from reading S-DAGs, which is more expensive than reading consistency reports.[1] Compare the results in Tab. 12.8 to the performance results of our running example, shown in Tab. 6.4 (pg. 84). There, the benefits of our incremental approach compensate the overhead introduced by reading consistency reports.

Tab. 12.8 lists the times needed for S-DAG generation only, i.e., the duration at which the repository is locked. But how is the performance of S-DAG augmentation and repair derivation? In our experiments, it turned out that S-DAG augmentation is quite cheap. Our system needed at most 6 seconds (at state 23). In contrast, repair derivation is expensive, if many rules are violated and there are many alternative repairs possible: at state 6 our system needed 0.18 seconds; at state 17 our system needed 5.23 seconds; and finally at state 23 our system needed 128.52 seconds. Recall that repairs are derived from the (reduced) current S-DAGs, i.e., documents in the repository are not accessed. Times needed for deriving repairs directly from the repository are significantly longer; also, the repair collections are significantly larger. These results give strong evidence that separating incremental S-DAG generation from repair derivation is a key to making our consistency maintenance approach viable.

[1] Clearly, this is mostly due to our prototype Haskell implementation, which uses derived instances of the type classes `Show` and `Read`. These instances tend to be slow for complex data structures like S-DAGs.

Chapter 13

Lessons Learnt: Costs and Benefits of Consistency Management

In this chapter, we summarize the most important lessons we have learnt from our case study. We have successfully applied our consistency maintenance approach to a complex application domain. The results of our case study prove our introductory claims:

- Tolerating inconsistencies rather than preventing them is a feasible way to manage semantic consistency requirements in document engineering. Inconsistencies are natural when multiple authors edit interrelated documents. The vast majority of rules is weak; only a few rules are strong. As a by-product, our approach also documents intentional design decisions, which show up as inconsistencies.

- By tolerating inconsistencies and automating consistency checks we have been able to smoothly integrate consistency maintenance into the everyday work of authors. Our approach does not require any adaptations to document engineering processes themselves.

- Our consistency maintenance approach scales to a practically relevant scenario. We can formalize many important consistency requirements. Temporal consistency rules support to restrict the development of documents and to implement document life cycle restrictions. Incremental consistency checking shows passable performance.

- Augmented S-DAGs provide a good means to show inconsistencies and possible repairs. Repair collections combine the repairs for the individual rules, which is important when many rules are violated. Due to the combinatorial explosion of the number of possible repairs, user-defined preference metrics are necessary, even though repairs are derived from *reduced* S-DAGs.

- Separating S-DAG generation from repair derivation is a key to making our consistency maintenance approach viable. Deriving repairs directly from the documents in the repository does not scale in practice.

- The quality of the generated repairs depends on the quality of hints and the document structure. The more detailed the document structure is, the better are the repairs.

Task	Time (Days)
Extract consistency requirements (140) from analysis modules	14
Define DTDs for analysis modules	4
Define document types for analysis modules	2
Generate parsers (automatically from DTDs)	0.2
Adjust HaXml generated parser output types to document types	2.5
Define additional functions and predicates	0.5
Formalize consistency rules (15)	4
Sum	27.2

Table 13.1: Cost of formalization

- Subtypes play an important rôle for defining the language SDM. The definition of document types is a laborious task. We expect, however, that languages can be re-used for other rules. Since the basic language Prelude includes large part of the functional programming language Haskell, language designers have to define a few new symbols only.

Clearly, we share a serious drawback with other formal consistency management approaches: the *cost of formalization.* Tab. 13.1 summarizes the time we spent for separate tasks, in order to run our full case study. We extracted 140 consistency requirements from sd&m's analysis modules. We defined document types for specification documents and the analysis modules goals scope constraints, overview of the technical architecture, cross-cutting concerns, business processes, use cases, analysis functions, data model, data types, dialogs, reading instructions, and glossary. We have formalized the 15 consistency rules shown in Chapter 11.

Most of the time we spent for determining useful consistency requirements. Surely, this task is of major importance and will be time consuming for other application areas as well. Formalizing consistency rules gives precious insights to the consistency requirements actually needed, which is of vital importance for any collaborative work. To our experience, prior to formalization, unclear consistency requirements lead to serious misunderstandings. Developing useful document structures was a laborious task, too. Defining document types and adjusting the HaXml parsers could be reduced to zero time, if HaXml supported XML Schema (see App. B.4). Then our document types could be derived directly from the XML Schemas.

Besides proving our claims, this case study also reveals some limitations of our approach, which opens interesting directions for future research. As yet, we cannot formalize *natural-language constraints.* We are, however, confident that our approach benefits from the ongoing development of natural-language tools. Recently, the knowledge management community has developed parsers [BQBW03, BHQW02] that extract knowledge from natural-language texts and present it by semantic nets (e.g., RDF graphs [W3C99a, Pow03] or Topic Maps [ISO03, WM02]). Annotating rules by *hints* and *costs* as well as defining *preference metrics* is done in a rather ad-hoc manner, which tends to be complex and time consuming. We plan to use data mining approaches [AIS93, Chi03] that employ historical information about the rules violated and the usefulness

of the repairs derived. This will help to define better metrics for repair ranking and to provide support for hint formalization.

At sd&m, our consistency requirements have been integrated into the analysis modules as a "requirements catalog," which gives an overview of all requirements. In the future, we will introduce consistency maintenance into the development process of real software specifications, where we shall concentrate on the following aspects.

- Industrial specifications for large systems, which are subdivided into subsystems and components, suffer from complex interrelations between analysis modules for functional requirements. Therefore, we shall first introduce formal consistency rules examining relationships between business processes, use cases, analysis functions, and dialogs in conjunction with the data model and data type definitions. We are confident that in this field our consistency maintenance approach already achieves major benefits by imposing relatively low formalization costs.

- Usually, new systems are integrated in a complex environment consisting of many neighborhood systems. Also, the migration from an old system to a new system is not trivial. Therefore, we shall find a formal structure for describing neighborhood interfaces, in order to apply our consistency maintenance approach.

In summary, we are convinced that the benefits of consistency management outweigh its costs. Once formalized, we can re-use consistency rules for a number of document engineering projects.

Part IV

Conclusions

Chapter 14
Comparison with Related Work

In this chapter, we compare our consistency management approach with related work in the literature. The major differences to these approaches are:

- We apply consistency management to documents and integrate our approach into the work with an arbitrary DMS.
- Instead of enforcing consistency, we tolerate inconsistencies.
- Our approach is independent from a particular document model and a particular document format.
- We support user-defined functions and predicates. Hints guide repair generation by providing domain-specific knowledge.

By the above design decisions we gain flexibility. We accept that incremental consistency checking is more coarse grained than in other approaches and that repairs have to be applied manually. Mainly, our work has been influenced by consistency management approaches in databases and recent research in the XML community. In particular, we share some similarities with the consistency checking tool xlinkit [NEF01, NCEF02, DENT02, NEF03, PNEF03]. In the following sections, we discuss related work in detail. In Sect. 14.1, we compare our approach to other consistency checking approaches. In Sect. 14.2, we put particular attention to efficient incremental evaluation of consistency rules. We compare our approach to other consistency maintenance approaches in Sect. 14.3. Finally, we review software engineering tools w.r.t. their consistency management facilities.

14.1 Consistency Checking

In consistency checking, we distinguish between consistency enforcement and inconsistency tolerance. In particular, database consistency checking approaches enforce consistency [Pac97].

Despite of its limited constraint language, the database programming language Thémis [BD95] shares some ideas with our approach. Higher-order complexity lays in methods, implemented in an imperative programming language. A Cardelli-style type system supports implicit subtyping but lacks variant types. Universal first-order formulae express integrity constraints; the predicate set is fixed. Recent works on semistructured databases [BFW00] and integrity constraints for WWW sites [FFLS99] use decidable subsets of first-order logic. We need more expressive power as our examples have shown.

At first sight, we might use the DMS metadata database and permit violations of database integrity constraints. The database "corset" is, however,

too strict and limited to document metadata only. Complex semantic rules, e.g., "referenced sections should keep similar over time," require to inspect document *content* via information retrieval techniques [WS99]. Our approach benefits from the semantics of underlying document models, expressed by our type system. Metadata impinging consistency may change, which would require to adapt the database schema. In addition, for smaller projects the database approach appears too heavy weight. Of course, our database-independent approach can still use databases for fast metadata access.

Recent approaches in the XML community [AFL02, AL03b] aim to guarantee consistency by construction. In contrast, we apply consistency checks at concrete document instances to determine inconsistencies. In spite of the drawback that we cannot guarantee consistency by construction, we argue that our approach is better suited for handling semantic (i.e., content-dependent) inconsistencies. Also, document models cannot handle consistency requirements that vary over time. We integrate incremental consistency checks into a DMS, in order to relieve the obstacle of recurring consistency checks.

In software engineering [Fin00, NER00], tolerating inconsistencies is considered preferable to enforcing consistency. Many-valued logics help to model inconsistencies and their consequences [EC01]. We argue, however, that classic logic is easier to understand for users and that our approach is powerful enough to manage consistency. Priority levels for consistency rules can be considered a coarse grained approach to many-valued logic.

The idea to compute inconsistency diagnoses rather than just detecting that an inconsistency has occurred is not new. It has been explored in the context of knowledge bases [VC99], in software engineering [Bal91, NG92, EFKN94, GHM93, GP97, GHM98, Fin00, NER00], and for analyzing consistency between documents [NCEF02]. Whereas for knowledge bases, decidability of the implication problem is crucial, we find close relationships in the context of distributed documents. The toolkit xlinkit [NCEF02] statically checks distributed documents against user-defined consistency rules and implements a tolerant consistency model. Rules are formalized in an untyped non-temporal first-order logic employing user-defined predicates, the semantics of which is implemented in Java-script [ECM99]. This "black box" hinders the re-use of already available algorithms. We share many ideas with xlinkit but use a DMS, which has a lot of advantages: (1) DMSs are widely used and provide useful management mechanisms, e.g., document locking; (2) Internet access to DMSs already supports distribution and collaboration; and (3) history information (already stored by DMSs) is necessary for temporal consistency rules and efficient consistency checking. In addition, we employ a sophisticated type system that helps to define meaningful consistency rules. We argue that DMS-managed repositories support collaborative distributed work and should be extended, in order to provide consistency management.

Henrich et al. [Hen95, HD96, HR01] also use a repository for consistency checking but employ a query language. By this approach query results correspond to our consistency reports. Using a query language is, however, limited

to consistency checking. In addition, Henrich et al. support a fixed set of predicates only.

For defining document types, we might also have used finite tree automata [Toz01]. We decided not to do so, because our type system is closer to the type system of Haskell — the programming language used by language designers.

14.2 Incremental Evaluation

The idea of incremental evaluation is old; we can trace it back to the Analytical Engine of Charles Babbage in the early 19th century [Bab84]. In this section, we discuss closely related work in the area of incremental evaluation and efficient consistency checking. In contrast to our approach, most of the other incremental approaches *enforce* consistency and thus benefit from total satisfaction of consistency rules prior to consistency checking.

Incremental programming languages [YS88, YS91, Liu99, Liu00, ABH03] provide a uniform approach to incremental computation. They attempt to minimize redundant computation provided that a program is executed repeatedly on slightly different inputs. In order to produce the current results of a program run, incremental computation uses previous results, differences between previous inputs and current inputs, and auxiliary information. The primary goal is to reduce the asymptotic running time of incremental programs by reducing inputs. Our treatment of quantifier spheres follows this approach. We can regard quantified formulae $Q\ x \in e \bullet \phi$ as incrementally computable "functions" having as input the value of the sphere term e. These functions have a complexity linear in the input size (the cardinality of the sphere). During consistency checking, we try to reduce the sphere. Subtypes aid our algorithm in reducing the amount of intermediately stored data.

From its origin, the database community has been striving for efficient algorithms that check static integrity constraints [WDSY91, GSUW94, SdS99, ABC99], check temporal integrity constraints [Ple93, GL95, Ple95, Ple96, MS96, BS98, BCP99], maintain views [GMS93, MS01], and optimize queries [DST94, DS95, BM95, Nak01]. Pacheco e Silva [Pac97] classifies recent constraint checking approaches in databases. Usually, database approaches distinguish between compile-time analysis and run-time techniques.

Compile-time analysis simplifies integrity constraints and identifies update types that might violate constraints. Usually, rewriting approaches (like miniscoping [dC86, MH89]) and quantifier elimination strategies are used [Bas99]. We also employ miniscoping but do not eliminate quantifiers, because quantified variables provide a precise characterization of inconsistencies. Also, database approaches limit their constraint language, in order to make constraint subsumption decidable. We cannot benefit from constraint subsumption, because this is undecidable in our rule language; miniscoping performs basic subsumption analyses only. In order to achieve the high expressivity needed by our applications, we sacrifice decidability.

Run-time approaches use previous results to avoid re-computation. In the context of first-order logic, they reduce quantifier spheres. Within constraints most database approaches use a *fixed* set of predicates, partly based on the formal database model. Then knowledge about predicate properties — such as transitivity or symmetry — supports incremental algorithms. In addition, some database approaches limit updates to insertion only [DST94, DS95, SdS99], in order to achieve high performance. In general, we follow the approach of the database community but do not have a formal database schema, formalized updates, and consistency prior to updates. This makes our approach more complicated. Another obstacle is that even recent revision control systems [C+02, Rou04, CSFP04] formalize state transitions by line based diffs or patches, which ignore the document model. In order to achieve the fine granularity needed for database approaches, we have to parse documents. Since we make no assumptions to the document model of the underlying DMS, our approach cannot show the high performance found in databases. In document management, however, our coarse grained approach towards incremental consistency checking suffices, because typically DMSs do not have to carry the high load of databases. We are confident that revision control approaches in the XML community [CRGW96, MACM01, CAM02, WL02, WDC03] lead to a formal document update language that respects the document model. In temporal databases, the problem of efficiently storing and managing historical database states arises. In our setting, historical repository states are managed by the DMS already.

Recently, incremental evaluation techniques have been used by the XML community to maintain consistency w.r.t. user-defined rules. In [BBG+02], Benedikt et al. extend XML Schema constraints by value based constraints. There, simplified first-order logic and formalized update operations are the basis for incrementally checking constraints. In [KSR02], Kane et al. propose to translate XML Schema constraints to XQuery; thus gaining declarative integrity constraints. The approach closely follows the database constraint checking procedure: First determine which rules might be affected by an update, second check these rules at updated elements. In [KG02], Kwong and Gertz develop a new language XSC for structural integrity constraints in XML documents. The authors propose XSC as an alternative way to DTDs or XML Schema for expressing document structure. Implication and satisfiability are both decidable for constraints in XSC. Xlinkit [PNEF03] uses static analysis to filter consistency rules relevant to document changes and a tree-diff algorithm that determines document parts that have to be re-checked. Its tolerant view of consistency distinguishes xlinkit from other approaches and makes it closer to our approach. By supporting distribution and avoiding a history-aware repository, xlinkit cannot implement temporal consistency rules and lacks some of the incremental techniques we benefit from.

Incremental evaluation approaches in the literature show that incrementality comes with a cost, which is, in general, hard to analyze, e.g., storage for previous results and history information, lookup costs, and computation of differences between inputs. The notion that incremental evaluation is "often" cheaper than

computation from scratch can, therefore, only be regarded as a heuristic leaving open the question when to use or when to avoid incremental computation. This highly depends on the actual application. Therefore, we have proved our claims by a case study. In general, database performance analyses show that the smaller the changes through updates are and the less expressive the constraint language is, the better performs incremental computation. This is also confirmed by our prototype, which performs better for small documents than for large documents.

