

S-DAGs: Towards Efficient Document Repair Generation

Jan Scheffczyk, Uwe M. Borghoff, Peter Rödiger, Lothar Schmitz
Universität der Bundeswehr München, Munich, Germany

ABSTRACT

Semantic inconsistencies are major obstacles when multiple authors edit interrelated documents. Current document management systems (DMS) often lack adequate facilities for consistency management. In previous work, we have proposed liberal use of formal consistency rules: We *permit inconsistencies* and exactly identify their sources.

In this paper, we extend our approach towards suggesting *repairs* that resolve inconsistencies. The critical issues are to suggest some of the best (i.e., least cost) repairs only and to generate repairs efficiently. We employ directed acyclic graphs (DAGs) to carry repairs. These graphs are called *suggestion DAGs* (short: S-DAGs). The main contributions of this paper are: (1) incorporation of domain knowledge to guide repair generation, (2) visualization of inconsistencies and repairs through S-DAGs, and (3) efficient generation of cheap repairs.

1. INTRODUCTION

Typically, a multitude of authors is involved in producing larger bodies of writing that contain many interrelated documents, such as books, technical documentations, or software specifications. For managing document versions and author access rights mainstream DMS rely on a document repository. In order to produce an overall consistent result authors have to spend plenty of time re-reading and revising their own and related documents. Worse, each check-in to the repository may violate consistency again.

This motivates the use of formal consistency rules to capture semantic constraints between documents. For flexible document management it is vital to allow inconsistencies w.r.t. these rules [11]. In [13] we have employed temporal predicate logic with linear time and have extended boolean semantics in order to generate *consistency reports* pointing out inconsistent document parts. This is, however, only a first step towards flexible consistency management.

In this paper we propose strategies to *repair inconsistencies*. Approaches to generate repairs must handle the combinatorial explosion of the possible repairs. In order to reduce the search space for reasonable repairs we allow to annotate consistency rules by hints, which guide repair generation. Instead of enumerating all possible repairs [9], we *describe* repairs by directed acyclic graphs (DAGs). We call these DAGs *suggestion DAGs* (short: S-DAGs). S-DAGs are optimized for efficient repair generation and also provide a convenient way to visualize repairs. In contrast to constraint maintenance in active databases, we do not aim at automatic repair. In document management, automatic repair is not a feasible option because (1) document structures¹ are less formal than database schemata and (2) the order of the content inside a document is 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 for-

¹We do not even require a common document format.

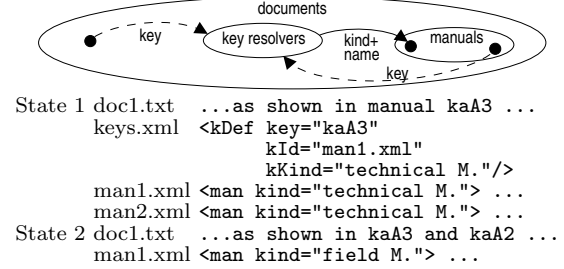


Figure 1: Example repository

mality because we do not know where to add the content. Thus repairs must be applied manually. Consequently, we aim at the generation of a few repairs and present them in S-DAGs. From S-DAGs authors can choose repairs.

Throughout this paper, we use the following example; for a more complex scenario see [14]. We find complex examples that raise closely related problems also in [8, 15].

EXAMPLE 1. Assume that we want to archive manuals over a long period of time. Documents reference manuals through a key (see Fig. 1). 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 the (two-step) references are valid. 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 repository contain a document doc1.txt, a key resolver keys.xml, and two manuals man1.xml and man2.xml. The check-ins at the second state cause two inconsistencies: (1) the reference to kaA3 is inconsistent because the kind of man1.xml is different from the kind of kaA3's key definition; (2) the reference to kaA2 is inconsistent because the key resolver lacks a definition for kaA2. □

Related Work: For our approach we borrow techniques from the field of active databases. There, current research concentrates on the derivation of active rules that fire repairs automatically in order to arrive at a consistent database state after an update [7, 3]. Compared to these approaches, our method is more flexible because it is independent of a specific document model. On the other hand, less formal update descriptions and less formal document semantics prohibit the automatic application of computer-generated repairs. Hence, we cannot apply iterative repair techniques, known from databases [6]. Instead, we present a few repair alternatives to authors only.

