Efficient (In-)Consistency Management for Heterogeneous Repositories

Jan Scheffczyk  Uwe M. Borghoff  Peter Rödig  Lothar Schmitz
Institute for Software Technology
University of the Federal Armed Forces, Munich
Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany
{jan,borghoff,roedig,lothar}@informatik.unibw-muenchen.de

Abstract

When a group of authors collaboratively edits interrelated documents, consistency problems occur almost immediately. Current document management systems (DMS) often lack adequate facilities for consistency management. We extend traditional DMS by explicit formal consistency rules. In contrast to many other approaches, we permit inconsistencies and present the consequences to the user, which is vital for flexible document management and information management. Based on a novel semantics our tools pinpoint inconsistent document parts and tell precisely when, where, and why a repository is inconsistent.

In this paper we focus on a key issue: efficient techniques for consistency rule evaluation. Our strategy is known from databases: (1) static analysis characterizes and simplifies consistency rules and (2) at run-time rules are evaluated incrementally. The major differences to databases are that we consider informal documents and explicitly allow inconsistencies. Consequently, we lack formal update descriptions and cannot rely on consistency prior to updates.

The contribution of this paper is to incrementally evaluate consistency rules in the presence of previous inconsistencies. We have implemented our techniques in a revision control system. Our experiments show that efficient incremental evaluation is the key to make our approach viable.

Key words  information management, document management, consistency, incremental evaluation

1. Introduction

Usually, 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. Managing, e.g., document versions and author access rights mainstream DMS store documents in a repository. In order to produce an overall consistent work authors have to spend much time re-reading and revising their own and related documents. Worse, each check-in to the repository can violate consistency again.

In [17] we proposed the use of explicit formal consistency rules that may be violated to manage consistency in heterogeneous repositories. Our approach allows inconsistencies, which is vital for flexible information management [10]. We employed a typed first-order temporal logic with linear time and extended traditional boolean semantics to generate consistency reports, which pinpoint inconsistent document parts.

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 paper we address the issue of efficiently evaluating first-order linear temporal formulae against a heterogeneous repository while retaining our tolerant semantics. We improve efficiency by two measures: First, we reduce checking complexity by static analysis that is performed prior to actually evaluating consistency rules. Second, we reduce the complexity of the checking algorithm itself during evaluation. Static rule analysis attempts to decrease the number of rules to be re-evaluated at a check-in by “localizing” and filtering, and to decrease the static computational complexity of a rule by rewriting. Our evaluation algorithm dynamically reduces quantifier domains. Since we allow inconsistencies in previous repository states we employ an incremental algorithm that makes heavy use of previous consistency reports.

This paper¹ is organized as follows: Sect. 2 and Sect. 3 summarize our extensions to a typical DMS, the abstract syntax of consistency rules, and our tolerant semantics. Sect. 4 forms the heart of this paper by detailing our techniques to speedup evaluation of consistency rules. Some related work is discussed in Sect. 5. Sect. 6 draws final conclusions and shows directions for future research. Throughout we use the following example from [17]. In practice we come across far more complex examples of

¹A more detailed version of this paper is published as [16].
the same kind, e.g., in [7]. Documents linking to the current version of another document and persistent URLs [19] impose closely related problems.

Example 1 Assume we want to archive manuals over a long period of time. Documents (and manuals) 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, e.g., manual kind and name. There may be many key resolvers, whose actual names are hidden from authors. 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. For example, a referenced key \( k \) is invalid if no resolver contains \( k \), a resolver maps \( k \) to a manual that does not exist, or a resolver maps \( k \) to an existing manual \( m \) but \( m \)'s kind is different from the kind of \( k \)'s resolver entry.

2. Consistency in Document Management

Formalizing consistency rules causes complexity in various problem areas. We, therefore, divide formalization into different tasks and supply tools to every stakeholder. In Fig. 2 ovals mark fixed components; rectangles mark customizable components.

From a repository, authors typically check out document working copies, modify them, and finally check them in again. Among other things, a classic DMS manages concurrent check-ins, author permissions, version control, and repository backup and distribution.