14.3 Consistency Maintenance

Like detecting inconsistencies, the issue of *repairing* inconsistencies has received much attention in the literature, particularly in the field of active databases. In an active database, so called ECA rules [ISO99a, ISO99b, Pat02] support to fire a repair *A*ction provided that a specified *E*vent has taken place and a *C*ondition holds. Current research in the field of active constraint maintenance concentrates on the derivation of such active rules.

[MT99] provides a general framework that characterizes recent research in the area of active constraint maintenance. With the help of this framework, we categorize our approach to repairing inconsistencies as follows:

- *Problem addressed*: In contrast to many database constraint maintenance approaches, we are concerned with consistency maintenance only; we do not consider view updating. Our approach supports to check, maintain, and restore consistency. Usually, databases neglect restoring consistency by requiring overall consistency prior to database updates. A notable exception is [Maa98], which we discuss below. We apply our techniques at compile time and run-time.

- *Database schema*: Similar to most database constraint maintenance approaches, we employ logic as rule language. We use, however, *full* first-order predicate logic. Functions correspond to database views, i.e., derived data not stored in the database. Our approach also facilitates non-flat consistency rules (which can contain database views) through the use of (possibly recursive) functions. We support both static and temporal consistency rules.

- *Update requests*: We can handle multiple update requests including insertion, change, and deletion.

- *Mechanism*: We employ active rules whose repairs are derived from S-DAGs, which in turn are generated from the documents in the repository (document contents correspond to base facts in a database). Instead of active rules, other constraint maintenance approaches use SLDNF resolution [TO95, Dec97]. Resolution approaches are, however, limited to decidable subsets of (predicate) logic. In contrast to databases, we rely

on user participation for resolving inconsistencies. We cannot apply repairs to documents automatically, because the semantics of documents is much more complex than that of databases.

- *Solutions*: In contrast to many database constraint maintenance approaches, we do not strive for complete repairs. Our repairs are correct w.r.t. the individual rules, provided that hints from the rule designer actually invert truth values of atomic formulae. We cannot guarantee correctness of repairs w.r.t. all rules in the rule system, even though we eliminate contradicting repairs.

When classified according to the above framework, our approach falls in the same category as [GL97] and [Maa98]. One advantage of our approach is flexibility, for we do not rely on a specific document model. On the other hand, less formal update descriptions and less formal document semantics prohibit automatic application of computer-generated repairs.

Our strategy for deriving repairs corresponds to the strategy of minimal change and preserving the update, as proposed by Gertz and Lipeck [GL97]. There, automatic execution of repairs is interleaved with further consistency checks, in order to arrive at a consistent database. We guide our strategy by repair hints and execute repairs manually. In contrast to many other approaches, [GL97] supports incomplete information by null values. We handle lack of information by value skeletons, which correspond to database tuples with null values. In contrast to our approach, the consistency rule language in [GL97] is limited to a fixed set of predicates and integrity constraints in implicative normal form, which restricts the interaction between universal and existential quantifiers. In addition, Gertz and Lipeck neglect database views.

Maabout [Maa98] proposes to generate update rule programs from consistency rules. In contrast to many other database constraint maintenance approaches, [Maa98] does not require full consistency prior to database updates; it can be used to *restore* consistency. The derivation of update programs is, however, mostly done on the syntax level, i.e., update programs are derived directly from the consistency rules.

In [BP00], Bertossi and Pinto extend the specifications of the dynamics of databases by an approach that handles active rules. Derived repairs are integrated into the logical specification of a database's behavior. In contrast to our approach, Bertossi and Pinto formalize the dynamic database behavior in a situation calculus. Clearly, this is beyond our intentions: We are interested only in suggesting repairs that resolve inconsistencies in a repository. Also, it is still unclear whether a fully formal approach is applicable to document engineering, because a formal model is lacking here.

Xlinkit also derives repairs, in order to resolve inconsistencies [NEF03]. Repairs are derived from a set of sets of actions and a link base, which is quite similar to a consistency report. The approach guarantees completeness and correctness of repairs. In contrast to xlinkit, we reduce repairs through the use of S-DAGs. We consider this a major contribution. For example, xlinkit does

not attempt to select a specific element from the sphere of an existential quantifier, in order to derive good repairs. We support high-level repairs generated from domain-specific hints; in contrast, xlinkit supports filtering repairs only. Since xlinkit does not pre-process the consistency rules (e.g., to miniscope) its semantics appears quite complex. Also, interaction of repairs for different rules and compatibility of repairs w.r.t. the document structure are neglected. We partially solve the former problem by computing the hitting collection for all repair collections and ensuring compatibility within repair sets. Similar to xlinkit, we cannot guarantee compatibility of derived repairs w.r.t. document structures. We consider, however, static type checking of hints an important step towards such a property.

14.4 Software Engineering Tools

In software engineering, special tools are used, in order to achieve consistency. These tools are, however, limited to a specific document format and to a specific application domain, e.g., the development of software specifications or specific programming languages.

Programming language environments [Rep84] evaluate semantic rules of the underlying programming language on abstract syntax trees of source code documents. Later work on software engineering environments [GOO94, EJS95] supports consistency checks across multiple documents. Rules are, however, limited to a subset of a (non-temporal) first-order logic. In addition, all rules are equally important.

CASE tools [Bal98] may be used to aid software development. Usually, these tools check a fixed set of consistency rules, e.g., referential integrity constraints or naming conventions, see e.g., [Ber04]. Even state of the art CASE tools are limited to the documents they maintain, e.g., UML models (see [ZK03] for a recent consistency maintenance approach). It is hardly imaginable that all results of a specification can be maintained by a *single* CASE tool. Also, CASE tools implement a strict view of consistency. In contrast, we have developed a tolerant consistency management approach that supports documents of heterogeneous content and structure.

Naturally, formal software specification approaches [Wir90, Rat94, ABK+02] provide powerful means for consistency management [RS01]. As yet, these approaches require enormous effort for formalization, which often means to implement a new software development process. In practice, such effort is infeasible for large parts of a specification. Even recent advances towards executable specifications [HD04] cannot eliminate these efforts. Therefore, formal approaches are limited to critical parts of software systems. In contrast, our consistency management approach can be integrated into arbitrary development processes without major efforts. It can also be used to combine the advantages of formal and informal software specification approaches, e.g., by consistency rules that capture requirements between Z specifications and business processes.

In UML [HK99, OMG03], constraints can be defined using the Object Constraint Language (OCL) [WK98]. Our syntax of consistency rules shares some similarities with OCL invariants. Thus, although large part of a specification cannot be described in UML [SSKK02], we could use OCL syntax to formalize consistency rules. Then we would employ a UML class model to define document types. Temporal OCL extensions [FM02a, FM02b, ZG02] could be used to formalize temporal rules. OCL supports user-defined referentially transparent methods, called {queries}. But, since object oriented programming languages cause side-effects, one would need to employ a functional programming language, in order to define these methods. Recall that referential transparency is a fundamental prerequisite to efficient consistency checking. Therefore, our syntax follows the syntax of functional programming languages. We regard, however, syntax issues a matter of personal preference. If needed, OCL invariants can be translated to our consistency rules. The main contributions of this thesis are precise inconsistency pointers, efficient consistency checking, efficient generation of useful repairs, and integration of consistency management into DMSs. These contributions had to be developed for OCL as well.

Chapter 15
Conclusions and Outlook

We are confident that this thesis is a significant step towards consistent document engineering by using consistency maintaining DMSs. With kind permission of Universität der Bundeswehr München, preliminary results of this thesis have been published as [SBRS03a, SBRS03b, SBRS04c, SSBS04, SBRS04b, SBRS04a].

15.1 Summary

In a multi-author environment, DMSs provide fundamental management facilities, e.g., version control, access management, or deployment management. DMSs neglect, however, semantic domain-specific consistency requirements. On the one hand, inconsistencies are major obstacles causing time-consuming manual effort, which results in either schedule delays or budget holes or both. On the other hand, inconsistencies are natural. Therefore, in this thesis, we complement traditional DMSs by consistency management. We employ a *tolerant* consistency model; i.e., we accept inconsistencies but precisely identify them for manual resolution. We smoothly integrate consistency management into arbitrary DMSs *without* requiring adaptations to document engineering processes. Thus, our formal consistency management approach is a useful aid to informal document engineering.

Semantic consistency requirements are formalized by consistency rules in a full first-order predicate logic with explicit linear time. This expressive rule language supports formalization of a wide range of useful semantic consistency requirements. As yet, we have not found a practically relevant example that requires higher-order logic. Consistency rules can express intra- and inter-document requirements regardless of the document model and the document format used. Function symbols and predicate symbols used in rules are defined in domain-specific languages. The basic language Prelude includes large part of Haskell, such that only a few new symbols have to be defined for a concrete project. Our Cardelli-style type system facilitates definition of complex document types and aids formalization of syntactically correct consistency rules. Parametric polymorphism, higher-order symbols, and subtyping have proven useful features. The separation of rule formalization (document model independent) from language definition (document model dependent) supports re-use of rules and languages.

Our new tolerant semantics points out inconsistent document parts. We precisely indicate when, where, and why inconsistencies occur. In order to achieve scalability to a practically relevant problem size, we employ several techniques to speed up consistency checking. At the time of rule definition, we associate a set of affected documents to a rule. Thus, at a check-in only

affected rules have to be re-evaluated. In addition, we rewrite rules, in order to lower their static evaluation time complexity. During consistency checking, we evaluate consistency rules only on modified documents if possible. Our performance measurements give strong evidence that by the above methods we achieve satisfactory performance.

Besides pinpointing inconsistencies, this thesis introduces new techniques for generating document repairs that can resolve inconsistencies. In contrast to many other approaches, we concentrate on generating a few useful repairs from which authors can choose. Exponential computational complexity of repair enumeration motivates our new two-step approach: During consistency checking, our system generates for each rule an S-DAG; on author demand our system derives a single repair collection from all S-DAGs. Annotations from the rule designer make domain knowledge available — a fundamental requirement for flexible inconsistency handling and useful domain-specific repairs. S-DAGs are optimized for efficient generation. Quantifier edges carry concrete repair actions. A major contribution of our S-DAG approach is an effective reduction strategy that eliminates expensive actions and actions that are probably not useful. From S-DAGs, authors can choose actions for individual rules interactively. The impact of an action towards consistency of other rules remains, however, unclear. Therefore, we derive a single repair collection from all S-DAGs. The collection contains alternative repair sets; within each set, all repairs are necessary to resolve all inconsistencies for all rules. By construction, each repair set contains compatible repairs only; the repair sets are mutually independent alternatives. The repair collection can be sorted w.r.t. user-defined preference metrics; only the best repair sets are presented to authors. Preference metrics also provide a good means for partial inconsistency resolution, which resolves the most troubling inconsistencies only and leaves other inconsistencies for later resolution.

In a case study, we apply our consistency maintenance techniques to a challenging research topic in software engineering. Inconsistencies are major obstacles in software specifications for large systems. The case study proves that our tolerant approach to consistency management is a pragmatic way to handle inconsistencies in industrial software specifications. Our case study also shows that our techniques provide a useful aid and scale to a complex scenario. Since the effort for formalization is tunable to the specific application and our prototype shows satisfactory performance, we are confident that our contributions scale to an industrial setting.

Although our prototype works well in practice, it is fairly complex to use. The main reason is lack of graphical tools, which would provide comfort for rule designers, language designers, and authors. Currently, rules and languages are formalized in an XML syntax. S-DAGs and repairs are generated as XML structures, too. We plan to develop an integrated consistency maintenance tool, which supports (1) to define languages and rules, and (2) to interactively choose repairs from S-DAGs and repair collections, respectively. This might be no research issue. End user comfort is, however, crucial for a wide application of formal consistency management. Also, our prototype implementation lacks

XML Schema support. Currently, due to limitations of HaXml, we can support document type generation from DTDs only. This causes some programming effort to convert the Haskell types generated by HaXml to Haskell types generated by our prototype system. Therefore, we plan to extend HaXml by support for an XML Schema subset.

15.2 Future Research

It might appear that this thesis opens more questions than it answers. In fact, we see our work as a good starting point for future research.

As already pointed out, formalization of consistency rules is not trivial. Although we use full first-order predicate logic, we could employ approaches from Bry and Torge [Tor98, BT99], in order to investigate whether the rules are satisfiable. Also, some limitations in our type system could be relieved: For example, we could support partial application of function symbols unless quantifiers iterate over functions. In particular, this would support a function composition operator, which proves its usefulness in functional programming. Though efficiency of repair derivation is not of vital importance, we could incorporate other (more efficient) hitting set algorithms for repair derivation. For example, efficient genetic algorithms [LY02] do not find all minimal hitting sets. It would be interesting to see how this property impacts the usefulness of the generated repairs.

Our used logic supports linear time only. We plan to extend our approach to branching repositories, which means to quantify over individual branches, e.g., by a branching time logic like CTL(*) [Eme90]. This appears a straightforward extension from the theoretical point of view. It is, however, still unclear which applications require branching support. Another interesting research direction is support for distributed repositories in conjunction with partial consistency checking. Consider a distributed repository: How does a check-in to one repository part impact consistency in other repository parts? Which parts of the distributed repository have to be re-checked? How do network delays influence consistency checking time?

So far, we have not made any assumptions about the document format. Clearly, a restriction to XML would improve our incremental consistency checking techniques, due to formal update descriptions that are related to the actual document format. Currently, incremental consistency checking is quite coarse grained, because DMS indicate only *that* a document has been changed; they are not aware of the document structure. Therefore, we evaluate XML revision control approaches [CRGW96, MACM01, CAM02, WL02, WDC03] and try to implement a suitable approach into a revision control system [Rön04]. Also, restricting documents to XML would facilitate generation of document patches from repairs. Our system could apply such patches to documents automatically and send these suggestions to authors. Moreover, the system could check whether a repair (patch) is compatible to the document structure.

Sometimes, repairs appear “long-winded” to authors, which is due to complex consistency rules. We are confident that extending our rule definition language by non-standard logical connectives can improve the quality of repairs. We consider connectives like “there exists exactly one ... ” or “... is unique.” Also, a `let` construct might be useful for “re-using” terms and defining local functions within rules. So far, our approach only uses S-DAGs for deriving repairs. Annotating rules by hints and costs as well as defining preference metrics is done in a rather ad-hoc manner, which tends to be complex and time consuming. In the future, we plan to use data mining approaches [AIS93, Chi03] that employ historical information about violated rules and the usefulness of derived repairs. This will help to define better metrics for repair ranking, to provide support for hint formalization, and to adapt our strategies for S-DAG reduction and repair derivation. In addition, we want to point out to rule designers those rules that have been violated frequently and might require adaptations.

Natural-language constraints are hard to formalize, due to lack of a formal semantics of natural language. By using information retrieval techniques [WS99, MPG01, FGLM02, BHQW02, BQBW03], we could extract from natural-language texts relevant metadata and model them by semantic nets. This would support formalization of constraints like: “The summary of a chapter should cover the chapter’s main content,” or “The entity relationship model is consistent with its description.” The open architecture of our system supports integration of such approaches.

15.3 Applications

Applications of this thesis are manifold: Currently, we evaluate our consistency management approach at personell reviews in the Federal Armed Forces. At sd&m, the analysis module approach is extended towards construction modules. We are about to apply consistency maintenance to construction modules, which also calls for consistency requirements between analysis modules and their corresponding construction modules. Currently, the model driven architecture (MDA), a standard by the OMG, receives much attention in the software engineering community [KWB03]. Although OCL is proposed to formalize constraints for MDA [WK03], we are confident that our approach can help to manage consistency between the various models of MDA and other parts of a software specification. Component-based approaches to software engineering [HC01, GTW03] as well as bottom-up chip design techniques [BL00] could benefit from our tolerant consistency model as well. We also consider applications in the documentation of simulation experiments and hardware/software codesign [Buc01]. In addition, we plan to integrate consistency management as a means of quality control into long-term preservation approaches of digital artifacts [BRSS03].