The toolkit xlinkit [9] generates repair actions to resolve inconsistencies in distributed documents. The approach enumerates all possible repairs and thus suffers from an exponential computational complexity. In contrast, we reduce the number of generated repairs. Recent approaches in the XML community [2] aim to guarantee consistency by construction. In contrast, we apply consistency checks

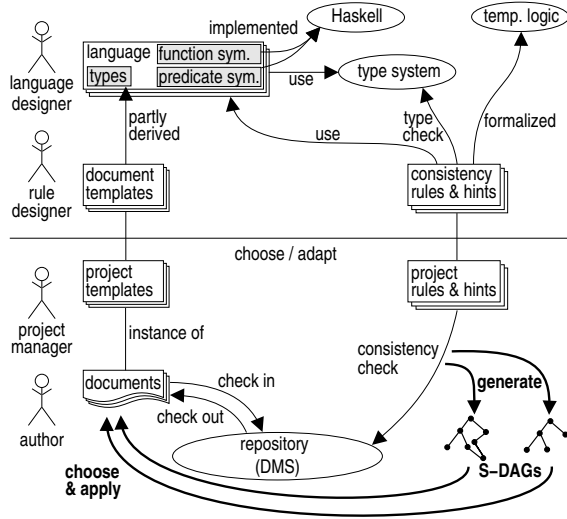


Figure 2: System overview (ovals are fixed components; rectangles are customizable components)

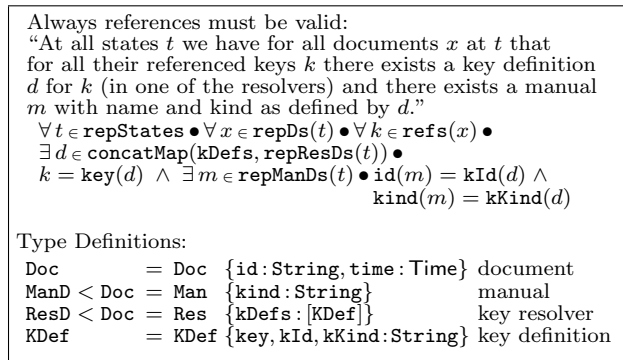


Figure 3: Example rule and type definitions

to concrete documents, which is better suited for managing semantic (i.e., content-dependent) consistency requirements. We integrate incremental consistency checks into a DMS to relieve the obstacle of recurring checks.

Outline: This paper is organized as follows: Sect. 2 summarizes our extensions to a typical DMS and shows how consistency rules are formalized. In Sect. 3 we formalize hints, which guide repair generation. Sect. 4 introduces S-DAGs as a powerful means to describe repairs and presents an algorithm to generate S-DAGs. In Sect. 5 we show how S-DAGs can be augmented in order to support interactive repair. In Sect. 6 we put augmented S-DAGs to practice. Sect. 7 shows directions for future research.

2. CONSISTENCY-AWARE DMS

In this section we outline how we integrate our approach into an existing DMS (for details see [13]).

2.1 Background: The DMS Model

Since formalizing consistency rules affects various problem areas, we divide formalization into different tasks (see Fig. 2). From a repository *authors* typically check out working copies of documents, modify them, and finally check them in again. Among other things, a classic DMS manages concurrent check-ins, author permissions, and version control. A *rule designer* formalizes consistency rules.

We check the repository for consistency at given events, e.g., document check-in. For each rule this generates an S-DAG describing repairs, from which authors can choose. Consistency rules use function symbols and predicate symbols from domain specific languages. Our static type system ensures well-typedness of rules. A *language designer* declares valid symbols and types. Symbol semantics is defined in Haskell [12]. For specific projects, a *project manager* chooses consistency rules. In some cases adaptations will be necessary, e.g., some rules may be weakened.

2.2 The Formal Rule Model

The rule designer formalizes consistency rules in a variant of the two-sorted temporal first-order predicate logic with linear time and equality [5]. Instead of temporal operators like *since* and *until*, we use explicit quantification over timestamps in rules. Timestamps are represented by the type *Time*. Consistency rules mostly correspond to standard first-order formulae where the domain of a quantifier is explicitly given through a term.