A rule designer formalizes consistency rules. We check the repository for consistency at given events, e.g., document check-in. This generates a consistency report. Consistency rules use function and predicate symbols from domain specific languages. A type system ensures well-typedness of rules w.r.t. the language used.

A language designer declares valid function symbols, predicate symbols, and types. He defines symbol semantics in Haskell [12] – a statically typed purely functional programming language. The use of Haskell is motivated by our incremental techniques, which we discuss in Sect. 4.

For specific projects, a project manager chooses consistency rules. In some cases adaptions will be necessary, e.g., some rules can be weakened.

3. Formalizing Rules

In this section we introduce the syntactical means to formalize consistency rules and sketch how they are evaluated. For details see our companion paper [17].

Rule designers formalize consistency rules in a variant of the two-sorted temporal first-order predicate logic with linear time and equality [1]. The two-sorts approach to temporal logic introduces a new temporal sort Time. To each non-temporal predicate or function symbol a timestamp is added – we call these symbols partially temporal.

Fig. 3 shows part of our term and formula language – Fig. 3. Formal consistency rule syntax

\[
\phi, \psi ::= p(e_1, \ldots, e_n) \quad \text{atomic formula} \\
| \neg \phi \quad \text{negation} \\
| \phi \cdot \psi \quad \text{junction, } \cdot \in \{\lor, \land, \Rightarrow\} \\
| Q x \in \phi \quad \text{quantified formula, } Q \in \{\forall, \exists\} \\
| e ::= x \quad f \quad \text{variable; (function) symbol} \\
| f(e_1, \ldots, e_n) \quad \text{application}
\]

Fig. 2. System overview
At each time links must be valid:

“At any time \( t \) 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 \( d \).

\[
\forall t \in \text{repStates} \cdot \forall x \in \text{repDa}(t) \cdot \forall k \in \text{refs}(x) \cdot \\
\exists d \in \text{concatMap}(t, \text{kDefs}, \text{repResDs}(t)) \cdot \\
\exists m \in \text{repManDs}(t) \cdot k = \text{key}(d) \land \\
\text{id}(m) = \text{kId}(d) \land \text{kind}(m) = \text{kKind}(d)
\]

\[\text{Figure 4. Example rule } r_1\]

Each symbol has a unique name. Due to their first temporal parameter the meaning of partially temporal function symbols, e.g., \( \text{kDefs} \). We can use them as arguments of higher-order functions, e.g., in \( \text{concatMap}(t, \text{kDefs}, \text{repResDs}(t)) \). For brevity we omit the formal declarations for function symbols and predicate symbols used by our example.

Our tolerant semantics pinpoints the trouble spots that make a repository inconsistent w.r.t. the rules formalized. The basic goal of our semantics is to provide authors with as few as possible information that precisely characterize when, where, and why a repository is inconsistent. Extending xlinkit [8] we compute for each rule a consistency report containing a boolean result (representing boolean truth semantics) and a diagnosis set. Each diagnosis consists of a consistency flag, a variable assignment, fulfilled predicates, and failed predicates. A diagnosis \((C, \text{as}, \text{ps}_I, \text{ps}_F)\) reads: “The processed formula is fulfilled (Consistent) for variable assignments \( \text{as} \) due to true predicates \( \text{ps}_I \) and false predicates \( \text{ps}_F \).” Similarly \((\text{IC}, \text{as}, \text{ps}_I, \text{ps}_F)\) indicates that the formula is not fulfilled (InConsistent). The assignment \( \text{as} \) maps variables to concrete values, e.g., timestamps or documents. Due to temporal quantification it tells when and where a rule is not fulfilled. The predicate sets \( \text{ps}_I \) and \( \text{ps}_F \) describe why a rule is not fulfilled. Evaluating rule \( r_1 \) illustrates the above:

**Example 2** The rule designer formalizes the first rule as shown in Fig. 4. Rule \( r_1 \) first quantifies over all states in the repository, provided by \( \text{repStates} \). The function \( \text{repDa} \) returns the current documents for a state \( t \). For each document \( x \) the variable \( k \) comprises the referenced keys, computed via \( \text{refs} \). Key definitions in the resolvers are computed via the higher-order function \( \text{concatMap} \). It applies \( \text{kDefs} \) to \( t \) and each key resolver, computed via \( \text{repResDs}(t) \). Applied to a key resolver \( \text{kDefs} \) returns its key definitions in a list. 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 the state \( t \), computed via \( \text{repManDa} \). For partially temporal symbols that do not depend on time, e.g., \( \text{refs} \), we can omit the temporal parameter.