In order to apply formal consistency management, we propose the following steps:

1. Identify the document kinds that are part of the development process. What are their goals and scopes? Who should edit which kind of document? In order to ease version management, some document kinds might require splitting. Notice that in the first step we neglect document structures.

2. Explore informal consistency requirements within and between documents. Also, investigate the document structures.

3. Define document structures and documents types. Develop languages and formalize consistency rules.

4. Localize consistency rules and optimize document structures.

We expect that steps (1) and (2) take long time; they give, however, precious insights to the consistency requirements actually needed. Steps (3) and (4) cover technical details and are, therefore, subject to experts.

In this thesis, we have developed a flexible formal consistency management approach for informal applications. We are confident that our work is a significant step towards consistency management in document engineering and provides a good basis for interesting, challenging, and useful future research. The ideas presented in this thesis are the basis for CDET (Consistent Document Engineering Toolkit). For up-to-date information, please consult the project WWW site

www2-data.informatik.unibw-muenchen.de/cde.html

Appendix A
Notation

Symbol	Description	Definition (page)
$\wp(s)$	Power set of the set s	
$\mathbb{B}$	Boolean values $\{\mathsf{True}, \mathsf{False}\}$	
$\mathbb{N}$	Natural numbers	
$\mathcal{X}$	Variables	28
$\mathcal{S}$	Symbols	28
$\mathcal{T}$	Terms	28
$\mathcal{F}$	Formulae	28
$\mathcal{F}_{\mathrm{at}}$	Atomic formulae	28
$\mathcal{F}^{\mathcal{H}}$	Formulae annotated by hints	102
$\mathcal{H}$	Hints	102
$\mathcal{M}$	Variable modes $\{\mathsf{Add}, \mathsf{Chg}, \mathsf{Del}, \mathsf{KEEP}\}$	137
Σ	Signatures	31
$\mathbb{V}$	Values	60
$\mathcal{X} \times \mathbb{V}$	Bindings of variables to values	61
$\mathbb{E}$	Variable assignments	61
$\mathbb{E}_{\mathrm{inc}}$	Incremental variable assignments	80
$\mathbb{E}_{\mathrm{incDAG}}$	Incremental variable assignments for S-DAG generation	113
$\mathbb{A}$	Σ-algebras	56
$\mathbb{D}$	Diagnoses	61
$\mathbb{B} \times \wp(\mathbb{D})$	Consistency reports	61
$\mathbb{K}$	Variable kinds $\{\mathsf{new}, \mathsf{old}\}$	80
$\mathbb{S}$	Predicate suggestions	109
$\mathbb{G}$	S-DAGs	109
$\mathbb{C}$	Actions	109
$\mathbb{R}$	Repairs	134

Table A.1: Notation: syntax and semantics

Name	Type and Description	Definition (page)
$. \models_{.} .$	$: \mathbb{A} \times \mathbb{E} \times \mathcal{F} \rightarrow \mathbb{B}$ Classic truth value semantics	49
$\mathcal{R}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times \mathcal{F} \times \mathbb{E} \rightarrow \mathbb{B} \times \wp(\mathbb{D})$ Basic report generation	51
$\delta_{\mathbb{A}}(.)$	$: \mathbb{A} \rightarrow \mathbb{A}$ Algebra transition	60
$\mathcal{V}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times \mathcal{T} \times \mathbb{E} \rightarrow \mathbb{V}$ Term evaluation	64
$\mathcal{IR}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times \mathcal{F} \times \mathbb{E}_{\text{inc}} \rightarrow \mathbb{B} \times \wp(\mathbb{D})$ Incremental report generation	81
$\mathcal{IV}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times \mathcal{T} \times \mathbb{E}_{\text{inc}} \rightarrow \wp(\mathbb{V}) \times \wp(\mathbb{V}) \times \wp(\mathbb{V}) \times \wp(\mathbb{V})$ Incremental sphere evaluation	86
$\mathcal{D}^{.}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times (\wp(\mathcal{X}) \times \mathbb{B}) \times \mathcal{F}^{\mathcal{H}} \times \mathbb{E} \rightarrow \mathbb{G}$ Basic S-DAG generation	110
$\mathcal{ID}^{.}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times (\wp(\mathcal{X}) \times \mathbb{B}) \times \mathcal{F}^{\mathcal{H}} \times \mathbb{E}_{\text{incDAG}} \rightarrow \mathbb{G}$ Incremental S-DAG generation	114
$\mathcal{S}^{.}_{.} [\![.]\!]_{.}$	$: \mathbb{A} \times \wp(\mathcal{X}) \times (\wp(\wp(\mathcal{H})) \times \mathcal{F}_{at} \times \mathbb{B}) \times \mathbb{E} \rightarrow \wp(\wp(\mathbb{S}))$ Hint evaluation	118
$\mathcal{A}(.)$	$: \mathbb{G} \rightarrow \mathbb{G}$ S-DAG augmentation	126
$\mathcal{REP}(.)\ [.,.]$	$: \mathbb{G} \times \mathbb{E} \times \wp(\mathcal{M}) \rightarrow \wp(\wp(\mathbb{R}))$ Repair derivation	137

Table A.2: Notation: central algorithms

Appendix B
Implementation

In this chapter, we describe the implementation of our prototype consistency maintenance tool. For implementation, we use the purely functional programming language Haskell (see, e.g., [Dav92, Tho96, Bir98]). The full language report is published as [PJ03]. Functional programming languages are well suited for prototyping, because their programming style is close to denotational semantics. Therefore, our algorithms are easy to implement. We have chosen Haskell for the following reasons:

- By default, Haskell evaluates expressions lazily. Strictness annotations provide fair control about the strictness of functions and data types, which severely influences efficiency of our prototype.
- Language designers use Haskell for the definition of symbol semantics, in order to guarantee referential transparency. Thus, we have no programming language gap when executing user-defined functions and predicates.
- Recently, the Glasgow Haskell Compiler [GHC04] has been complemented by support for dynamic loading [Sjö02, Ste04]. Thus, we can compile symbol semantics on the fly and load the resulting object code while the consistency checker is running.
- Haskell's Foreign Function Interface [C^+03] facilitates the use of other pre-compiled libraries, e.g., natural-language parsers.

At first sight, Haskell's lack of subtyping support appears a major drawback. Recent approaches to add subtyping have shown that it is still unclear how subtyping can be combined with other language extensions present in GHC [Nor99, SPJ01, Maz03]. An implementation in the GHC appears too laborious a task. In our setting, however, subtyping can be simulated, because our type system is simpler than that of GHC.

Our prototype implementation differs from our theoretical framework in the handling of partial functions. First, Haskell evaluates expressions lazily (our value evaluation function $\mathcal{V}$ is strict); e.g., a function application $f(e)$ is not necessarily undefined if e is undefined. Second, in Haskell, we cannot "catch" undefinedness; our prototype fails if an expression is not defined.

In the following sections, we describe key parts of our current prototype implementation. In Sect. B.1, we show how a consistency check is performed after a check-in to a DMS. In Sect. B.2, we describe a generic repository interface to arbitrary DMSs. Sect. B.3 is dedicated to the simulation of subtyping. In Sect. B.4, we outline how the semantics of function and predicate symbols is defined in Haskell. Notice that details are subject to rapid change. For up-to-date information, consult the project WWW site

www2-data.informatik.unibw-muenchen.de/cde.html.

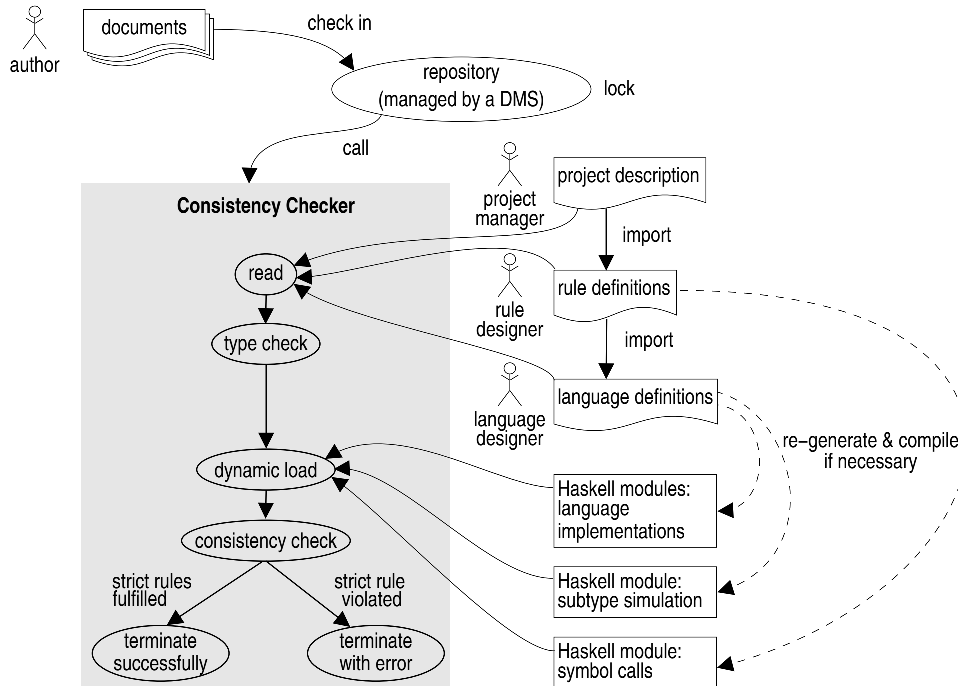


Figure B.1: Performing a consistency check in practice

B.1 Consistency Checking in Practice

Fig. B.1 shows how a consistency check is performed at a check-in to a consistency-aware DMS. First, the DMS completes some tests to ensure that the changes submitted may be applied to the repository. If these tests succeed, the DMS calls our consistency checker.[1] During consistency checking, the repository is locked, i.e., any check-ins are prohibited. If all strict rules are satisfied, our consistency checker terminates successfully and the DMS acknowledges the check-in. Otherwise, the check-in is rejected.

Our consistency checker reads the rules to check from the project description supplied by the project manager. Rule definitions from the rule designer import languages. The next step is to type-check the rules against the functions and predicates they use. Typically, rules and languages used do not change between consistency checks; then we dynamically link auxiliary Haskell modules to the running consistency checker (these modules have been compiled during a previous consistency check). Now, we can use the functions and predicates defined by the language designer, in order to check the repository for consistency. If, however, a language has changed since the previous consistency check, we re-compile this language with the help of GHC, which also type-checks the implementations of functions and predicates. All this is performed in the background.

[1] A simple system call is sufficient here. Recent revision control system support pre-commit hook scripts, which are executed automatically prior to storing a check-in [CSFP04, Rou04].

Dynamically loaded modules contain (1) implementations of functions, predicates, and types defined by the language designer, (2) auxiliary Haskell functions for simulating subtyping, and (3) two auxiliary Haskell functions actually used by our consistency checker. The latter Haskell functions are necessary, because a-priori our consistency checker cannot “know” the symbols defined in languages. Therefore, it uses two Haskell functions, in order to calculate the result value of a term or an atomic formula:

```
functions  :: Repository -> SymbolName -> [Value] -> Value
predicates :: Repository -> SymbolName -> [Value] -> Bool
```

Each Haskell function above maps the name of a symbol to its actual implementation and applies it to the argument values given as third argument. The first argument carries auxiliary information about the repository used. If a rule has changed since the previous consistency check, we have to re-generate this module.

B.2 Accessing Repositories

In this section, we describe a generic interface to the repository of an arbitrary DMS. As already mentioned, we make only few assumptions about the underlying DMS or revision control system. We have developed a simple repository interface, which must be instantiated for a specific DMS. The interface consists of six Haskell functions (the Haskell data type `Repository` represents the repository itself):

- `repStates :: Repository -> [State]` returns all states of a given repository.
- `repHead :: Repository -> State` returns the current repository state.
- `repDocs :: Repository -> State -> String -> [Doc]` returns the documents in the repository that are current at a given state and match a regular expression. For example, `repDocs repo 2 "*.xml"` returns all XML documents current at state 2 in the repository `repo`. Notice that the returned documents only include the name and the last modification state (up to the given state). The content of the document has to be parsed by the function `parseDoc` described below.
- `repDirs :: Repository -> State -> String -> [Dir]` behaves like `repDocs`, but returns directories instead. Similar to documents, directories include the name and the last modification state only.
- `parseDoc :: Repository -> Doc -> (String -> a) -> Maybe a` accesses a given document in the repository. The third parameter is a function that parses the document content and converts it to an appropriate Haskell data structure. For XML documents, such parser functions can be generated [WR99]. If the given document does not exist in the repository, `parseDoc` returns `Nothing`.

- `repChangesSince :: ClockTime -> Repository -> [String]` returns a list of document names and directory names that have been modified since the previous evaluation, determined by the first parameter.

Except for `repChangesSince` all interface functions above can be used by the language designer. The interface function `repChangesSince` is needed for filtering consistency rules.

For each supported DMS, the Haskell data type `Repository` includes an extra alternative, which stores auxiliary information. Currently, we support a simple directory structure, where each state is stored in an extra directory (as done in some open source DMS [Rön03]) and the revision control system DARCS [Rou04]:[2]

```
data Repository = DIR   DirDMS   -- simple directory structure
                | DARCS DarcsDMS -- DARCS repository
```

Accessing a directory structure is simple; we only need the base directory:

```
newtype DirDMS = DirDMS {baseDir :: String}
```

For accessing a DARCS repository, we define a more complex Haskell data type containing extra labels necessary for speeding up repository access.

```
data DarcsDMS = DarcsDMS {base      :: String,
                          states    :: FiniteMap State PatchInfo,
                          times     :: FiniteMap PatchInfo State,
                          testDir   :: String,
                          headState :: State,
                          final     :: Maybe Patch}
```

Above, `base` denotes the repository base directory, `states` maps a repository state to its associated patch, `times` is the reverse mapping of `states`, `testDir` denotes the directory in which the consistency check takes place (for consistency checking DARCS copies the current repository to a special directory), `headState` carries the current repository state, and `final` denotes changes made by the current check-in (which will be recorded, if our consistency checker terminates successfully). In DARCS, a `Patch` formally describes a check-in to the repository (where a document is modelled by a sequence of lines). A `PatchInfo` provides auxiliary information about a `Patch`, e.g., its file name stored on disk.

B.3 Haskell Meets Subtypes

Haskell lacks subtyping. Since functions and predicates are implemented in Haskell, we have to simulate subtyping. We resolve subtype relationships by the following measures:

- From each type declaration from the language designer, we generate a Haskell data type declaration.
- We coerce Haskell types to their supertypes. For Haskell record types, this means to drop fields.

[2] We are about to implement an interface to subversion [CSFP04].

- We marshal values needed by our prototype to Haskell values to which function and predicate implementations are applied. The resulting Haskell values are marshalled back to values needed by our prototype.