Our static type system comprises record and variant types. Cardelli-style subtypes [10] prove practical for document management; we model document subsets from our example via subtypes. We define a document type *Doc* to carry the name and the check-in time of a document; other document types are subtypes of *Doc*.

EXAMPLE 2. The rule designer formalizes our example rule as shown in Fig. 3. The rule first quantifies over all states in the repository, provided by *repStates*. The term *repDs(t)* returns the current documents for a state t . For each document x the variable k comprises the referenced keys, obtained by *refs(x)*. Key definitions in the resolvers are computed using the higher-order function *concatMap*. It applies *kDefs* to each key resolver, computed by *repResDs(t)*. Applied to a key resolver *kDefs* returns a list of key definitions, which are finally concatenated. Thus d ranges over the key definitions from all key resolvers; we require that the referenced key k equals the key defined by d . The variable m ranges over all manuals at the state t , computed via *repMandS(t)*. The name of the manual m must equal the name d points to and the kind of m must equal the kind d points to.

The language designer defines types as shown in Fig. 3. Definitions of standard types (lists `[]` and `Strings`) are omitted. A subtype relation between record types is denoted by $<$. Record labels induce new function symbols, e.g., *kDefs*. For brevity we omit declarations and implementations of function symbols and predicate symbols. $\square$

From the formalized consistency rule and the repository as shown in Fig. 1 our system generates an S-DAG, which describes repairs for the inconsistencies occurred. For example, these repairs suggest to change the kind of the manual *man1.xml* back to *technical M.* and to delete the key reference *kaA2* in *doc1.txt*.

3. HINTS FOR “BETTER” REPAIRS

Repairs are derived from *predicate suggestions*, which in turn are generated from atomic formulae. A predicate suggestion indicates how the truth value of an atomic formula should be changed. For example, the reference to *kaA2* causes violation of the atomic formula $k = \text{key}(d)$ for the variable assignment

$$\left\{ \begin{array}{l} k \mapsto \text{kaA2}, \\ d \mapsto \{\text{key} = \text{kaA3}, \text{kId} = \text{man1.xml}, \text{kKind} = \text{technical M.}\} \end{array} \right\}$$

$\phi, \psi ::= p(e_1, \dots, e_n) \{ \{ \bar{h} \} \}$	atomic formula
$\neg \phi \mid \phi \cdot \psi$	negation; junction, $\cdot \in \{\vee, \wedge\}$
$Q x \in e \bullet \phi$	quantified formula, $Q \in \{\forall, \exists\}$
$e ::= x \mid f$	variable; (function) symbol
$f(e_1, \dots, e_n)$	application
$h ::= x \rightsquigarrow e \mid b$	Hint: set x to e if the predicate result is b
$x.f \rightsquigarrow e \mid b$	set the field f of x to e

Figure 4: Parts of the formal rule syntax

$\forall t \in \text{repStates} \bullet \forall x \in \text{repDs}(t) \bullet \forall k \in \text{refs}(x) \bullet$	
$\exists d \in \text{concatMap}(\text{kDefs}, \text{repResDs}(t)) \bullet$	
$k = \text{key}(d)$	$\{k \rightsquigarrow \text{key}(d) \text{ False}\},$ $\{d.\text{key} \rightsquigarrow k \text{ False}\}$
$\wedge$	
$\left(\begin{array}{l} \exists m \in \text{repManDs}(t) \bullet \\ \text{id}(m) = \text{kId}(d) \quad \{ \{m.\text{id} \rightsquigarrow \text{kId}(d) \text{ False} \} \} \\ \wedge \\ \text{kind}(m) = \text{kKind}(d) \quad \{ \{m.\text{kind} \rightsquigarrow \text{kKind}(d) \text{ False} \} \} \end{array} \right)$	

Figure 5: Example rule with hints

Then we should generate two suggestions:

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

Clearly, both alternatives are plausible and make the predicate $k = \text{key}(d)$ true under the given assignment. In some situations, however, 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.

Suggestions as above cannot be derived automatically because they require domain knowledge about the predicates used. Thus we allow rule designers to formalize *hints* for variables, depending on the truth value of an atomic formula. Our consistency checker evaluates hints and generates the corresponding predicate suggestions. If no hint is given, a predicate suggestion simply proposes to flip the truth value of the predicate.