The language designer defines the types shown in Fig. 5, where we omitted the definitions of the list type \([\] \) and the type \( \text{String} \). A subtype relation between record types is denoted by \(<\). Record labels induce new partially temporal function symbols, e.g., \( \text{kDefs} \). We can use them as arguments of higher-order functions, e.g., in \( \text{concatMap}(t, \text{kDefs}, \text{repResDs}(t)) \). For brevity we omit the formal declarations for function symbols and predicate symbols used by our example.

Our tolerant semantics pinpoints the trouble spots that make a repository inconsistent w.r.t. the rules formalized. The basic goal of our semantics is to provide authors with as few as possible information that precisely characterize when, where, and why a repository is inconsistent. Extending xlinkit [8] we compute for each rule a consistency report containing a boolean result (representing boolean truth semantics) and a diagnosis set. Each diagnosis consists of a consistency flag, a variable assignment, fulfilled predicates, and failed predicates. A diagnosis \((C, \text{as}, \text{ps}_I, \text{ps}_F)\) reads: “The processed formula is fulfilled (Consistent) for variable assignments \( \text{as} \) due to true predicates \( \text{ps}_I \) and false predicates \( \text{ps}_F \).” Similarly \((\text{IC}, \text{as}, \text{ps}_I, \text{ps}_F)\) indicates that the formula is not fulfilled (InConsistent). The assignment \( \text{as} \) maps variables to concrete values, e.g., timestamps or documents. Due to temporal quantification it tells when and where a rule is not fulfilled. The predicate sets \( \text{ps}_I \) and \( \text{ps}_F \) describe why a rule is not fulfilled. Evaluating rule \( r_1 \) illustrates the above:

**Example 3** For brevity we omit Haskell sources of function and predicate symbol implementations. Fig. 6 shows a small repository containing a text document, a key resolver, and a manual the kind of which changed during the transition from state 1 to state 2. The report for the rule \( r_1 \) reflects the inconsistency introduced at state 2.

\[
\text{False, } \{ \text{IC, } \{ t \mapsto 2, k \mapsto \text{kaAlc3}, \} \}, \{ \text{id, } \{ \text{doc1.txt, time } = 1 \} \} , \{ \text{0, } \{ \text{kind}(m) = \text{kKind}(d) \} \}
\]

The assignment above lacks \( d \) and \( m \) – the repository is inconsistent w.r.t. \( r_1 \) for all possible assignments to \( d \) and \( m \). The report also lacks \( k = \text{key}(d) \) and \( \text{id}(m) = \text{kId}(d) \) – the key resolver contains \( \text{kaAlc3} \), but its 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.
of rules to be re-evaluated at a repository check-in. A rule is simplified by minimizing its quantifier nesting and thus reducing the static evaluation time complexity. We shall only sketch our methods here because they are straightforward adoptions to techniques known from databases [3, 4].

When the DMS signals that the repository has changed, we want to re-evaluate only those consistency rules that might be affected by these updates. With each rule we associate a set of affected documents because we also want to extend revision control systems, such as CVS, that lack a formal document model. Rule localization depends on appropriate metadata for functions and predicates. Static Haskell code analysis helps the language designer to add these metadata. Then, a straightforward algorithm associates a consistency rule with a document set. For example, it associates the rule \( r_1 \) with all text documents and all XML documents (* .txt | *.xml). That is because \( \text{repDs} \) accesses text documents and XML documents (* .txt | *.xml), \( \text{repPlanDs} \) accesses manuals only (\( \text{man}* .xml \)), \( \text{repResDs} \) accesses key resolvers only (\( \text{key}* .xml \)), and \( \text{refs} \) accesses the documents of its second argument only.