For each type, declared in a language, our prototype generates a Haskell data type declaration. For a record type, the generated Haskell data type contains all record labels including those of all supertypes. For a variant type, the generated Haskell data type contains all alternatives including those of all subtypes. In order to avoid name conflicts, Haskell named fields and Haskell data constructors are annotated by their type. For example, for the type declarations

$$\begin{array}{lcl} \texttt{Doc} & = & \{\texttt{dId} : \texttt{String}, \texttt{dState} : \texttt{State}\} \\ \texttt{DocSDM} < \{\texttt{Doc}\} & = & \{\texttt{valid} : \texttt{Bool}, \texttt{status} : \texttt{Status}, \ldots\} \\ \texttt{Status} & = & \texttt{InProgress} \mid \texttt{QualAssurance} \mid \texttt{Finished} \end{array}$$

our system generates the following Haskell data type declarations:

```
data Doc     = Doc {doc_dId    :: String,
                    doc_dState :: State}
  deriving (Eq, Ord)

data DocSDM  = DocSDM {docSDM_dId    :: String, -- inherited from Doc
                       docSDM_dState :: State,  -- inherited from Doc
                       docSDM_valid  :: Bool,
                       docSDM_status :: Status,
                       ...}
  deriving (Eq, Ord)

data Status  = Status_InProgress | Status_QualAssurance
             | Status_Finished
  deriving (Eq, Ord)
```

Instances for the type classes `Eq` and `Ord` are necessary, because equality and ordering are fully polymorphic relations in our setting.

We implement subtype coercion by a two parameter type class. The class member `coerce` converts a type `sub` to its supertype `super`.

```
class Coerce sub super where
 coerce :: sub -> super
```

Our system generates instances for the type class `Coerce`. The instances below are generated for the type declaration of `DocSDM`. All labels except `dId` and `dState` are dropped when coercing a Haskell value of type `DocSDM` to a Haskell value of type `Doc`. Since the subtyping relation is reflexive, we also generate a reflexive instance for the type `DocSDM`.

```
instance Coerce DocSDM Doc where
 coerce rec = Doc {doc_dId    = coerce (docSDM_dId rec),
                   doc_dState = coerce (docSDM_dState rec)}

instance Coerce DocSDM DocSDM where
 coerce = id
```

Instance definitions for parameterized data types are straightforward. We need an extra instance for function types, because we support higher-order symbols. The below instance definition may be read: If `aSub` is a subtype of `a` and `bSub` is a subtype of `b`, then `a -> bSub` is a subtype of `aSub -> b`. Notice the covariance of the function result type and the contravariance of the function argument type. Since in Haskell the arrow type `->` associates to the right we need one instance for function types only.

```
instance (Coerce aSub a, Coerce bSub b) =>
         Coerce (a -> bSub) (aSub -> b) where
 coerce f x = coerce (f (coerce x))
```

Finally, we marshal values needed by our prototype to Haskell values and vice versa. Our prototype generates instances for the following type class. The class member `toVal` converts a Haskell value to a value needed by our prototype; `fromVal` performs the reverse conversion, i.e., `toVal . fromVal == id`.

```
class HasValue a where
 toVal   :: a -> Value -- convert Haskell value to internal value
 fromVal :: Value -> a -- convert internal value to Haskell value
```

Our internal `Value` type includes atomic values, record values, variant values, and function values:

```
data Value = VAt String                    -- atomic value (generic)
           | VRec (FiniteMap String Value) -- record value mapping
                                           -- labels to values
           | VVar String [Value]
                   -- variant value (constructor name, argument values)
           | VFun String                   -- function value (name)
```

For the type declaration of `DocSDM`, our system generates the following marshalling instance:

```
instance HasValue DocSDM where
  toVal c = VRec fm
    where fm = listToFM (zip ["dId","dState","status","valid",...]
                             [toVal (docSDM_dId c),
                              toVal (docSDM_dState c),
                              toVal (docSDM_status c),
                              toVal (docSDM_valid c),
                              ...])
  fromVal (VRec fm) = DocSDM {docSDM_valid  = conv "valid",
                              docSDM_status = conv "status",
                              docSDM_dId    = conv "dId",
                              docSDM_dState = conv "dState",
                              ...}
    where conv = fromVal . fromJust . lookupFM fm
```

Our system uses the above coercion and marshalling instances when generating the Haskell module for symbol calls, which maps symbol names to their implementation.

One might wonder that we have developed an own approach to subtyping, instead of using approaches already present. Shields and Peyton-Jones propose an alternative approach towards subtyping in Haskell [SPJ01], which lacks, however, support for parameterized data types. Extensible records [JPJ99], as implemented in the Haskell interpreter Hugs [Jon04], lack support for subtyping variant types. Also, extensible records did as yet not find their way into

the GHC. Implementing Nordlander style subtyping in GHC appeared a complex task because of non-trivial interaction of subtyping with other language extensions present in GHC [Maz03].

B.4 Defining Symbol Semantics

In this section, we exemplify how the language designer defines symbol semantics in Haskell. In our current implementation, this is a rather complex matter, because it requires deep knowledge about the programming language Haskell, the XML parser HaXml, and the way our prototype resolves subtype relationships. Consider the implementation of the function symbol `repDocSDM` : `State` → `DocSDM`, which returns all sd&m documents in the repository. For the implementation, we need the following:

- Access to the repository, in order to get the names of all sd&m documents. This is achieved by the repository access function `repDocs`.
- A parser function of type `String -> DocSDM`, which converts the document content to a Haskell data structure of type `DocSDM`. This parser function serves as a parameter for the repository access function `parseDoc`.

For XML documents, HaXml can derive from a DTD corresponding Haskell data type declarations and appropriate parser functions. HaXml lacks, however, subtyping support. Since `DocSDM` is no real document type (it has no corresponding DTD) but only a supertype of all sd&m document types, HaXml is not applicable in first place. In contrast, for concrete sd&m document types (which have a corresponding DTD) we can derive appropriate Haskell types and parser functions.

Assume we have already defined the other document access functions. Then we can define `repDocSDM` by using subtype coercion as follows (the first argument `repo` stands for the repository):

```
repDocSDM repo t = specs ++ anafuns ++ busprocs ++ crosscuts ++
                   datamodels ++ datatypes ++ dialogs ++ glossaries ++
                   goalsscopes ++ readinginstrs ++ usecases
 where specs         = map coerce (repSp repo t)
       anafuns       = map coerce (repAnaFun repo t)
       busprocs      = map coerce (repBusProc repo t)
       crosscuts     = map coerce (repCrosscut repo t)
       datamodels    = map coerce (repDataMod repo t)
       datatypes     = map coerce (repDataTs repo t)
       dialogs       = map coerce (repDialog repo t)
       glossaries    = map coerce (repGlossary repo t)
       goalsscopes   = map coerce (repGoalsscope repo t)
       readinginstrs = map coerce (repReadingInst repo t)
       usecases      = map coerce (repUsecase repo t)
```

Next, we discuss implementation of the function `repSp` : State → `DocSpec`, which returns all specification documents in the repository. Implementations of other repository access functions are similar. For the document type `DocSpec`, HaXml generates (among others) the following Haskell data type declarations:[3]

```
data DocSpec      = DocSpec Head Readers Kind Modules

data Head         = Head Status ...

data Status       = Status Status_Attrs String

data Status_Attrs = Status_Attrs {statusValid :: (Maybe Status_valid)}

data Status_valid = Status_valid_yes  |  Status_valid_no

newtype Readers   = Readers String

data Kind         = Kind  { kindPhase :: Kind_phase }

data Kind_phase   = Kind_phase_study  |  Kind_phase_coarse  |
                    Kind_phase_fine

data Modules      = Modules Essentials Misc Functional Data ...

data Essentials   = Essentials [Doc_crosscut] ...

data Misc         = Misc       Doc_reading ...

data Functional   = Functional [Doc_busproc] ...

data Data         = Data       (Maybe Doc_datamod) ...
...
newtype Doc_crosscut = Doc_crosscut String
newtype Doc_reading  = Doc_reading  String
newtype Doc_busproc  = Doc_busproc  String
newtype Doc_datamod  = Doc_datamod  String
...
```

In addition, for the data type `DocSpec`, HaXml derives an instance of the type class `XmlContent`:

```
class XmlContent a where
 readXml :: String -> Maybe a -- convert XML String to Haskell value
 showXml :: a      -> String  -- convert Haskell value to XML String
```

We can use the class member `readXml` as a parser argument for the repository access function `parseDoc`. The above Haskell data type is, however, different from the Haskell data type generated from the declaration of `DocSDM` in the language SDM:

```
data DocSpec  = DocSpec {docSpec_dId          :: String,
                         docSpec_dState       :: State,
                         docSpec_doc_crosscut :: [String],
                         docSpec_doc_busproc  :: [String],
                         docSpec_doc_datamod  :: Maybe String,
```

[3]For brevity, data types are simplified; declarations for derived instances are omitted.

```
                         docSpec_doc_reading  :: String,
                         docSpec_specKind     :: SpecKind,
                         docSpec_specReaders  :: String,
                         docSpec_status       :: Status,
                         docSpec_valid        :: Bool, ...}
```

Therefore, the language designer must convert the type `HaXml.DocSpec` (generated by HaXml) to `DocSpec` (generated by our prototype). Thus, the function `repSp` can be implemented as follows:

```
repSp repo t = map (\ d -> mkSpec d $ fromJust $ fromJust $
                      parseDoc d repo readXml) ds
 where ds = repDocs "Spec*.xml" t repo
       mkSpec :: Doc -> HaXml.DocSpec -> DocSpec
       mkSpec d (HaXml.DocSpec spHead (Readers readers) spKind
                       (Modules (Essentials crosscuts ... )
                                (Misc (Doc_reading reading) ...)
                                (Functional busprocs ... )
                                (Data datamod ...)
                                ...))
         = DocSpec
            {docSpec_dId         = doc_dId d,
             docSpec_dState      = doc_dState d,
             docSpec_valid       = val,
             docSpec_status      = st,
             docSpec_specReaders = readers,
             docSpec_specKind    = kind,
             docSpec_doc_crosscut = map  (\ (Doc_crosscut s) -> s)
                                        crosscuts,
             docSpec_doc_busproc  = map  (\ (Doc_busproc s) -> s)
                                        busprocs,
             docSpec_doc_datamod  = fmap (\ (Doc_datamod s) -> s)
                                        datamod,
             docSpec_doc_reading  = reading, ...}
           where (val,st) = mkHead spHead
                 kind     = case kindPhase spKind of
                              Kind_phase_study  -> SpecKind_Study
                              Kind_phase_coarse -> SpecKind_Coarse
                              Kind_phase_fine   -> SpecKind_Fine

mkHead :: Head -> (Bool,Status)
mkHead (Head (Status stAttrs st) ...)
 = (val,st')
   where val = case statusValid stAttrs of
                 Just Status_valid_yes -> True
                 _                         -> False
         st' = case st of
                 "QualAssurance" -> Status_QualAssurance
                 "Finished"      -> Status_Finished
                 _               -> Status_InProgress
```

We see that defining document parsers is significantly alleviated through subtyping and the use of HaXml. Our current implementation suffers, however,

from lack of subtyping support in Haskell and HaXml. But, since language designers define languages rarely and re-use them in multiple projects, we regard this drawback a minor issue only.

Appendix C
Proofs

In this chapter, we prove the theorems 5.1, 5.2, 5.3, and 8.1. The definition of the validity relation can be found in Fig. 5.3 (pg. 49). The definition of report generation is given in Fig. 5.4 (pg. 51); for auxiliary functions see Fig. 5.10 (pg. 63). The definition of S-DAG generation is given in Fig. 8.12 (pg. 110); for auxiliary functions see Fig. 8.21 (pg. 121).

Proof C.1 (Generated consistency reports are sound) Let ϕ be a formula, A a first-order structure, η a variable assignment, and ds a set of diagnoses. Then we have:

$$(\mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{False}, ds) \Rightarrow ds \neq \emptyset \ \wedge \ d = (\mathsf{IC}, _, _, _) \text{ for every } d \in ds) \ \wedge$$
$$(\mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{True}, ds) \ \Rightarrow ds \neq \emptyset \ \wedge \ d = (\mathsf{C}, _, _, _) \text{ for every } d \in ds)$$

Proof: The proof proceeds by induction on the structure of the formula ϕ as follows (for simplicity, we neglect undefined values):

- Let $\phi \equiv p(e_1, \ldots, e_n)$:

  $$\begin{aligned}\mathcal{R}_A[\![p(e_1,\ldots,e_n)]\!]\eta = \ & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{p(e_1,\ldots,e_n)\}, \emptyset)\}) \\ & \quad \text{if } (e_1^A,\ldots,e_n^A) \in p^A \\ & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{p(e_1,\ldots,e_n)\})\}) \\ & \quad \text{otherwise}\end{aligned}$$

  Thus, the theorem follows directly from $\mathcal{R}$'s definition.

- Let $\phi \equiv \neg\psi$:

  $$\mathcal{R}_A[\![\neg\psi]\!]\eta = \overline{\mathsf{flip}}(\mathcal{R}_A[\![\psi]\!]\eta)$$

  The induction hypothesis implies the theorem for ψ. By applying $\overline{\mathsf{lift}}$ to the report for ψ we do not loose any diagnoses, because $\overline{\mathsf{lift}}$ applies the function flip to every diagnosis; in addition, the truth value of the report is inverted. flip inverts a diagnosis. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \psi_1 \wedge \psi_2$:

  $$\begin{aligned}\mathcal{R}_A[\![\psi_1 \wedge \psi_2]\!]\eta = \ & r_{\psi_1} \otimes r_{\psi_2} && \text{if } \mathsf{fst}(r_{\psi_1}) = \mathsf{fst}(r_{\psi_2}) \\ & r_{\psi_1} && \text{else if } \mathsf{fst}(r_{\psi_1}) = \mathsf{False} \\ & r_{\psi_2} && \text{else if } \mathsf{fst}(r_{\psi_2}) = \mathsf{False} \\ \text{where } & r_{\psi_1} = \mathcal{R}_A[\![\psi_1]\!]\eta \\ & r_{\psi_2} = \mathcal{R}_A[\![\psi_2]\!]\eta\end{aligned}$$

  If the reports r_{ψ_1} and r_{ψ_2} have different truth values, the theorem holds by the induction hypothesis. If both reports are True, then their diagnoses contain consistent diagnoses only. Applying $\otimes$ to the reports results in a True report, only carrying consistent diagnoses, because $\otimes$ applies the function join to every possible pair of diagnoses. join retains the consistency flag. If both reports are False, proceed similarly. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \psi_1 \vee \psi_2$:

$$\begin{array}{rcll} \mathcal{R}_A[\![\psi_1 \vee \psi_2]\!]\eta & = & r_{\psi_1} \oplus r_{\psi_2} & \text{if } \mathsf{fst}(r_{\psi_1}) = \mathsf{fst}(r_{\psi_2}) \\ & & r_{\psi_1} & \text{else if } \mathsf{fst}(r_{\psi_1}) = \mathsf{True} \\ & & r_{\psi_2} & \text{else if } \mathsf{fst}(r_{\psi_2}) = \mathsf{True} \\ \text{where } r_{\psi_1} & = & \mathcal{R}_A[\![\psi_1]\!]\eta & \\ r_{\psi_2} & = & \mathcal{R}_A[\![\psi_2]\!]\eta & \end{array}$$

If the reports r_{ψ_1} and r_{ψ_2} have different truth values, the theorem holds by the induction hypothesis. If both reports are True, then their diagnoses contain consistent diagnoses only. Applying $\oplus$ to the reports results in a True report, only carrying consistent diagnoses, because $\oplus$ applies the function condense to the union of the diagnoses from both reports. condense combines diagnoses that contain the same variable assignment by join. If both reports are False, proceed similarly. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \psi_1 \Rightarrow \psi_2$:

$$\mathcal{R}_A[\![\psi_1 \Rightarrow \psi_2]\!]\eta = \mathcal{R}_A[\![\neg\psi_1 \vee \psi_2]\!]\eta$$

The theorem holds by the induction hypothesis.