A hint is formalized through a term and has one of the two forms, shown in Fig. 4. In a hint $x \rightsquigarrow e \mid b$, 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. If x has a record type, we also allow a reduced hint, which applies to a record field f of x only. Inside 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 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 of hint sets. Each hint set is an alternative. Within a hint set, all hints are evaluated simultaneously. Our static type checker ensures well-typedness of hints such that predicate suggestions are compatible with the document structure.

For our example rule we formalize hints as shown in Fig. 5. We “blame” the manuals m for inconsistent manual identifiers and kinds since we did not formalize a hint for the key definition d . Our consistency checker evaluates these hints to predicate suggestions as shown above.

Now that the rule designer has formalized hints, we wish to generate repairs from these hints. One might be tempted to enumerate repairs similar to [9]: The collection of predicate suggestions for an atomic formula forms a basic repair collection. One could join the repair collections from the subformulae of a disjunction or an existential quantifier. For conjunctions and universal quantifiers

$d ::= \boxed{\text{Abandoned}}$	abandoned S-DAG
$\phi \mid b \mid \{ \{ \text{psug} \} \}$	predicate leaf
$\ominus d$	negation node, child DAG d
$\odot \{ \bar{d} \}$	junction node, children $\bar{d}$
$\odot x \{ \{ (v, d) \} \}$	quantifier node, variable x ; edges with bound value v , child d
$\text{psug} ::= x [v_1 \rightsquigarrow v_2]$	Predicate suggestion: change x from v_1 to v_2
$x.f [v_1 \rightsquigarrow v_2]$	change field f of x from v_1 to v_2

Figure 6: Abstract syntax of S-DAGs

one could compute the cartesian product of the repair collections from the subformulae. This causes an exponential increase in the number of repair alternatives, which makes the approach infeasible. Therefore, we employ S-DAGs to avoid the computation of cartesian products.

4. DESCRIBING REPAIRS BY S-DAGS

Describing repairs by S-DAGs has several advantages:

- Generating S-DAGs (which can also be done incrementally) is much faster than enumerating repair collections. S-DAGs can be reduced during their generation in order to abandon redundant repairs.
- S-DAGs visualize repairs in a convenient way to authors. Sharing of nodes avoids redundancies and helps authors to anticipate the effect of a repair.
- From an S-DAG we can derive a repair collection on demand, i.e., *after* the actual consistency check.

4.1 What is an S-DAG?

The structure of an S-DAG resembles that of a consistency rule. Nodes represent logical connectives; edges target S-DAGs for the subformulae. As usual, edges point always downwards. We distinguish seven kinds of nodes:

Universal nodes $\forall$ and existential nodes $\exists$ represent universal and existential quantification, respectively. Outgoing edges carry value bindings to their quantified variable. A value represents a state in the repository, content inside a document, or a document itself. Conjunction nodes $\wedge$ and disjunction nodes $\vee$ stand for conjunctions and disjunctions, respectively. A negation node $\neg$ represents a negation. A predicate leaf contains the atomic formula in question, its truth value, and its predicate suggestion collection. The leaf $\boxed{\text{Abandoned}}$ represents an S-DAG that has been reduced (see Sect. 4.2). Formally, we define S-DAGs as shown in Fig. 6.

From the user perspective, a universal node represents inconsistencies resulting from *failure prone* document content, where each edge blames a value for inconsistencies. 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. 5 we present a heuristics to determine which of these possibilities applies. Below conjunction nodes and universal nodes each S-DAG must be repaired. In contrast, it is sufficient to repair only one S-DAG below disjunction nodes and existential nodes. Naturally, S-DAGs containing the same information are shared.

Fig. 7 shows the complete S-DAG for our example. In the leaves we find predicate suggestions for atomic formulae. The lower right hand leaf tells that the atomic formula $\text{kind}(m) = \text{kKind}(d)$ is violated when m is bound to