The computational cost to evaluate a consistency rule depends on the deepest quantifier nesting, which is minimal if the scope of each quantifier is minimal. Such formulae are called miniscope [2]. Consider the rule \( r_1' \):

\[
r_1' = \forall t \in \text{repStates} \bullet \forall x \in \text{repDs}(t) \bullet \forall k \in \text{refs}(x) \bullet \exists d \in \text{concatMap}(t, k\text{Defs}, \text{repResDs}(t)) \bullet
\]

\[ k = \text{key}(d) \land
\]

\[ (\exists m \in \text{repManDs}(t) \bullet \text{id}(m) = \text{id}(d) \land \text{kind}(m) = \text{kind}(d))
\]

Here, the existential quantifier over \( m \) was moved into the conjunction, which is sound because \( k = \text{key}(d) \) does not contain \( m \). Since the existential quantifier has a smaller scope now, \( r_1' \) can be evaluated faster than \( r_1 \). We adapt the techniques in [2] to convert our rules to miniscope. Miniscopy also removes implications, pushes negations into formulae, and “flattens” nested conjunctions and disjunctions. Our incremental evaluation algorithm benefits from these simplifications.

How does static analysis improve evaluation time? The second last column of Fig. 10 (see end of Sect. 4.3) shows some benefits, especially when the second rule is not re-evaluated. Improvements achieved by miniscopy can be seen in states where both rules are re-evaluated. The results are, however, unsatisfactory: Still, rule evaluation time depends on the repository state. One of the major reasons is that we access documents at \( \text{previous} \) repository states. Usually, the DMS rebuilds these documents step by step using state transition descriptions, e.g., diffs or patches.

4.4.1 Static Analysis

Static analysis tries to localize and simplify consistency rules before they are evaluated against a repository. It is performed almost exclusively on a syntactical level. Localizing a rule means to associate it with the set of those documents that can possibly affect it — thus reducing the number of rules to be re-evaluated at a repository check-in. A rule is simplified by minimizing its quantifier nesting and thus reducing the static evaluation time complexity. We shall only sketch our methods here because they are straightforward adoptions to techniques known from databases [3,4].

When the DMS signals that the repository has changed, we want to re-evaluate only those consistency rules that might be affected by these updates. With each rule we associate a set of affected documents because we also want to extend revision control systems, such as CVS, that lack a formal document model. Rule localization depends on appropriate metadata for functions and predicates. Static Haskell code analysis helps the language designer to add these metadata. Then, a straightforward algorithm associates a consistency rule with a document set. For example, it associates the rule \( r_1 \) with all text documents and all XML documents (* .txt | *.xml). That is because \( \text{repDs} \) accesses text documents and XML documents (* .txt | *.xml), \( \text{repPlanDs} \) accesses manuals only (\( \text{man}* .xml \)), \( \text{repResDs} \) accesses key resolvers only (\( \text{key}* .xml \)), and \( \text{refs} \) accesses the documents of its second argument only.

The computational cost to evaluate a consistency rule depends on the deepest quantifier nesting, which is minimal if the scope of each quantifier is minimal. Such formulae are called miniscope [2]. Consider the rule \( r_1' \):

\[
r_1' = \forall t \in \text{repStates} \bullet \forall x \in \text{repDs}(t) \bullet \forall k \in \text{refs}(x) \bullet \exists d \in \text{concatMap}(t, k\text{Defs}, \text{repResDs}(t)) \bullet
\]

\[ k = \text{key}(d) \land
\]

\[ (\exists m \in \text{repManDs}(t) \bullet \text{id}(m) = \text{id}(d) \land \text{kind}(m) = \text{kind}(d))
\]

Here, the existential quantifier over \( m \) was moved into the conjunction, which is sound because \( k = \text{key}(d) \) does not contain \( m \). Since the existential quantifier has a smaller scope now, \( r_1' \) can be evaluated faster than \( r_1 \). We adapt the techniques in [2] to convert our rules to miniscope. Miniscopy also removes implications, pulls negations into formulae, and “flattens” nested conjunctions and disjunctions. Our incremental evaluation algorithm benefits from these simplifications.