- Let $\phi \equiv \forall\, x \in e \bullet \psi$:

$$\begin{array}{rcll} \mathcal{R}_A[\![\forall\, x \in e \bullet \psi]\!]\eta & = & \overline{\oplus}(F) & \text{if } F \neq \emptyset \\ & & \overline{\mathsf{min}}(T) & \text{else if } T \neq \emptyset \\ & & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{\mathtt{null}(e)\}, \emptyset)\}) & \text{otherwise} \\ \text{where } rs & = & \{(v, \mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\})) \mid v \in \mathcal{V}_A[\![e]\!]\eta\} & \\ F & = & \{\mathsf{push}(x \mapsto v, r) \mid (v, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\} & \\ T & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} & \end{array}$$

By definition, all reports in F are False. Due to the induction hypothesis, they contain non-empty sets of inconsistent diagnoses. If F is not empty, then $\overline{\oplus}(F)$ results in a False report containing inconsistent diagnoses only. $\overline{\oplus}$ folds $\oplus$ over a set. If F is empty but T is not, then $\overline{\mathsf{min}}(T)$ results in a True report carrying a non-empty set of consistent diagnoses. By definition, all reports in T are True. $\overline{\mathsf{min}}(T)$ applies min to the conjunction of all reports in T. min returns the greatest lower bounds w.r.t. the diagnosis ordering $\sqsubseteq$. The set returned by min is not empty, the consistency flags are not modified. If both F and T are empty, then the theorem holds by definition of $\mathcal{R}$. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \exists\, x \in e \bullet \psi$:

$$\begin{array}{rcll} \mathcal{R}_A[\![\exists\, x \in e \bullet \psi]\!]\eta & = & \overline{\oplus}(T) & \text{if } T \neq \emptyset \\ & & \overline{\mathsf{min}}(F) & \text{else if } F \neq \emptyset \\ & & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \{\mathtt{null}(e)\}, \emptyset)\}) & \text{otherwise} \\ \text{where } rs & = & \{(v, \mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\})) \mid v \in \mathcal{V}_A[\![e]\!]\eta\} & \\ T & = & \{\mathsf{push}(x \mapsto v, r) \mid (v, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} & \\ F & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\} & \end{array}$$

Proceed as above but with F and T reversed.

Due to the above induction, the theorem holds for all formulae ϕ. □

Proof C.2 (Reasons for inconsistencies are sound) Let ϕ be a formula, A a first-order structure, η a variable assignment, ds a diagnoses set, and ps_t and ps_f sets of atomic formulae. Then we have:

$$\begin{array}{lcl} \mathcal{R}'_A[\![\phi]\!]\eta = (_, ds) \wedge & \Rightarrow & A \models_\eta \phi' \text{ for every } \phi' \text{ in } ps_t \wedge \\ (_, \eta, ps_t, ps_f) \in ds & & A \models_\eta \neg\phi' \text{ for every } \phi' \text{ in } ps_f \end{array}$$

The report generation function $\mathcal{R}'$ deviates from $\mathcal{R}$ as follows: For an atomic formula, we push the complete variable assignment into the resulting diagnosis. For quantified formulae, we do not push bindings into the diagnoses. This is necessary in order to ensure that η contains all free variables of the predicate sets ps_t and ps_f.

Proof: The proof proceeds by induction on the structure of the formula ϕ as follows (for simplicity, we neglect undefined values):

- Let $\phi \equiv p(e_1, \ldots, e_n)$:

  $$\begin{array}{rcl} \mathcal{R}'_A[\![p(e_1, \ldots, e_n)]\!]\eta & = & (\mathsf{True}, \{(\mathsf{C}, \eta, \{p(e_1, \ldots, e_n)\}, \emptyset)\}) \\ & & \quad \text{if } (e_1^A, \ldots, e_n^A) \in p^A \\ & & (\mathsf{False}, \{(\mathsf{IC}, \eta, \emptyset, \{p(e_1, \ldots, e_n)\})\}) \\ & & \quad \text{otherwise} \end{array}$$

  If $(e_1^A, \ldots, e_n^A) \in p^A$, then we have $A \models_\eta p(e_1, \ldots, e_n)$ due to the definition of $\models$. Otherwise, the definition of $\models$ immediately implies $A \models_\eta \neg\, p(e_1, \ldots, e_n)$. Thus, the theorem holds for atomic formulae.

- Let $\phi \equiv \neg\psi$:

  $$\mathcal{R}'_A[\![\neg\psi]\!]\eta = \overline{\mathsf{flip}}(\mathcal{R}'_A[\![\psi]\!]\eta)$$

  The induction hypothesis implies the theorem for ψ. Application of $\overline{\mathsf{lift}}$ to the report does not modify the predicate sets. Only the report's truth value and consistency flags are inverted. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \psi_1 \wedge \psi_2$:

  $$\begin{array}{rcll} \mathcal{R}'_A[\![\psi_1 \wedge \psi_2]\!]\eta & = & r_{\psi_1} \otimes r_{\psi_2} & \text{if } \mathsf{fst}(r_{\psi_1}) = \mathsf{fst}(r_{\psi_2}) \\ & & r_{\psi_1} & \text{else if } \mathsf{fst}(r_{\psi_1}) = \mathsf{False} \\ & & r_{\psi_2} & \text{else if } \mathsf{fst}(r_{\psi_2}) = \mathsf{False} \\ \text{where } r_{\psi_1} & = & \mathcal{R}'_A[\![\psi_1]\!]\eta & \\ r_{\psi_2} & = & \mathcal{R}'_A[\![\psi_2]\!]\eta & \end{array}$$

  If the reports r_{ψ_1} and r_{ψ_2} have different truth values, the theorem holds by the induction hypothesis. Consider the case where both reports have the same truth value. The operator $\otimes$ applies the function join to every possible pair of diagnoses. join computes the union of the corresponding sets of atomic formulae, i.e., fulfilled (violated) atomic formulae of the first diagnoses are joined with fulfilled (violated) atomic formulae of the second diagnosis. condense does not modify the sets of atomic formulae. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \psi_1 \vee \psi_2$:

$$\begin{array}{rcll} \mathcal{R}'_A[\![\psi_1 \vee \psi_2]\!]\eta & = & r_{\psi_1} \oplus r_{\psi_2} & \text{if } \mathsf{fst}(r_{\psi_1}) = \mathsf{fst}(r_{\psi_2}) \\ & & r_{\psi_1} & \text{else if } \mathsf{fst}(r_{\psi_1}) = \mathsf{True} \\ & & r_{\psi_2} & \text{else if } \mathsf{fst}(r_{\psi_2}) = \mathsf{True} \\ \text{where } r_{\psi_1} & = & \mathcal{R}'_A[\![\psi_1]\!]\eta & \\ r_{\psi_2} & = & \mathcal{R}'_A[\![\psi_2]\!]\eta & \end{array}$$

  If the reports r_{ψ_1} and r_{ψ_2} have different truth values, the theorem holds by the induction hypothesis. Consider the case where both reports have the same truth value. The operator $\oplus$ applies the function condense to the union of the diagnoses of both reports, which does not modify the sets of atomic formulae. Thus, the theorem holds for ϕ.

- Let $\phi \equiv \psi_1 \Rightarrow \psi_2$:

$$\mathcal{R}'_A[\![\psi_1 \Rightarrow \psi_2]\!]\eta = \mathcal{R}'_A[\![\neg\psi_1 \vee \psi_2]\!]\eta$$

  The theorem holds by the induction hypothesis.

- Let $\phi \equiv \forall\, x \in e \bullet \psi$:

$$\begin{array}{rcll} \mathcal{R}'_A[\![\forall\, x \in e \bullet \psi]\!]\eta & = & \overline{\oplus}(F) & \text{if } F \neq \emptyset \\ & & \overline{\mathsf{min}}(T) & \text{else if } T \neq \emptyset \\ & & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{\mathtt{null}(e)\}, \emptyset)\}) & \text{otherwise} \\ \text{where } rs & = & \{(v, \mathcal{R}'_A[\![\psi]\!](\eta \cup \{x \mapsto v\})) \mid v \in \mathcal{V}_A[\![e]\!]\eta\} & \\ F & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\} & \\ T & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} & \end{array}$$

  For the reports in T and F, respectively, the theorem holds by the induction hypothesis. Applying $\overline{\oplus}$ to F does not change the sets of atomic formulae in F's diagnoses, because $\overline{\oplus}$ folds $\oplus$ over F. Applying $\overline{\mathsf{min}}$ to T employs the function min, which retains the sets of atomic formulae. If both F and T are empty, then we have $\mathcal{V}_A[\![e]\!]\eta = []$. Due to the interpretation of the predicate symbol null (see Fig. 5.7, pg. 60), we have $[] \in \mathtt{null}^A$ and, hence, $A \models_\eta \mathtt{null}(e)$ (due to the definition of $\models$). Thus, the theorem holds for ϕ.

- Let $\phi \equiv \exists\, x \in e \bullet \psi$:

$$\begin{array}{rcll} \mathcal{R}'_A[\![\exists\, x \in e \bullet \psi]\!]\eta & = & \overline{\oplus}(T) & \text{if } T \neq \emptyset \\ & & \overline{\mathsf{min}}(F) & \text{else if } F \neq \emptyset \\ & & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \{\mathtt{null}(e)\}, \emptyset)\}) & \text{otherwise} \\ \text{where } rs & = & \{(v, \mathcal{R}'_A[\![\psi]\!](\eta \cup \{x \mapsto v\})) \mid v \in \mathcal{V}_A[\![e]\!]\eta\} & \\ T & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} & \\ F & = & \{r \mid (_, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\} & \end{array}$$

  Proceed as above but with F and T reversed.

Due to the above induction, the theorem holds for all formulae ϕ. □

Proof C.3 (Consistency reports indicate real inconsistencies) Let ϕ be a formula, A a first-order structure, and η a variable assignment. Then we have:

$$\text{not } A \models_\eta \phi \iff \exists\ ds \in \wp(\mathbb{D}) \bullet \mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{False}, ds) \quad \wedge$$

Proof: We prove the following equivalence:

$$A \models_\eta \phi \iff \mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{True}, _)$$

This implies Theorem 5.3. Again, the proof proceeds by straightforward induction on the structure of the formula ϕ. For the above equivalence, we prove two directions (for simplicity, we neglect undefined values):

$$\begin{array}{lll} (\Longrightarrow) & A \models_\eta \phi & \Longrightarrow \mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{True}, _) \\ (\Longleftarrow) & \mathcal{R}_A[\![\phi]\!]\eta = (\mathsf{True}, _) & \Longrightarrow A \models_\eta \phi \end{array}$$

- Let $\phi \equiv p(e_1, \dots, e_n)$:

$$\begin{array}{lll} (\Longrightarrow) & & \\ & A \models_\eta p(e_1, \dots, e_n) & \\ \Rightarrow & (e_1^A, \dots, e_n^A) \in p^A & \text{Def. } \models \\ \Rightarrow & \mathcal{R}_A[\![p(e_1, \dots, e_n)]\!]\eta = (\mathsf{True}, _) & \text{Def. } \mathcal{R} \end{array}$$

$$\begin{array}{lll} (\Longleftarrow) & & \\ & \mathcal{R}_A[\![p(e_1, \dots, e_n)]\!]\eta = (\mathsf{True}, _) & \\ \Rightarrow & (e_1^A, \dots, e_n^A) \in p^A & \text{Def. } \mathcal{R} \\ \Rightarrow & A \models_\eta p(e_1, \dots, e_n) & \text{Def. } \models \end{array}$$

- Let $\phi \equiv \neg\psi$:

$$\begin{array}{lll} (\Longrightarrow) & & \\ & A \models_\eta \neg\psi & \\ \Rightarrow & \text{not } A \models_\eta \psi & \text{Def. } \models \\ \Rightarrow & \mathcal{R}_A[\![\psi]\!]\eta = (\mathsf{False}, _) & \text{induction hypothesis} \\ \Rightarrow & \overline{\mathsf{flip}}(\mathcal{R}_A[\![\psi]\!]\eta) = (\mathsf{True}, _) & \text{Def. } \overline{\mathsf{flip}} \\ \Rightarrow & \mathcal{R}_A[\![\neg\psi]\!]\eta = (\mathsf{True}, _) & \text{Def. } \mathcal{R} \end{array}$$

$$\begin{array}{lll} (\Longleftarrow) & & \\ & \mathcal{R}_A[\![\neg\psi]\!]\eta = (\mathsf{True}, _) & \\ \Rightarrow & \overline{\mathsf{flip}}(\mathcal{R}_A[\![\psi]\!]\eta) = (\mathsf{True}, _) & \text{Def. } \mathcal{R} \\ \Rightarrow & \mathcal{R}_A[\![\psi]\!]\eta = (\mathsf{False}, _) & \text{Def. } \overline{\mathsf{flip}} \\ \Rightarrow & \text{not } A \models_\eta \psi & \text{induction hypothesis} \\ \Rightarrow & A \models_\eta \neg\psi & \text{Def. } \models \end{array}$$

- Let $\phi \equiv \psi_1 \wedge \psi_2$:

$(\Longrightarrow)$		
	$A \models_\eta \psi_1 \wedge \psi_2$	
$\Rightarrow$	$A \models_\eta \psi_1$ and $A \models_\eta \psi_2$	Def. $\models$
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1]\!]\eta = (\mathsf{True}, _)$ and $\mathcal{R}_A[\![\psi_2]\!]\eta = (\mathsf{True}, _)$	induction hypothesis
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1]\!]\eta \otimes \mathcal{R}_A[\![\psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\otimes$
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1 \wedge \psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$
$(\Longleftarrow)$		
	$\mathcal{R}_A[\![\psi_1 \wedge \psi_2]\!]\eta = (\mathsf{True}, _)$	
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1]\!]\eta \otimes \mathcal{R}_A[\![\psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1]\!]\eta = (\mathsf{True}, _)$ and $\mathcal{R}_A[\![\psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\otimes$
$\Rightarrow$	$A \models_\eta \psi_1$ and $A \models_\eta \psi_2$	induction hypothesis
$\Rightarrow$	$A \models_\eta \psi_1 \wedge \psi_2$	Def. $\models$

- Let $\phi \equiv \psi_1 \vee \psi_2$:

$(\Longrightarrow)$		
	$A \models_\eta \psi_1 \vee \psi_2$	
$\Rightarrow$	$A \models_\eta \psi_1$ or $A \models_\eta \psi_2$	Def. $\models$
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1]\!]\eta = (\mathsf{True}, _)$ or $\mathcal{R}_A[\![\psi_2]\!]\eta = (\mathsf{True}, _)$	induction hypothesis
case 1: both reports are True		
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1]\!]\eta \oplus \mathcal{R}_A[\![\psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\oplus$
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1 \vee \psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$
case 2: only one report is True		
$\Rightarrow$	$\mathcal{R}_A[\![\psi_1 \vee \psi_2]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$
$(\Longleftarrow)$		
	$\mathcal{R}_A[\![\psi_1 \vee \psi_2]\!]\eta = (\mathsf{True}, _)$	
case 1: both reports are True		
$\Rightarrow$	$A \models_\eta \psi_1$ and $A \models_\eta \psi_2$	induction hypothesis
$\Rightarrow$	$A \models_\eta \psi_1 \vee \psi_2$	Def. $\models$
case 2: only one report is True		
$\Rightarrow$	$A \models_\eta \psi_1$ or $A \models_\eta \psi_2$	induction hypothesis
$\Rightarrow$	$A \models_\eta \psi_1 \vee \psi_2$	Def. $\models$

The case that both reports are False cannot occur, due to the definition of $\oplus$.

- Let $\phi \equiv \psi_1 \Rightarrow \psi_2$: holds trivially by the induction hypothesis.