$\mathcal{D}_{(TL,I)}^{neg} \llbracket p(e_0, e_1, \dots, e_n) \text{ hss} \rrbracket \eta$	$(p \text{ partially temporal})$
$= \frac{p(e_0, e_1, \dots, e_n) \text{ b sugss}}{\bigcirc}$	if $neg = b$ otherwise
where $\dot{A}$	$= I(\mathcal{V}_{TL} \llbracket e_0 \rrbracket \eta)$
b	$= (\mathcal{V}_{(TL,I)} \llbracket e_1 \rrbracket \eta, \dots, \mathcal{V}_{(TL,I)} \llbracket e_n \rrbracket \eta) \in p^{\dot{A}}$
$sugss$	$= \mathcal{S}_{(TL,I)} \llbracket \text{filterHints}(\text{hss}, b, \eta) \rrbracket \eta$
$\mathcal{D}_{(TL,I)}^{neg} \llbracket p(e_1, \dots, e_n) \text{ hss} \rrbracket \eta$	$(p \text{ fully temporal})$
$= \frac{p(e_1, \dots, e_n) \text{ b sugss}}{\bigcirc}$	if $neg = b$ otherwise
where b	$= (\mathcal{V}_{TL} \llbracket e_1 \rrbracket \eta, \dots, \mathcal{V}_{TL} \llbracket e_n \rrbracket \eta) \in p^{TL}$
$sugss$	$= \mathcal{S}_{(TL,I)} \llbracket \text{filterHints}(\text{hss}, b, \eta) \rrbracket \eta$
$\mathcal{D}_A^{neg} \llbracket \neg \phi \rrbracket \eta$	$= \frac{\ominus d \ d \neq \bigcirc}{\bigcirc}$ where $d = \mathcal{D}_A^{neg} \llbracket \phi \rrbracket \eta$ otherwise
$\mathcal{D}_A^{neg} \llbracket \phi \wedge \psi \rrbracket \eta$	$= \frac{\bigotimes \text{reduce}_{\wedge}(d_{\phi}, d_{\psi})}{\bigcirc}$ if $d_{\phi} \neq \bigcirc \wedge d_{\psi} \neq \bigcirc$ $\bigotimes \{d_{\phi}\}$ if $d_{\phi} = \bigcirc$ $\bigotimes \{d_{\psi}\}$ if $d_{\psi} = \bigcirc$ $\bigcirc$ otherwise
where $d_{\phi} = \mathcal{D}_A^{neg} \llbracket \phi \rrbracket \eta$	$d_{\psi} = \mathcal{D}_A^{neg} \llbracket \psi \rrbracket \eta$
$\mathcal{D}_A^{neg} \llbracket \phi \vee \psi \rrbracket \eta$	$= \frac{\bigotimes \text{reduce}_{\vee}(d_{\phi}, d_{\psi})}{\bigcirc}$ if $d_{\phi} \neq \bigcirc \wedge d_{\psi} \neq \bigcirc$ otherwise
where $d_{\phi} = \mathcal{D}_A^{neg} \llbracket \phi \rrbracket \eta$	$d_{\psi} = \mathcal{D}_A^{neg} \llbracket \psi \rrbracket \eta$
$\mathcal{D}_A^{neg} \llbracket \forall x \in e \bullet \phi \rrbracket \eta$	$= \frac{\bigotimes x \ F}{\bigcirc}$ if $F \neq \emptyset$ otherwise
where $F = \{(v, d) \mid v \in \mathcal{V}_A \llbracket e \rrbracket \eta \text{ and } d \neq \bigcirc\}$	where $d = \mathcal{D}_A^{neg} \llbracket \phi \rrbracket \eta[x \mapsto v]$
$\mathcal{D}_A^{neg} \llbracket \exists x \in e \bullet \phi \rrbracket \eta$	$= \frac{\text{null}(e) \ \text{False} \ \emptyset}{\bigotimes x \ \text{reduce}_{\exists}(F)}$ if $\mathcal{V}_A \llbracket e \rrbracket \eta = \emptyset$ if $ F = \mathcal{V}_A \llbracket e \rrbracket \eta $ $\bigcirc$ otherwise
where $F = \{(v, d) \mid v \in \mathcal{V}_A \llbracket e \rrbracket \eta \text{ and } b \neq \bigcirc\}$	where $d = \mathcal{D}_A^{neg} \llbracket \phi \rrbracket \eta[x \mapsto v]$

Figure 9: Generating S-DAGs

5. AUGMENTING S-DAGS

The above algorithm generates the S-DAG shown in Fig. 8 for our example. This S-DAG lacks, however, sufficient information about how inconsistencies can be repaired, e.g., by adding, changing, or deleting content. In order to support interactive repair we *augment* S-DAGs, which is independent of consistency checking. Therefore, the repository is not locked during S-DAG augmentation.