How does static analysis improve evaluation time? The second last column of Fig. 10 (see end of Sect. 4.3) shows some benefits, especially when the second rule is not re-evaluated. Improvements achieved by miniscopy can be seen in states where both rules are re-evaluated. The results are, however, unsatisfactory: Still, rule evaluation time depends on the repository state. One of the major reasons is that we access documents at \( \text{previous} \) repository states. Usually, the DMS rebuilds these documents step by step using state transition descriptions, e.g., diffs or patches.

4. Speeding up Consistency Checking

In this section we consider static and dynamic methods to decrease the computational complexity of evaluating first-order consistency rules. For simplicity we shall neglect the individual computational complexity of functions and predicates. We designed our methods to make as few as possible assumptions to the underlying DMS. We only require (1) a repository to access past documents, (2) a locking mechanism that avoids updates during a consistency check, and (3) that the DMS signals the documents added or changed by an update. Since we do not require a specific document model our techniques also apply to revision control systems, such as CVS or darc.

Due to space restrictions we exemplify our techniques for rule \( r_1 \) only.

4.1 Static Analysis

Static analysis tries to localize and simplify consistency rules before they are evaluated against a repository. It is performed almost exclusively on a syntactical level. Localizing a rule means to associate it with the set of those documents that can possibly affect it — thus reducing the number of rules to be re-evaluated at a repository check-in. A rule is simplified by minimizing its quantifier nesting and thus reducing the static evaluation time complexity. We shall only sketch our methods here because they are straightforward adoptions to techniques known from databases [3, 4].

When the DMS signals that the repository has changed, we want to re-evaluate only those consistency rules that might be affected by these updates. With each rule we associate a set of affected documents because we also want to extend revision control systems, such as CVS, that lack a formal document model. Rule localization depends on appropriate metadata for functions and predicates. Static Haskell code analysis helps the language designer to add these metadata. Then, a straightforward algorithm associates a consistency rule with a document set. For example, it associates the rule \( r_1 \) with all text documents and all XML documents (* .txt | *.xml). That is because \( \text{repDs} \) accesses text documents and XML documents (* .txt | *.xml), \( \text{repPlanDs} \) accesses manuals only (\( \text{man}* .xml \)), \( \text{repResDs} \) accesses key resolvers only (\( \text{key}* .xml \)), and \( \text{refs} \) accesses the documents of its second argument only.

The computational cost to evaluate a consistency rule depends on the deepest quantifier nesting, which is minimal if the scope of each quantifier is minimal. Such formulae are called miniscope [2]. Consider the rule \( r_1' \):

\[
r_1' = \forall t \in \text{repStates} \bullet \forall x \in \text{repDs}(t) \bullet \forall k \in \text{refs}(x) \bullet \exists d \in \text{concatMap}(t, k\text{Defs}, \text{repResDs}(t)) \bullet
\]

\[ k = \text{key}(d) \land
\]

\[ (\exists m \in \text{repManDs}(t) \bullet \text{id}(m) = \text{id}(d) \land \text{kind}(m) = \text{kind}(d))
\]

Here, the existential quantifier over \( m \) was moved into the conjunction, which is sound because \( k = \text{key}(d) \) does not contain \( m \). Since the existential quantifier has a smaller scope now, \( r_1' \) can be evaluated faster than \( r_1 \). We adapt the techniques in [2] to convert our rules to miniscope. Miniscopy also removes implications, pushes negations into formulae, and “flattens” nested conjunctions and disjunctions. Our incremental evaluation algorithm benefits from these simplifications.

How does static analysis improve evaluation time? The second last column of Fig. 10 (see end of Sect. 4.3) shows some benefits, especially when the second rule is not re-evaluated. Improvements achieved by miniscopy can be seen in states where both rules are re-evaluated. The results are, however, unsatisfactory: Still, rule evaluation time depends on the repository state. One of the major reasons is that we access documents at \( \text{previous} \) repository states. Usually, the DMS rebuilds these documents step by step using state transition descriptions, e.g., diffs or patches.\(^{2}\)

\(^{2}\)Tests are performed on a Dell X200 laptop; 800 MHz PIII CPU.
Next, we discuss dynamic methods, which attempt to avoid accessing past document versions.