- Let $\phi \equiv \forall\, x \in e \bullet \psi$:

($\Longrightarrow$)

	$A \models_\eta \forall\, x \in e \bullet \psi$	
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. $\models$
$\Rightarrow$	$\mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\}) = (\mathsf{True}, _)$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
case 1: $\mathcal{V}_A[\![e]\!]\eta \neq []$		
$\Rightarrow$	$F = \emptyset$ and $T \neq \emptyset$ and $r = (\mathsf{True}, _)$ for all $r \in T$	Def. F, T in Def. $\mathcal{R}$
$\Rightarrow$	$\overline{\mathsf{min}}(T) = (\mathsf{True}, _)$	Def. $\overline{\mathsf{min}}$
$\Rightarrow$	$\mathcal{R}_A[\![\forall\, x \in e \bullet \psi]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$
case 2: $\mathcal{V}_A[\![e]\!]\eta = []$		
$\Rightarrow$	$F = \emptyset$ and $T = \emptyset$	Def. F, T in Def. $\mathcal{R}$
$\Rightarrow$	$\mathcal{R}_A[\![\forall\, x \in e \bullet \psi]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$

($\Longleftarrow$)

	$\mathcal{R}_A[\![\forall\, x \in e \bullet \psi]\!]\eta = (\mathsf{True}, _)$	
case 1: $\mathcal{V}_A[\![e]\!]\eta \neq []$		
$\Rightarrow$	$F = \emptyset$ and $T \neq \emptyset$ and $r = (\mathsf{True}, _)$ for all $r \in T$ and $\lvert T \rvert = \lvert \mathcal{V}_A[\![e]\!]\eta \rvert$	Def. $F, T, \mathcal{R}$
$\Rightarrow$	$\mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\}) = (\mathsf{True}, _)$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. T
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$A \models_\eta \forall\, x \in e \bullet \psi$	Def. $\models$
case 2: $\mathcal{V}_A[\![e]\!]\eta = []$		
$\Rightarrow$	$\mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\}) = (\mathsf{True}, _)$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	$\mathcal{V}_A[\![e]\!]\eta = []$
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$A \models_\eta \forall\, x \in e \bullet \psi$	Def. $\models$

- Let $\phi \equiv \exists\, x \in e \bullet \psi$:

($\Longrightarrow$)

	$A \models_\eta \exists\, x \in e \bullet \psi$	
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. $\models$
$\Rightarrow$	$\mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\}) = (\mathsf{True}, _)$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$T \neq \emptyset$ and $r = (\mathsf{True}, _)$ for all $r \in T$	Def. T in Def. $\mathcal{R}$
$\Rightarrow$	$\overline{\oplus}(T) = (\mathsf{True}, _)$	Def. $\overline{\oplus}$
$\Rightarrow$	$\mathcal{R}_A[\![\exists\, x \in e \bullet \psi]\!]\eta = (\mathsf{True}, _)$	Def. $\mathcal{R}$

($\Longleftarrow$)

	$\mathcal{R}_A[\![\exists\, x \in e \bullet \psi]\!]\eta = (\mathsf{True}, _)$	
$\Rightarrow$	$T \neq \emptyset$ and $r = (\mathsf{True}, _)$ for all $r \in T$	Def. T in Def. $\mathcal{R}$
$\Rightarrow$	$\mathcal{R}_A[\![\psi]\!](\eta \cup \{x \mapsto v\}) = (\mathsf{True}, _)$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. T in Def. $\mathcal{R}$
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$A \models_\eta \exists\, x \in e \bullet \psi$	Def. $\models$

Due to the above induction, the theorem holds for all formulae. □

Proof C.4 (S-DAGs indicate real inconsistencies) Let ϕ be a consistency rule and A a first-order structure. Then we have:

$$\text{not } A \models_\emptyset \phi \iff \mathcal{D}_A^{(_, \mathsf{False})}[\![\phi]\!]\emptyset \neq \bigcirc$$

Proof: We prove the following for variable assignments η and formulae ϕ that do not appear in a negated context. Due to miniscoping, we can handle negation as a special case. Then, the following equivalence implies the above theorem:

$$A \models_\eta \phi \iff \mathcal{D}_A^{(_,\mathsf{False})}[\![\phi]\!]\eta = \bigcirc$$

Due to miniscoping, only atomic formulae may appear in a negated context and implications are removed. Thus, the following induction on the structure of ϕ suffices.

- Let $\phi \equiv p(e_1, \ldots, e_n)$, where ϕ does not appear in a negated context:

$$\begin{array}{lll}
(\Longrightarrow) & & \\
 & A \models_\eta p(e_1, \ldots, e_n) & \\
\Rightarrow & (e_1^A, \ldots, e_n^A) \in p^A & \text{Def. } \models \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{False})}[\![p(e_1, \ldots, e_n)]\!]\eta = \bigcirc & \text{Def. } \mathcal{D}, b = \mathsf{True} \neq \mathsf{False} = neg
\end{array}$$

$$\begin{array}{lll}
(\Longleftarrow) & & \\
 & \mathcal{D}_A^{(_,\mathsf{False})}[\![p(e_1, \ldots, e_n)]\!]\eta = \bigcirc & \\
\Rightarrow & (e_1^A, \ldots, e_n^A) \in p^A & \text{Def. } \mathcal{D}, b = \mathsf{True} \neq \mathsf{False} = neg \\
\Rightarrow & A \models_\eta p(e_1, \ldots, e_n) & \text{Def. } \models
\end{array}$$

- Let $\phi \equiv \neg\, p(e_1, \ldots, e_n)$, where ϕ does not appear in a negated context:

$$\begin{array}{lll}
(\Longrightarrow) & & \\
 & A \models_\eta \neg\, p(e_1, \ldots, e_n) & \\
\Rightarrow & \text{not } A \models_\eta p(e_1, \ldots, e_n) & \text{Def. } \models \\
\Rightarrow & (e_1^A, \ldots, e_n^A) \notin p^A & \text{Def. } \models \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{True})}[\![p(e_1, \ldots, e_n)]\!]\eta = \bigcirc & \text{Def. } \mathcal{D}, b = \mathsf{False} \neq \mathsf{True} = neg \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{False})}[\![\neg\, p(e_1, \ldots, e_n)]\!]\eta = \bigcirc & \text{Def. } \mathcal{D}
\end{array}$$

$$\begin{array}{lll}
(\Longleftarrow) & & \\
 & \mathcal{D}_A^{(_,\mathsf{False})}[\![\neg\, p(e_1, \ldots, e_n)]\!]\eta = \bigcirc & \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{True})}[\![p(e_1, \ldots, e_n)]\!]\eta = \bigcirc & \text{Def. } \mathcal{D} \\
\Rightarrow & (e_1^A, \ldots, e_n^A) \notin p^A & \text{Def. } \mathcal{D}, b = \mathsf{False} \neq \mathsf{True} = neg \\
\Rightarrow & \text{not } A \models_\eta p(e_1, \ldots, e_n) & \text{Def. } \models \\
\Rightarrow & A \models_\eta \neg\, p(e_1, \ldots, e_n) & \text{Def. } \models
\end{array}$$

- Let $\phi \equiv \psi_1 \wedge \psi_2$:

$$\begin{array}{lll}
(\Longrightarrow) & & \\
 & A \models_\eta \psi_1 \wedge \psi_2 & \\
\Rightarrow & A \models_\eta \psi_1 \text{ and } A \models_\eta \psi_2 & \text{Def. } \models \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1]\!]\eta = \bigcirc \text{ and } \mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_2]\!]\eta = \bigcirc & \text{induction hypothesis} \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1 \wedge \psi_2]\!]\eta = \bigcirc & \text{Def. } \mathcal{R}
\end{array}$$

$$\begin{array}{lll}
(\Longleftarrow) & & \\
 & \mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1 \wedge \psi_2]\!]\eta = \bigcirc & \\
\Rightarrow & \mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1]\!]\eta = \bigcirc \text{ and } \mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_2]\!]\eta = \bigcirc & \text{Def. } \mathcal{D},\ \mathsf{reduce}_\wedge \\
\Rightarrow & A \models_\eta \psi_1 \text{ and } A \models_\eta \psi_2 & \text{induction hypothesis} \\
\Rightarrow & A \models_\eta \psi_1 \wedge \psi_2 & \text{Def. } \models
\end{array}$$

- Let $\phi \equiv \psi_1 \vee \psi_2$:

($\Longrightarrow$)

	$A \models_\eta \psi_1 \vee \psi_2$	
$\Rightarrow$	$A \models_\eta \psi_1$ or $A \models_\eta \psi_2$	Def. $\models$
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1]\!]\eta = \bigcirc$ or $\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_2]\!]\eta = \bigcirc$	induction hypothesis
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1 \vee \psi_2]\!]\eta = \bigcirc$	Def. $\mathcal{D}$

($\Longleftarrow$)

	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1 \vee \psi_2]\!]\eta = \bigcirc$	
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_1]\!]\eta = \bigcirc$ or $\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi_2]\!]\eta = \bigcirc$	Def. $\mathcal{D}$, $\mathsf{reduce}_\vee$
$\Rightarrow$	$A \models_\eta \psi_1$ or $A \models_\eta \psi_2$	induction hypothesis
$\Rightarrow$	$A \models_\eta \psi_1 \vee \psi_2$	Def. $\models$

- Let $\phi \equiv \forall\, x \in e \bullet \psi$:

($\Longrightarrow$)

	$A \models_\eta \forall\, x \in e \bullet \psi$	
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. $\models$
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi]\!](\eta \cup \{x \mapsto v\}) = \bigcirc$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\forall\, x \in e \bullet \psi]\!]\eta = \bigcirc$	Def. $\mathcal{D}$, $F = \emptyset$

($\Longleftarrow$)

	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\forall\, x \in e \bullet \psi]\!]\eta = \bigcirc$	
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi]\!](\eta \cup \{x \mapsto v\}) = \bigcirc$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. $\mathcal{D}$, $F = \emptyset$
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for all $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$A \models_\eta \forall\, x \in e \bullet \psi$	Def. $\models$

- Let $\phi \equiv \exists\, x \in e \bullet \psi$:

($\Longrightarrow$)

	$A \models_\eta \exists\, x \in e \bullet \psi$	
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. $\models$
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi]\!](\eta \cup \{x \mapsto v\}) = \bigcirc$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\exists\, x \in e \bullet \psi]\!]\eta = \bigcirc$	Def. $\mathcal{D}$, $\mathcal{V}_A[\![e]\!]\eta \neq []$, $\lvert F\rvert \neq \lvert\mathcal{V}_A[\![e]\!]\eta\rvert$

($\Longleftarrow$)

	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\exists\, x \in e \bullet \psi]\!]\eta = \bigcirc$	
$\Rightarrow$	$\mathcal{D}_A^{(_,\mathsf{False})}[\![\psi]\!](\eta \cup \{x \mapsto v\}) = \bigcirc$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	Def. $\mathcal{D}$, $\mathcal{V}_A[\![e]\!]\eta \neq []$, $\lvert F\rvert \neq \lvert\mathcal{V}_A[\![e]\!]\eta\rvert$
$\Rightarrow$	$A \models_{\eta \cup \{x \mapsto v\}} \psi$ for any $v \in \mathcal{V}_A[\![e]\!]\eta$	induction hypothesis
$\Rightarrow$	$A \models_\eta \exists\, x \in e \bullet \psi$	Def. $\models$

Due to the above induction, the theorem holds for all consistency rules.

□

List of Figures

1.1 Relationships between chapters 7

2.1 Chapter 2 in context 9
2.2 Example repository for this thesis 10
2.3 Formal example rules 11
2.4 Example consistency reports at state 2 12
2.5 Example consistency reports at state 3 13
2.6 Example consistency reports at state 4 13
2.7 Example consistency reports at state 5 14
2.8 S-DAG for rule ϕ_1 at state 4 15
2.9 S-DAG for rule ϕ_2 at state 4 15
2.10 Repairs generated for rules ϕ_1 and ϕ_2 at state 4 16
2.11 Formal example rules with hints 17

3.1 Chapter 3 in context 21
3.2 Overview of a consistency-aware DMS 22

4.1 Chapter 4 in context 24
4.2 Formal example rules 25
4.3 Example types and symbols 27
4.4 Abstract syntax of formulae $\mathcal{F}$ and terms $\mathcal{T}$ 28
4.5 Base signature 31
4.6 Abstract syntax of types 32
4.7 Well-typedness rules for terms and formulae 36
4.8 Subtype relation 38
4.9 Combined type inference and type checking algorithm 40
4.10 Solving subtype constraints 41

5.1 Chapter 5 in context 45
5.2 Example consistency report for rule ϕ_1 at state 4 47
5.3 Classic truth value semantics 49
5.4 A basic report generation algorithm 51
5.5 Example quantifier spheres 54
5.6 Temporally evolving algebra 58
5.7 Algebra transition $\delta_{\mathbb{A}}$ 60
5.8 Repository invariants 61
5.9 Variable assignment $\mathbb{E}$ 61
5.10 Auxiliary functions for report generation 63
5.11 Evaluating terms 64

6.1 Chapter 6 in context 66
6.2 Formal example rules 67
6.3 Example consistency reports at state 2 68

6.4 Example consistency reports at state 3 68
6.5 Pushing quantifiers into formulae 74
6.6 Incremental evaluation of rule ϕ_1' at state 2 and state 3 77
6.7 Incremental evaluation of rule ϕ_2' at state 2. 79
6.8 Incremental variable assignment $\mathbb{E}_{\text{inc}}$ 80
6.9 An incremental report generation algorithm 81
6.10 Auxiliary functions for incremental rule evaluation 83
6.11 Incremental quantifier sphere evaluation 86

7.1 Chapter 7 in context 90
7.2 Overview of a consistency maintaining DMS 95

8.1 Chapter 8 in context 97
8.2 Example rule ϕ_1 (miniscoped) with hints 98
8.3 S-DAG for rule ϕ_1 at state 4 99
8.4 Augmented S-DAG for rule ϕ_1 at state 4 100
8.5 Augmented S-DAG resulting from changing the kind for the manual man1.xml towards technical M. 100
8.6 Abstract syntax of annotated formulae $\mathcal{F}^{\mathcal{H}}$ and hints $\mathcal{H}$ 102
8.7 Typing hints 104
8.8 Complete S-DAG for rule ϕ_1 105
8.9 Reducing the existential node for m at state 4 107
8.10 Reduced S-DAG for rule ϕ_1 at state 4 108
8.11 Abstract syntax of S-DAGs $\mathbb{G}$, predicate suggestions $\mathbb{S}$, and repair actions $\mathbb{C}$ 109
8.12 A basic S-DAG generation algorithm 110
8.13 Variable assignments $\mathbb{E}_{\text{incDAG}}$ for incremental S-DAG generation 113
8.14 An incremental S-DAG generation algorithm 114
8.15 Incremental S-DAG generation for rule ϕ_1 at state 4 116
8.16 Example Hasse diagram 117
8.17 Resulting S-DAG for the existential quantifier for m in rule ϕ_1 at state 4 117
8.18 Evaluating hint collections 118
8.19 Partial orders for S-DAGs: $\preceq$, $\preceq_c$ 119
8.20 Calculating costs for S-DAGs 120
8.21 Auxiliary functions for S-DAG generation 121
8.22 Augmented S-DAG for rule ϕ_1 at state 4 123
8.23 Augmented S-DAG resulting from changing the kind of the manual man1.xml to technical M. 124
8.24 An S-DAG augmentation algorithm 126
8.25 Auxiliary functions for S-DAG augmentation 127