Our basic idea is to annotate quantifier edges by concrete repairs. A repair proposes to add a value to (Add), or change a value inside (Chg), or delete a value from the domain of this quantifier (Del). We augment the S-DAG from Fig. 8 as shown in Fig. 10; repairs are marked grey. For example, we propose to change the key reference kaA2 into kaA3 or to delete doc1.txt. Repairs for a quantifier edge are obtained from the predicate suggestions below it. For brevity we omit our S-DAG augmentation algorithm but instead describe the rationale behind it.

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 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 inside the domain, i.e., one of the S-DAGs below

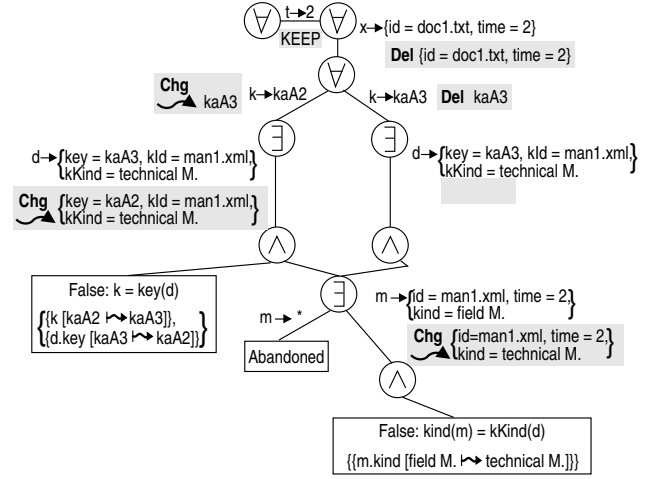


Figure 10: Augmented S-DAG from Fig. 8

the existential node is minimal w.r.t. the S-DAG ordering $\prec$. 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 add a new value to the domain. Usually, this value is partially defined by the predicate suggestions below it.⁵

Inconsistencies resulting from *failure prone document content* can be resolved either by changing the content or by deleting it; adding new content cannot resolve this kind of inconsistency. For universal nodes (representing failure prone content) we know exactly which values cause inconsistencies. We have to decide whether an offending value should be changed or deleted. Again, we employ the criterion of minimal change. Assume we follow an edge binding a variable x to a failure prone value v . If the edge’s target S-DAG contains a predicate suggestion that proposes to change x then we derive a repair to change v . Otherwise, we derive a repair to delete v because it is unclear how this inconsistency can be resolved.

In our example, we have not augmented the universal node for the temporal variable t . We cannot change t because it represents a repository state. Because of this situation, we allow rule designers to specify whether our algorithm should take a quantified variable into account.

An important question is the impact of a repair to the *document structure*. Since hints are type-checked, the derived repairs respect the document structure. But static type-checking cannot detect whether adding or deleting content could break the document structure (this requires dynamic type-checking). Consequently, repairs are type safe by construction only if they propose to change content inside a document. For repairs proposing to add content static type-checking can only assure that this content is type safe provided it is allowed to add content at all. Repairs proposing to delete content are not affected by type-checking because we generate these fall back solutions only when an S-DAG lacks predicate suggestions.

6. INTERACTIVE REPAIR

S-DAG augmentation derives repairs for each quantifier edge. It is, however, unclear how authors should combine

⁵We handle missing information by “value skeletons,” in which some record fields can be undefined. This is similar to handling null values in databases (see [6]).

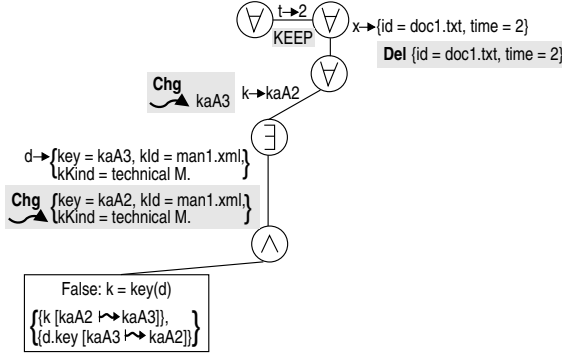


Figure 11: Repairing the S-DAG from Fig. 10

repairs in order to resolve all inconsistencies w.r.t. the corresponding rule. In our interactive approach the author walks through the augmented S-DAG and picks repairs individually. Our system applies a chosen repair to the S-DAG and re-augments it. If this results in an empty S-DAG all inconsistencies are resolved. Otherwise, the remaining inconsistencies can be resolved by picking another repair. Obviously, it makes only sense to choose repairs from the *current* S-DAG, which represents inconsistencies in the current repository state. Due to the nature of DMS, we cannot change older document versions. Previous inconsistencies document the development of the repository, which has been proven a quite useful feature.