### 4.2. Incremental Evaluation

The major goals of incremental evaluation are: (1) access only current document versions and (2) as far as possible access changed or new documents only. A very simple strategy would be the following: Static rules quantify only once over all repository states and lack calculations of time, e.g., \( r_1 \). Thus we can copy the report from the previous evaluation and evaluate the rule w.r.t. the new repository state only. This, of course, breaks down for temporal rules (e.g., \( r_2 \)), which relate different repository states.

We, therefore, developed a general technique that also applies to temporal rules. As usual, it is most challenging to balance the speedup against the space needed for storing auxiliary information. Our approach needs only few additional information [16]. Instead it exploits previous reports 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 last evaluation.
2. If possible re-evaluate a rule only on new or changed domain values. For old domain values copy the relevant part from the old report.

A fundamental prerequisite to incremental evaluation is that the consistency report of a rule remains constant if its quantifier domains remain constant. This requires that the result of a function or predicate depends on its parameters only – a feature called referential transparency in functional programming. In general, Haskell guarantees referential transparency. Since in our approach language designers use Haskell to define symbol semantics, the strategy above is sound. A notable exception is the “unsafe” function \( \text{repStates} \), which is not referentially transparent because it reads the number of already performed check-ins directly from the repository [16].

To support incremental evaluation, the language designer associates with each function symbol and each predicate symbol those parameters it actually depends on. For example, the partially temporal function symbol \( \text{repents} \) does not depend on its temporal parameter. Within rules a straightforward algorithm marks subformulas and quantifier bounding terms with the variables they actually depend on. Below, we have marked rule \( r'_1 \) where variables appear as superscripts. (\( \ast \) means that a subformula or quantifier bounding term contains \( \text{repStates} \), i.e., it is unsafe.)

\[
\forall v \forall t \in \text{repStates}^\ast \cdot \forall v \forall x \in \text{repDa}(t) \cdot \forall v \forall k \in \text{refs}(x) \cdot \\
\exists \gamma, \delta \ k \in \text{concatMap}(\gamma, \text{KDef}, \text{repDa}(t)) \cdot \\
k = k_0 \cdot \text{key}(d) \land ^\gamma, \delta (\exists \gamma', \delta', m \in \text{repManDa}(t) \cdot id(m) = m \cdot \text{kind}(m) = m \cdot \text{kind}(d))
\]

**Example 4** Consider a check-in of a new manual man2.xml at state 3 also referencing the key kaAlc3:

\[
\text{<man kind="field M."}> ... \text{<see>kaAlc3</see>} ... \text{</man>}
\]

We have to re-evaluate both rules. Fig. 8 shows how our incremental algorithm evaluates rule \( r'_1 \). In the tree vertices represent conjunctions, disjunctions, or quantifier domains where old values are grey and new values are black. A path from the root to a leaf can be seen as an assignment.

Evaluating \( t \)'s domain results in the sets \( \text{new} = \{ 3 \}, \text{chg} = \emptyset, \text{old} = \{ 1, 2 \}, \text{and del} = \emptyset \). Since \( t \)'s subformula depends only on \( t \) and is referentially transparent we can abort re-evaluation for values in \( \text{old} \). Instead, we copy relevant states from the old report \( @ \), which results in a true report \( \{ \text{True}, \emptyset \} \) for \( t \to 1 \) and the following report for \( t \to 2 \) (assignments in reports lack variable markers):

\[
\text{False.}
\]

\[\left[ \{ \text{True}, \emptyset \} \right. \text{for } t \to 1 \text{ and the following report for } t \to 2 (\text{assignments in reports lack variable markers})\]

The report above lacks the binding \( t \to 2 \), which is removed while copying. When we finally evaluate \( t \)'s quan-
Due to our strategy we copied some part of the old report.