9.1 Chapter 9 in context 129
9.2 Repairing inconsistencies (overview) 130
9.3 Augmented S-DAGs for rule ϕ_1 and ϕ_2, respectively, at state 4 131
9.4 Repairs generated for the rules ϕ_1 and ϕ_2 132

9.5 Abstract syntax of repairs $\mathbb{R}$ 134
9.6 Deriving repair collections from S-DAGs 137
9.7 Auxiliary functions for deriving repair collections 139
9.8 HC-DAG for our example rules ϕ_1 and ϕ_2 145

10.1 Chapter 10 in context 151
10.2 Relationships between analysis modules, document templates, result types, specifications, documents, and results 152
10.3 Overview of analysis modules by sd&m 153

11.1 Chapter 11 in context 167

12.1 Chapter 12 in context 179
12.2 Business process BP_[Plan classes] 182
12.3 Business process BP_[Assign teacher to classes] 183
12.4 Business process BP_[Register students] 184
12.5 Overview of use cases 185
12.6 Interactivity diagram for dialog Dia_[New course and classes] . 189
12.7 Interactivity diagram for dialog Dia_[Assign student to course] 190
12.8 Data model 191
12.9 Augmented S-DAG for Rule 3 at state 1 198
12.10 Augmented S-DAG for Rule 4 at state 23 199
12.11 Augmented S-DAG for Rule 19 at state 6 200
12.12 Augmented S-DAG for Rule 21 at state 17 200
12.13 Augmented S-DAG for Rule 30 at state 17 201
12.14 Augmented S-DAG for Rule 41 at state 23 202
12.15 Augmented S-DAG for Rule 48 at state 17 202
12.16 Augmented S-DAG for Rule 64 at state 17 203
12.17 Augmented S-DAG for Rule 70 at state 6 203
12.18 Augmented S-DAG for Rule 73 at state 17 204
12.19 Augmented S-DAG for Rule 74 at state 17 204
12.20 Augmented S-DAG for Rule 80 at state 23 206
12.21 Augmented S-DAG for Rule 82 at state 6 207
12.22 Augmented S-DAG for Rule 84 at state 6 207
12.23 Top-ranked repair set at state 6 208
12.24 Top-ranked repair set at state 17 210
12.25 Top-ranked repair set at state 23 212

B.1 Performing a consistency check in practice 235

List of Tables

5.1 Example repository up to state 4 46
5.2 Brute force consistency checking performance 65

6.1 Example repository up to state 3 67
6.2 Example symbol metadata 69
6.3 Performance improvements by static analysis 75
6.4 Performance improvements of incremental evaluation over static analysis 84

8.1 Example repository up to state 4 99

9.1 Example ratings calculated by rating_{Σ} and $\mathsf{rating}_{\max}$ 146

10.1 Summary of consistency requirements for analysis modules . . 166

12.1 Contents of the ski school specification for different reader species 180
12.2 Reader species listed in the reading instructions 180
12.3 Who should read which part of a specification? 181
12.4 Naming conventions in the cross-cutting concerns 181
12.5 Data types 192
12.6 Responsibilities of the developers 193
12.7 Inconsistency summary 198
12.8 Performance summary 213

13.1 Cost of formalization 216

A.1 Notation: syntax and semantics 232
A.2 Notation: central algorithms 233

List of Definitions and Theorems

4.1 Signature 31
5.1 Generated consistency reports are sound 52
5.2 Reasons for inconsistencies are sound 52
5.3 Consistency reports indicate real inconsistencies 53
5.4 Σ-algebra 56
5.5 Value 60
5.6 Consistency Report 61
6.1 Weak compressed miniscope formula 72
8.1 S-DAGs indicate real inconsistencies 111

Bibliography

[ABC99] Y. André, F. Bossut, and A. Caron. On first-order constraint checking in object-oriented databases. Technical Report 1999-03, Laboratoire d'Informatique Fondamentale de Lille, 1999.

[ABH03] U. A. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Proc. of the 30th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 14–25, New Orleans, LA, 2003. ACM Press.

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[AFL02] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *Proc. of the 21st ACM Symp. on Principles of Database Systems*, pages 259–270, Madison, WI, 2002. ACM Press.

[AHV95] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal connectives versus explicit timestamps in temporal query languages. In *Proc. of the VLDB Int. Workshop on Temporal Databases*, pages 43–57, Zurich, Switzerland, 1995. Springer-Verlag.

[AHV96] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal versus first-order logic to query temporal databases. In *ACM Symp. on Principles of Database Systems*, pages 49–57, Montreal, Canada, 1996. ACM Press.

[AIS93] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 207–216, Washington, D.C., 1993. ACM Press.

[AL03a] T. E. Ahlswede and R. Y. Lee. Cognitive issues in software requirements analysis. In *Proc. of the 4th Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 113–119, Lübeck, Germany, 2003. ACIS.

[AL03b] M. Arenas and L. Libkin. An information-theoretic approach to normal forms for relational and XML data. In *Proc. of the 22nd ACM Symp. on Principles of Database Systems*, pages 15–26, San Diego, CA, 2003. ACM Press.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 493–574. Elsevier Science Publishers, 1990.

[Bab84] H. Babbage. *Babbage's Calculating Machines*. Charles Babbage Institute Reprint Series. MIT Press, 1984.

[Bal91] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th Int. Conf. on Software Engineering*, pages 158–165, Austin, TX, 1991. IEEE Computer Society Press.

[Bal98] H. Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, volume 2. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[Bas99] S. Basu. New results on quantifier elimination over real closed fields and applications to constraint databases. *J. ACM*, 46(4):537–555, 1999.

[BBG+02] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *Proc. Workshop on Programming Languages for XML*, Pittsburgh, PA, 2002.

[BCP99] V. Benzaken, S. Cerrito, and S. Praud. Static verification of dynamical integrity constraints: a semantics based approach. *Networking and Information System Journal*, 2(5/6):549–569, 1999.

[BD95] V. Benzaken and A. Doucet. Thémis: A database programming language handling integrity constraints. *VLDB Journal*, 4(3):493–518, 1995.

[Bec00] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, Reading, MA, 2000.

[Ber04] B. Berenbach. The evaluation of large, complex UML analysis and design models. In *Proc. of the 26th Int. Conf. on Software Engineering*, pages 232–241, Edinburgh, Scotland, 2004. IEEE Computer Society Press.

[BFW00] P. Buneman, W. Fan, and S. Weinstein. Path consistency rules in semistructured databases. *Journal of Computer and System Sciences*, 61(2):146–193, 2000.

[BHQW02] K. Böhm, G. Heyer, U. Quasthoff, and C. Wolff. Topic Map generation using text mining. *J. UCS*, 8(6):623–633, 2002.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, 2nd edition, 1998.

[BK00] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[BL00] R. A. Bergamaschi and W. R. Lee. Designing systems-on-chip using cores. In *Proc. of the 37th Conf. on Design automation*, pages 420–425. ACM Press, 2000.

[BM95] L. Baekgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM Trans. on Database Systems*, 20(2):111–148, 1995.

[Boo94] G. Booch. *Objektorientierte Analyse und Design.* Addison Wesley, Reading, MA, 1994.

[BP00] L. Bertossi and J. Pinto. Specifying active rules for database maintenance. In *Trans. and Database Dynamics, 8th Int. Workshop on Foundations of Models and Languages for Data and Objects*, volume 1773 of *Lecture Notes in Computer Science*, pages 112–129, Schloß Dagstuhl, Germany, 2000. Springer-Verlag.

[BQBW03] C. Biemann, U. Quasthoff, K. Böhm, and C. Wolff. Automatic discovery and aggregation of compound names for the use in knowledge representations. *J. UCS*, 9(6):530–541, 2003.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, Reading, MA, 1999.

[BRSS03] U. Borghoff, P. Rödig, J. Scheffczyk, and L. Schmitz. *Langzeitarchivierung: Methoden zur Erhaltung digitaler Dokumente.* dpunkt.Verlag, Heidelberg, Germany, 2003.

[BS98] V. Benzaken and X. Schaefer. Static management of integrity in object-oriented databases: Design and implementation. In *Advances in Database Technology - EDBT'98, 6th Int. Conf. on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 311–325, Valencia, Spain, 1998. Springer-Verlag.

[BT99] F. Bry and S. Torge. Solving database satisfiability problems. In *11. Workshop Grundlagen von Datenbanken*, pages 122–126, Luisenthal, Germany, 1999. Friedrich-Schiller-Universität Jena.

[Buc01] K. Buchenrieder, editor. *Hardware / Software Codesign.* ITpress, 2001.

[C+02] P. Cederqvist et al. *Version Management with CVS*, 2002. see www.cvshome.org/docs/manual/.

[C+03] M. Chakravarty et al. *The Haskell Foreign Function Interface 1.0 (Addendum to the Haskell 98 Report)*, 2003. see www.cse.unsw.edu.au/~chak/haskell/ffi/.

[CAM02] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proc. of the 18th Int. Conf. on Data Engineering*, pages 41–52, San Jose, CA, 2002. IEEE Computer Society Press.

[Chi03] Boris Chidlovskii. A structural adviser for the XML document authoring. In *Proc. of the 2003 ACM Symp. on Document Engineering*, pages 203–211, Grenoble, France, 2003. ACM Press.

[CK91] J. M. Crawford and B. Kuipers. ALL: Formalizing access-limited reasoning. In J. F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 299–330. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1991.

[Coc01] A. Cockburn. *Agile Software Development.* Addison Wesley, Reading, MA, 2001.

[CRGW96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 493–504, Montreal, Canada, 1996. ACM Press.

[CSFP04] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion.* O'Reilly and Associates, 2004.

[Dav92] A. J. T. Davie. *An Introduction to Functional Programming Systems using Haskell.* Cambridge University Press, 1992.

[dC86] D. de Champeaux. Subproblem finder and instance checker, two cooperating modules for theorem provers. *J. ACM*, 33(4):633–657, 1986.

[Dec97] H. Decker. One abductive logic programming procedure for two kinds of updates. In *Proc. of Workshop "DYNAMICS'97" at Int. Logic Programming Symposium*, Long Island, NY, 1997. Springer-Verlag.

[Den91] E. Denert. *Software-Engineering.* Springer-Verlag, Berlin, Germany, 1991.

[Den93] E. Denert. Dokumentenorientierte Software-Entwicklung. *Informatik Spektrum*, 16(3):159–164, 1993.

[DENT02] D. Dui, W. Emmerich, C. Nentwich, and B. Thal. Consistency checking of financial derivatives transactions. In *Proc. of NetObject-Days 2002*, pages 172–189, Erfurt, Germany, 2002. Transit GmbH.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[DS95] G. Dong and J. Su. Space-bounded FOIES (extended abstract). In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of database systems*, pages 139–150. ACM Press, 1995.

[DST94] G. Dong, J. Su, and R. Topor. First-order incremental evaluation of Datalog queries. In *Proc. of the 4th Int. Workshop on Database Programming Languages — Object Models and Languages*, pages 295–308, New York, NY, 1994. Springer-Verlag.

[EC01] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *23rd Int. Conf. on Software Engineering*, pages 411–420, Toronto, Canada, 2001. IEEE Computer Society Press.

[ECM99] ECMA Standardizing Information and Communication Systems. ECMAScript language specification. Standard ECMA-262, 1999. see www.ecma.ch.

[EFKN94] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating distributed ViewPoints: the anatomy of a consistency check. Technical Report 333, School of Cognitive and Computing Sciences, University of Sussex, UK, 1994.

[EJS95] W. Emmerich, J.-H. Jahnke, and W. Schäfer. Object oriented specification and incremental evaluation of static semantic constraints. Technical Report 24, Universität Dortmund, Germany, 1995. ESPRIT-III Project GOODSTEP.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science Publishers, 1990.

[Far01] W. M. Farmer. STMM: A set theory for mechanized mathematics. *Journal of Automated Reasoning*, 26:269–289, 2001.

[FFLS99] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Verifying integrity constraints on web sites. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence*, pages 614–619, Stockholm, Sweden, 1999. Morgan Kaufmann Publishers Inc.

[FGLM02] A. Fentechi, S. Gnesi, G. Lami, and A. Maccari. Application of linguistic techniques for use case analysis. In *10th Anniversary IEEE Joint Int. Conf. on Requirements Engineering*, pages 157–164, Essen, Germany, 2002. IEEE Computer Society Press.

[Fin00] Anthony Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Proc. of 11th Int. Conf. on Database and Expert Systems Applications*, volume 1873 of *Lecture Notes in Computer Science*, pages 1–5, London, UK, 2000. Springer-Verlag.

[FM02a] S. Flake and W. Müller. An OCL extension for real-time constraints. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 150–171. Springer-Verlag, 2002.

[FM02b] S. Flake and W. Müller. Specification of real-time properties for UML models. In *Proc. 35th Annual Hawaii Int. Conf. on System Sciences*, volume 9, pages 277–287, Big Island, HI, 2002. IEEE Computer Society Press.

[GHC04] *GHC — The Glasgow Haskell Compiler*, 2004. see www.haskell.org/ghc.

[GHM93] J. C. Grundy, J. G. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Software Engineering*, 24(11):960–981, 1993.

[GHM98] J. C. Grundy, J. G. Hosking, and W. B. Mugridge. Coordinating distributed software development projects with integrated process modelling and enactment environments. In *Proc. of 7th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 39–44, Stanford, CA, 1998. IEEE Computer Society Press.

[GL95] M. Gertz and U. W. Lipeck. Temporal integrity constraints in temporal databases. In *Proc. of the VLDB Int. Workshop on Temporal Databases*, pages 77–92, Zurich, Switzerland, 1995. Springer-Verlag.

[GL97] M. Gertz and U. W. Lipeck. An extensible framework for repairing constraint violations. In *In Proc. 1st Working Conf. on Integrity and Internal Control in Information Systems*, pages 89–111. Chapman & Hall Ltd., 1997.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of data*, pages 157–166, Washington, D.C., 1993. ACM Press.

[GOO94] The GOODSTEP Team. The GOODSTEP project: General object-oriented database for software engineering processes. In *Proc. of the 1st Asian Pacific Software Engineering Conf.*, pages 410–419, Tokio, Japan, 1994. IEEE Computer Society Press.

[GP97] M. Goedicke and C. Piwetz. On modelling inconsistencies in software development processes with graph based notations. In *Proc. of the ICSE'97 Workshop on Living with Inconsistencies*, Boston, MA, 1997. ACM Press.

[GSUW94] A. Gupta, Y. Sagiv, J. Ullman, and J. Widom. Constraint checking with partial information. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 45–55, Minneapolis, MS, 1994. ACM Press.

[GSW89] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[GTW03] J. Z. Gao, H.-S. J. Tsao, and Y. Wu. *Testing and Quality Assurance for Component-Based Software.* Artech House Publishers, 2003.

[Gur00] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Trans. on Computational Logic*, 1(1):77–111, 2000.

[HC01] G. T. Heineman and W. T. Councill. *Component Based Software Engineering: Putting the Pieces Together.* Addison Wesley Professional, 2001.

[HD96] A. Henrich and D. Däberitz. Using a query language to state consistency constraints for repositories. In *Proc. of the 7th Int. Conf. on Database and Expert Systems Applications*, volume 1134 of *Lecture Notes in Computer Science*, pages 59–68, Zurich, Switzerland, 1996. Springer-Verlag.

[HD04] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *Proc. of the 26th Int. Conf. on Software Engineering*, pages 449–458, Edinburgh, Scotland, 2004. IEEE Computer Society Press.

[Hen95] A. Henrich. P-OQL: an OQL-oriented query language for PCTE. In *Proc. of the 7th Conf. on Software Engineering Environments*, pages 48–60, Noordwijkerhout, Netherlands, 1995. IEEE Computer Society Press.