For our example S-DAG in Fig. 10 we choose to repair the existential node for m because it has two incoming paths from the S-DAG root and is, therefore, responsible for two inconsistencies. Thus, we change the kind of the manual `man1.xml` from field `M.` into `technical M.` This causes the lower 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 S-DAG on the right hand side of the universal node for k collapses. We arrive at the S-DAG shown in Fig. 11, which we can repair, e.g., by changing the key reference `kaA2` into `kaA3` in `doc1.txt`.

Choosing repairs from augmented S-DAGs is an interactive trial and error process, which determines a reasonable repair set. A repair may, however, cause new inconsistencies. We can determine this only after a new consistency check of the repository. For example, changing the key inside the key definition of `kaA3` (as proposed by the repair for d in Fig. 11) imposes new inconsistencies. Solving this issue as well as deriving repairs for multiple rules is beyond the scope of this paper and is dealt with elsewhere.

7. CONCLUSION AND OUTLOOK

In this paper, we present an approach to interactively repair inconsistencies in heterogeneous repositories. Due to lack of formality in the semantics of documents, we suggest repairs only. We describe repairs through S-DAGs in order to avoid exponential computational complexity of simple repair enumeration. Preliminary experiments [14] confirm that our system generates useful repairs.

Currently, we extend our approach by deriving repairs for multiple rules from their S-DAGs. In the future we plan to use data mining approaches [1] to improve the usefulness of repairs. We are confident that the contributions of this paper provide useful guidance to repair inconsistencies in

heterogeneous repositories and, therefore, are a suitable basis for flexible consistency management.

8. REFERENCES

- [1] 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.
- [2] 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. ACM, 2003.
- [3] 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, Schloß Dagstuhl, Germany*, volume 1773 of *LNCS*, pages 112–129. Springer, 2000.
- [4] D. de Champeaux. Subproblem finder and instance checker, two cooperating modules for theorem provers. *J. ACM*, 33(4):633–657, 1986.
- [5] 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.
- [6] M. Gertz and U. Lipeck. An extensible framework for repairing constraint violations. In *In Proc. 1st Working Conf. on Integrity and Internal Control in Information Systems (IICIS'97)*, pages 89–111. Chapman & Hall, 1997.
- [7] E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *Advances in Conceptual Modeling: ER '99 Workshops, Paris, France*, volume 1727 of *LNCS*, pages 62–73. Springer, 1999.
- [8] Minister of Defence. *ZDv 90/1 – Dienstvorschriften der Bundeswehr*. Department of Defence, 1983. (English: Service Manual Rules for the Federal Armed Forces).
- [9] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. of the 25th Int. Conf. on Software Engineering 2003*, Portland, OR, 2003. ACM.
- [10] J. Nordlander. *Reactive Objects and Functional Programming*. Chalmers Tekniska, Högskola, 1999.
- [11] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.
- [12] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge Univ. Press, 2003.
- [13] J. Scheffczyk, U. Borghoff, P. Rödiger, and L. Schmitz. Consistent document engineering. In *Proc. of the 2003 ACM Symp. on Document Engineering*, pages 140–149, Grenoble, France, 2003. ACM.
- [14] J. Scheffczyk, C. Stutz, U. M. Borghoff, and J. Siedersleben. Formale Konsistenzsicherung in informellen Software-Spezifikationen. *Informatik Forschung und Entwicklung*, to appear 2004. (English: Ensuring Formal Consistency of Informal Software Specifications).
- [15] K. Shafer, S. Weibel, E. Jul, and J. Fausey. Introduction to persistent uniform resource locators, 1996. see purl.oclc.org/OCCL/PURL/INET96.