\[ \mathcal{D}_{\text{TL}}^b[p(e_0, e_1, \ldots, e_n)^n] \eta \quad (p \text{ partially temporal}) \]
\[ = \text{copy}(\bar{\beta}, \eta) \text{ if notEval}(b, xx, \eta) \]
\[ (\text{True, } \{ (C, \emptyset, \{p(e_0, e_1, \ldots, e_n), \emptyset) \}) \]
\[ \text{if } (\forall x_1 \exists x_2, \forall x_3 \exists x_4, \forall x_5 \exists x_6 \eta \in p^4) \]
\[ \text{otherwise} \]

where \( \bar{A} = I \) \( (\forall x_1 \exists x_2, \forall x_3 \exists x_4, \forall x_5 \exists x_6 \eta) \);

\[ \mathcal{D}_{\text{TL}}^b[p(e_1, e_2, e_3, e_4)^4] \eta \quad (p \text{ temporal}) \]
\[ = \text{copy}(\bar{\beta}, \eta) \text{ if notEval}(b, xx, \eta) \]
\[ (\text{True, } \{ (C, \emptyset, \{p(e_1, e_2, e_3, e_4), \emptyset) \}) \]
\[ \text{if } (\forall x_1 \exists x_2, \forall x_3 \exists x_4, \forall x_5 \exists x_6 \eta \in p^4) \]
\[ \text{otherwise} \]

where \( \bar{A} = I \) \( (\forall x_1 \exists x_2, \forall x_3 \exists x_4, \forall x_5 \exists x_6 \eta) \);

\[ \mathcal{D}_{\text{\bar{\beta}}}^b[\phi \land \psi \eta] = \text{copy}(\bar{\beta}, \eta) \text{ if notEval}(b, xx, \eta) \]
\[ \text{if } \text{fst}(r_0) = \text{fst}(r_0) = \text{False} \]
\[ \text{True, } \emptyset \}
\[ \text{otherwise} \]

where \( r_0 = \mathcal{D}^b_{\text{\bar{\beta}}}[^{\phi \land \psi} \eta];
\]

\[ \mathcal{D}_{\text{\bar{\beta}}}^b[\phi \land \psi \eta] = \text{copy}(\bar{\beta}, \eta) \text{ if notEval}(b, xx, \eta) \]
\[ \text{if } \text{fst}(r_0) = \text{fst}(r_0) = \text{False} \]
\[ \text{True, } \emptyset \}
\[ \text{otherwise} \]

where \( r_0 = \mathcal{D}^b_{\text{\bar{\beta}}}[^{\phi \land \psi} \eta];
\]

Due to our strategy we copied some part of the old report during evaluation of the new repository state \((t \to 3)\). Notice, however, that bindings to the current state 3 are lifted to the previous state 2 because the old report cannot contain a binding \(t \to 3\).
For every formula notEval(b, xs, η) determines whether it appears immediately below a quantifier, it depends only on variables xs marked as old in the current assignment η, and contains referentially transparent functions and predicates only. In this case we copy the relevant states from the old report \( \mathcal{R} \).4 (copy replaces bindings to the current repository state with bindings to the previous repository state.)

The most important case is an existentially quantified formula. Due to miniscoping all quantifiers occur positively in a formula. So existential quantifiers cannot appear “disguised” as negated universal quantifiers. We first compute the domain, resulting in the four sets new\( v \), chg, old\( v \), and del. The general idea is to mark the values in new and chg as new (n) and the values in old as old (o). Then the subformula is evaluated for possible assignment extensions to these values (\( η[x → (v, n)] \) and \( η[x → (v, o)] \), respectively). Alas this is only sound if no values were changed or deleted in the domain, i.e., chg = del = ∅.

Figure 10. Improvements by static analysis (4.1) and incremental evaluation (4.3)

<table>
<thead>
<tr>
<th>state repository changes</th>
<th>rules</th>
<th>CPU time (4.1)</th>
<th>CPU time (4.1 &amp; 4.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 txt; ln, key; ln, man; 9n                                           r(_1), r(_2) 4.44 sec.</td>
<td>4.47 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 txt; lc, 4n                                                        r(_1)  9.55 sec.</td>
<td>2.33 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 man; 2c                                                            r(_1), r(_2) 15.25 sec.</td>
<td>6.45 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 man; 2c                                                            r(_1), r(_2) 20.40 sec.</td>
<td>6.58 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 txt; lc                                                            r(_1) 23.46 sec.</td>
<td>2.68 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 txt; lc                                                            r(_1) 27.98 sec.</td>
<td>2.53 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 key; ln                                                            r(_1) 32.50 sec.</td>
<td>3.09 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 man; ln                                                            r(_1), r(_2) 44.66 sec.</td>
<td>4.01 sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 key; lc                                                            r(_1) 45.34 sec.</td>
<td>3.61 sec.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4A state in the old report may contain an assignment more detailed or more general than η. Therefore, we copy all states containing assignments that subsume or are subsumed by η. An assignment \( a\_3 \) subsumes an assignment \( a\_2 \) if each variable binding in \( a\_3 \) also occurs in \( a\_2 \), where we neglect variable markers.