[Hen96] F. Henglein. Syntactic properties of polymorphic subtyping. Technical Report D-293, University of Copenhagen, 1996.

[HK99] M. Hitz and G. Kappel. *UML@Work: Von der Analyse zur Realisierung.* dpunkt.Verlag, Heidelberg, Germany, 1999.

[HPT98] H. Hosoya, B. C. Pierce, and D. N. Turner. Datatypes and subtyping. manuscript, 1998. see www.cis.upenn.edu/~bcpierce/papers/.

[HR01] A. Henrich and G. Robbert. POQL^{MM}: A query language for structured multimedia documents. In *Proc. 1st Int. Workshop on Multimedia Data and Document Engineering*, pages 17–26, Lyon, France, 2001. Sun SITE Central Europe (CEUR).

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society — Student Texts.* Cambridge University Press, 1986.

[HWZ00] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106(1-3):85–134, 2000.

[ISO99a] International Organization for Standardization ISO. *ISO/IEC 9075-1:1999: Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework).* International Organization for Standardization, Geneva, Switzerland, 1999.

[ISO99b] International Organization for Standardization ISO. *ISO/IEC 9075-2:1999: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation).* International Organization for Standardization, Geneva, Switzerland, 1999.

[ISO03] International Organization for Standardization ISO. *ISO/IEC 13250:2003: Information technology — SGML applications — Topic maps.* International Organization for Standardization, Geneva, Switzerland, 2003.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[JCJO92] I. Jacobson, M. Christerson, P. Johnson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[Jon04] M. P. Jones. *The Hugs 98 User's Guide*, 2004. see www.haskell.org/hugs/.

[JPJ99] M. P. Jones and S. Peyton-Jones. Lightweight extensible records for Haskell. In *Proc. of the 1999 Haskell Workshop*, pages 55–66, Paris, France, 1999. Universiteit Utrecht.

[KG02] A. Kwong and M. Gertz. Structural constraints for XML. Technical Report CSE-2002-24, University of California at Davis, 2002.

[KSR02] B. Kane, H. Su, and E. A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Proc. of the 4th Int. Workshop on Web information and data management*, pages 1–8, McLean, VA, 2002. ACM Press.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture — Practice and Promis.* Addison Wesley Pub Co, 1st edition, 2003.

[Liu99] Y. A. Liu. Efficient computation via incremental computation. In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, volume 1574 of *Lecture Notes in Computer Science*, pages 194–203, Beijing, China, 1999. Springer-Verlag.

[Liu00] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher Order Symbol. Comput.*, 13(4):289–313, 2000.

[LM92] P. Lincoln and J. C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Conf. Record of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 293–304, Albuquerque, NM, 1992. ACM Press.

[LY02] L. Li and J. Yunfei. Computing minimal hitting sets with genetic algorithm. *Algorithmica*, 32(1):95–106, 2002.

[Maa98] S. Maabout. Maintaining and restoring database consistency with update rules. In *Proc. of DYNAMICS: Transactions and change in logic databases (Post-Conf. Workshop of JICSLP'98)*, pages 59–74, Manchester, UK, 1998. University of Passau, Germany.

[MACM01] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proc. of the 27th Int. Conf. on Very Large Data Bases*, pages 581–590, Roma, Italy, 2001. Morgan Kaufmann Publishers Inc.

[Maz03] S. Mazanek. Higher-kinded types in the context of subtyping. Diploma thesis, Universität der Bundeswehr München, 2003. UniBwM-ID 29/2003.

[Men87] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks /Cole Advanced Books & Software, 3rd edition, 1987.

[MH89] W. W. McCune and L. J. Henschen. Maintaining state constraints in relational databases: a proof theoretic basis. *J. ACM*, 36(1):46–68, 1989.

[Min83] Minister of Defence. *ZDv 90/1 — Dienstvorschriften der Bundeswehr*. Department of Defence of the Federal Republic of Germany, 1983. English: Service Manual Rules for the Federal Armed Forces Germany.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 365–458. Elsevier Science Publishers, 1990.

[MPG01] A. Macedo, M. Pimentel, and J. Guerrero. Latent semantic linking over homogeneous repositories. In *Proc. of the 2001 ACM Symp. on Document engineering*, pages 144–151, Atlanta, GA, 2001. ACM Press.

[MS96] C. Martín and J. Sistac. An integrity constraint checking method for temporal deductive databases. In *3rd Workshop on Temporal Representation and Reasoning*, pages 136–144, Key west, FL, 1996. IEEE Computer Society Press.

[MS01] G. Moro and C. Sartori. Incremental maintenance of multi-source views. In *Proc. of the 12th Australasian conference on Database technologies*, pages 13–20, Queensland, Australia, 2001. IEEE Computer Society Press.

[MT99] E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *Proc. of Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems,*

and the World Wide Web and Conceptual Modeling, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73, Paris, France, 1999. Springer-Verlag.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proc. of the 2nd ACM SIGPLAN Int. Conf. on Functional Programming*, pages 136–149, Amsterdam, Netherlands, 1997. ACM Press.

[Nak01] H. Nakamura. Incremental computation of complex object queries. In *Proc. of the OOPSLA '01 Conf. on Object Oriented Programming Systems Languages and Applications*, pages 156–165, Tampa Bay, FL, 2001. ACM Press.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Inter. Tech.*, 2(2):151–185, 2002.

[NEF01] C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *Proc. of the 16th Int. Conf. on Automated Software Engineering*, San Diego, CA, 2001. IEEE Computer Society Press.

[NEF03] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. of the 25th Int. Conf. on Software Engineering*, pages 455–464, Portland, OR, 2003. IEEE Computer Society Press.

[NER00] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.

[NG92] K. Narayanaswamy and N. M. Goldman. "Lazy" consistency: A basis for cooperative software development. In *Proc. of the Conf. on Computer Supported Cooperative Work*, pages 257–264, Toronto, Canada, 1992. ACM Press.

[Nor98] J. Nordlander. Pragmatic subtyping in polymorphic languages. In *Proc. of the 3rd ACM SIGPLAN Int. Conf. on Functional Programming*, pages 216–227, Baltimore, MD, 1998. ACM Press.

[Nor99] J. Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Chalmers Tekniska, Högskola, 1999.

[Nor02] J. Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2-3):93–127, 2002.

[OMG03] Object Management Group OMG. OMG Unified Modelling Language specification, version 1.5. OMG document formal/03-03-01, 2003.

[Pac97] M. A. Pacheco e Silva. Dynamic integrity constraints definition and enforcement in databases: a classification framework. In *Proc. of the IFIP TC-11 Working Group 11.5 1st Working Conf. on Integrity and Internal Control in Information Systems*, pages 65–87, Zurich, Switzerland, 1997. Chapman Hall.

[Pat02] J. Patrick. *SQL fundamentals.* Prentice Hall PTR, 2nd edition, 2002.

[PJ03] S. L. Peyton-Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[Ple93] D. Plexousakis. Integrity constraint and rule maintenance in temporal deductive knowledge bases. In *Proc. of the 19th Conf. on Very Large Databases*, pages 146–157, Dublin, Ireland, 1993. Morgan Kaufmann Publishers Inc.

[Ple95] D. Plexousakis. Compilation and simplification of temporal integrity constraints. In *Proc. of the 2nd Int. Workshop on Rules in Database Systems*, volume 985 of *Lecture Notes in Computer Science*, pages 260–276, Athens, Greece, 1995. Springer-Verlag.

[Ple96] D. Plexousakis. *On the Efficient Maintenance of Temporal Integrity in Knowledge Bases.* PhD thesis, University of Toronto, 1996.

[PNEF03] C. Pérez Arroyo, C. Nentwich, W. Emmerich, and A. Finkelstein. Scaling consistency checking, 2003. see www.cs.ucl.ac.uk/staff/nentwich/publications/scalingchecking.html.

[Pot98] François Pottier. *Type inference in the presence of subtyping: from theory to practice.* PhD thesis, Inria, France, 1998.

[Pot01] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, 2001.

[Pow03] S. Powers. *Practical RDF.* O'Reilly & Associates, 2003.

[Rat94] B. Ratcliff. *Introducing Specification Using Z: A Practical Case Study Approach.* McGraw Hill Book Co Ltd, 1994.

[Rep84] T. W. Reps. *Generating language-based environments.* PhD thesis, Massachusetts Institute of Technology, 1984.

[Rön03] S. Rönnau. Open-source Dokumentenmanagementsysteme. Studienarbeit, Universität der Bundeswehr München, 2003. UniBwM-IS 31/2003.

[Rön04] S. Rönnau. Versionsverwaltung von XML-Dokumenten am Beispiel von OpenOffice.org. Diploma thesis, Universität der Bundeswehr München, Munich, Germany, forthcoming 2004. English: Revision Control of XML Documents.

[Rou04] D. Roundy. DARCS: David's advanced revision control system, 2004. see www.abridgegame.org/darcs/.

[RS01] M. Roggenbach and L. Schröder. Towards trustworthy specifications I: consistency checks. In *Recent Trends in Algebraic Development Techniques, 15th Int. Workshop*, volume 2267 of *Lecture Notes in Computer Science*, pages 305–327, Genova, Italy, 2001. Springer-Verlag.

[Sal88] G. Salton. Automatic indexing and abstracting. In *Document retrieval systems*, pages 42–80. Taylor Graham Publishing, 1988.

[SBRS03a] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz. Consistent document engineering. In *Proc. of the 2003 ACM Symp. on Document Engineering*, pages 140–149, Grenoble, France, 2003. ACM Press.

[SBRS03b] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz. Efficient (in-) consistency management for heterogeneous repositories. In *Proc. of the 4th Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 370–377, Lübeck, Germany, 2003. ACIS.

[SBRS04a] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz. Managing inconsistent repositories via prioritized repair actions. In *Proc. of the 2004 ACM Symp. on Document Engineering*, pages 137–146, Milwaukee, WI, 2004. ACM Press.

[SBRS04b] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz. S-DAGs: Towards efficient document repair generation. In *Proc. of the 2nd Int. Conf. on Computing, Communications and Control Technologies*, volume 2, pages 308–313, Austin, TX, 2004.

[SBRS04c] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz. Towards efficient consistency management for informal applications. *Int. Journal of Computer & Information Science*, 5(2):109–121, 2004.

[SdS99] R. R. Seljée and H. C. M. de Swart. Three types of redundancy in integrity checking; an optimal solution. *Data & Knowledge Engineering*, 30(2):135–151, 1999.

[Sjö02] M. Sjögren. Dynamic loading and web servers in Haskell. see www.mdstud.chalmers.se/~md9ms/hws-wp/, 2002.

[SK02] J. Siedersleben and W. Krug. Bausteine der Spezifikation. In J. Siedersleben, editor, *Softwaretechnik*. Hanser, 2nd edition, 2002.

[Smi91] G. S. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, 1991.

[Smi94] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989. see spivey.oriel.ox.ac.uk/~mike/zrm/.

[SPJ01] M. Shields and S. L. Peyton-Jones. Object-oriented style overloading for Haskell. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[SSBS04] J. Scheffczyk, C. Stutz, U. M. Borghoff, and J. Siedersleben. Formale Konsistenzsicherung in informellen Software-Spezifikationen. *Informatik Forschung und Entwicklung*, 19(1):17–29, 2004.

[SSKK02] C. Stutz, J. Siedersleben, D. Kretschmer, and W. Krug. Analysis beyond UML. In *10th Anniversary IEEE Joint Int. Conf. on Requirements Engineering*, pages 215–218, Essen, Germany, 2002. IEEE Computer Society Press.

[Ste04] D. B. Stewart. Dynamically loaded Haskell plugins. see www.cse.unsw.edu.au/~dons/hs-plugins/, 2004.

[SWJF96] K. Shafer, S. Weibel, E. Jul, and J. Fausey. *Introduction to Persistent Uniform Resource Locators*, 1996. see purl.oclc.org/OCLC/PURL/INET96.

[SZ01] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Co., 2001.

[Tho96] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1996.

[TO95] E. Teniente and A. Olivé. Updating knowledge bases while maintaining their consistency. *VLDB Journal*, 4(2):193–241, 1995.

[Tor98] S. Torge. *Überprüfung der Erfüllbarkeit im Endlichen: Ein Verfahren und seine Anwendung*. PhD thesis, Ludwig-Maximilians-Universität München, 1998.

[Toz01] A. Tozawa. Towards static type checking for XSLT. In *Proc. of the 2001 ACM Symp. on Document Engineering*, pages 18–27, Atlanta, GA, 2001. ACM Press.

[VC99] A. I. Vermesan and F. Coenen, editors. *Validation and Verification of Knowledge Based Systems - Theory, Tools and Practice, Papers from EUROVAV '99, 5th European Symp. on Validation and Verification of Knowledge Based Systems*, Oslo, Norway, 1999. Kluwer Academic Publishers.

[W3C99a] World Wide Web Consortium W3C. RDF primer. W3C Recommendation, 1999. see www.w3.org/TR/REC-rdf-syntax/.

[W3C99b] World Wide Web Consortium W3C. XML path language (XPath) version 1.0. W3C Recommendation, 1999. see www.w3.org/TR/xpath.

[W3C01] World Wide Web Consortium W3C. XML Schema part 0: Primer. W3C Recommendation, 2001. see www.w3.org/TR/xmlschema-0/.

[W3C03] World Wide Web Consortium W3C. XML path language (XPath) 2.0. W3C Working Draft, 2003. see www.w3.org/TR/xpath20/.

[W3C04] World Wide Web Consortium W3C. Extensible Markup Language (XML) 1.1. W3C Recommendation, 2004. see www.w3.org/TR/xml11.

[Wan60] Hao Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2–22, 1960.

[WDC03] Y. Wang, D. J. DeWitt, and J. Cai. X-Diff: An effective change detection algorithm for XML-documents. In *19th Int. Conf. on Data Engineering*, pages 519–530, Bangalore, India, 2003. IEEE Computer Society Press.

[WDSY91] O. Wolfson, H. M. Dewan, S. J. Stolfo, and Y. Yemini. Incremental evaluation of rules and its relationship to parallelism. In *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, pages 78–87, Denver, CO, 1991. ACM Press.

[Wir90] M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675–788. Elsevier Science Publishers, 1990.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley Pub Co, 1998.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison Wesley Pub Co, 2nd edition, 2003.

[WL02] R. K. Wong and N. Lam. Managing and querying multi-version XML data with update logging. In *Proc. of the 2002 ACM Symp. on Document Engineering*, pages 74–81. ACM Press, 2002.

[WM02] R. Widhalm and T. Mück. *Topic Maps: Semantische Suche im Internet*. Springer-Verlag, Berlin, Germany, 2002.

[WR99] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the 4th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 148–159, Paris, France, 1999. ACM Press.

[WS99] R. Wilkinson and A. F. Smeaton. Automatic link generation. *ACM Comput. Surv.*, 31(4es):27, 1999.

[YS88] D. M. Yellin and R. E. Strom. INC: a language for incremental computations. In *Proc. of the SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pages 115–124, Atlanta, GA, 1988. ACM Press.

[YS91] D. M. Yellin and R. E. Strom. INC: a language for incremental computations. *ACM Trans. on Programming Languages and Systems*, 13(2):211–236, 1991.

[ZG02] P. Ziemann and M. Gogolla. An extension of OCL with temporal logic. In *Critical Systems Development with UML — Proc. of the UML'02 Workshop*, pages 53–62, Dresden, Germany, 2002. Technische Universität München.

[ZK03] A. Zisman and A. Kozlenkov. Managing inconsistencies in UML specifications. In *Proc. of the 4th Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 128–138, Lübeck, Germany, 2003. ACIS.