that evaluation of existential quantifiers can be slower than evaluation of universal quantifiers. Universal quantifiers do not suffer from changed or deleted domain values because our reports already store the values that make a rule inconsistent – they pinpoint inconsistencies.

Let us return to our toy repository. What does incremental evaluation buy? Fig. 10 shows the performance of our consistency checker using both static analysis and incremental evaluation. Now, evaluation time depends rather on the changed content than on the repository state. Notice that except for states 3 and 4 every evaluation is faster than the initial evaluation although the repository grows. The simple initial strategy from Sect. 4.2 cannot achieve this.

5. Related Work

Since managing consistency is a fundamental problem a huge body of related work exists; see, e.g., [8, 17]. Here, we focus on incremental consistency checking and discuss only some closely related work in this area. 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 [6] provide a uniform approach to incremental computation. They attempt to minimize redundant computation provided that a program runs repeatedly on slightly different inputs. To produce the current results of a program run, incremental computation uses previous results, differences between previous inputs and current inputs, and auxiliary information. We can regard quantified formulae as “functions” having the quantifier domain as input, which is reduced during incremental evaluation.

From its origin, the database community has been striving for efficient algorithms that check integrity constraints [14, 3, 18, 4]. Usually, these approaches employ the following techniques: Compile-time analysis simplifies integrity constraints and identifies update types that might violate constraints. Run-time approaches use previous results to avoid re-computation. In the context of first-order logic they reduce quantifier domains.

In general, we follow the database community approach but lack a formal database schema, formalized updates, and consistency prior to updates. This makes our approach more complicated. We cannot benefit from constraint subsumption because this is undecidable in our rule language; miniscoping performs only some very simple subsumption analysis. Localizing rules roughly corresponds to using update information in [4]. In order to gain the fine granularity needed for database approaches we parse documents. Consequently, our approach would lack efficiency if we implemented only strict rules. Tolerating inconsistencies, however, allows to defer evaluation of weak rules and to
acknowledge or prohibit a check-in after the strict consistency rules have been evaluated. In (temporal) databases the problem of efficiently storing and managing historical database states arises. In our setting, historical repository states are already stored and managed by the DMS.

Recently, incremental evaluation techniques were used by the XML community to maintain consistency w.r.t. user defined rules [5]. The consistency checking toolkit xlinkit [11] uses static analysis to filter consistency rules relevant to document changes and a treediff algorithm to determine 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 allowing distribution and avoiding a history-aware repository, xlinkit cannot implement temporal consistency rules and lacks several incremental techniques we can benefit from.

6. Conclusion and Outlook

We have shown how first-order consistency rules can be evaluated efficiently by employing incremental techniques. Exploiting domain knowledge supplied by the language designer, we statically analyze consistency rules to (1) associate with each rule a document set the rule depends on, and (2) miniscope rules to reduce their static evaluation time complexity. At run-time we re-evaluate a rule only on new and changed documents if possible. It turned out that Haskell’s referential transparency provides fundamental support for the soundness of our techniques. We provided concrete performance measurements proving that static analysis combined with incremental evaluation provides significant speedup compared to brute force evaluation. We conjecture that our incremental evaluation algorithm could be of value in other research areas.

In the future, we will focus on fine tuning our incremental approach based on further experiments. Sophisticated caching techniques, which consider function and predicate properties (such as transitivity), become important when we evaluate complex functions and predicates. Function and predicate properties also may prove useful for taking advantage of rewriting techniques in the Haskell compiler GHC [13]. Furthermore, we develop strategies to suggest reasonable repair actions that can resolve inconsistencies.

Extending our previous work [17], the contributions of this paper provide the key to make our approach to consistency management viable.

References