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

In this paper, we propose to use explicit formal consistency rules for heterogeneous repositories that are managed by traditional DMS. Rules are formalized in a variant of first-order temporal logic. Functions and predicates, implemented in a full programming language, provide complex (even higher-order) functionality. A static type system supports rule formalization, where types also define (formal) document models. In the presence of types, the challenge is to smoothly combine a first-order logic with a useful type system including subtyping. In implementing a tolerant view of consistency, we do not expect that repositories satisfy consistency rules. Instead, a novel semantics precisely pinpoints inconsistent document parts and indicates when, where, and why a repository is inconsistent.

Our major contributions are (1) the use of explicit formal rules giving a precise (and still comprehensible) notion of consistency, (2) a static type system securing the formalization process, (3) a novel semantics pinpointing inconsistent document parts and indicates when, where, and why a repository is inconsistent, and (4) a design of how to automatically check consistency for document engineering projects that use existing DMS. We have implemented a prototype of a consistency checker. Applied to real world content, it shows that our contributions can significantly improve consistency in document engineering processes.

Categories and Subject Descriptors
H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing—Abstracting methods, Linguistic processing; I.7.1 [Document and Text Processing]: Document and Text Editing—Document management

General Terms
Theory, Algorithms, Management, Design.

Keywords
document management, consistency in document engineering, temporal logic

1. INTRODUCTION
Larger works of writing – e.g., books, technical documentations, or software specifications – contain many documents1 that are collaboratively and concurrently edited by a number of authors. Mainstream DMS store documents in repositories and provide version control, rights management etc. Usually, authors aim to produce an overall consistent work, i.e., certain relations between the documents are maintained. These relations, however, are mostly implicit and vague, e.g., “Links inside documents must have a valid target.” In order to achieve consistency, authors have to spend much time re-reading and revising their own and related documents. Worse, each check-in to the repository potentially invalidates consistency. Larger companies define guidelines and policies for writing; but still, a human reviewer is required to enforce them. What prevents automatic checks is that guidelines are implicit or at least informal. Yet recent proposals for managing XML documents appear to neglect these shortcomings [33].

We, therefore, propose the use of explicit formal consistency rules to manage consistency in heterogeneous repositories. Strict rules must be adhered to, whereas weak rules may be violated. In addition, each rule has a priority, which allows to gauge the impact of an inconsistency. Rules may restrict how documents evolve in time – hence we employ temporal logic. Since rule design is a complex task, a static

---

1We use the term “document” informally here, being aware that documents do not need to follow a formal document model. Using XML, however, we gain several advantages.
type system helps to define syntactically well-formed consistency rules. Being aware that check-ins potentially violate consistency rules, we replace traditional boolean semantics by a novel semantics, which pinpoints inconsistencies within documents.

This provides automatic checks that indicate precisely when, where, and why inconsistencies occur and opens the road to tolerating inconsistencies – a necessity in many areas [10, 17]. If documents evolve at different rates enforcing consistency may cause deadlocks. Consistency rules may be too strict for some purposes – inconsistencies may indicate exceptions or design alternatives. Finally, the impact of an inconsistency can be low compared to the costs of resolving it.

The following running example illustrates the formal part of our paper. In practice we come across far more complex examples of the same kind, e.g., in [14]. Documents linking to the current version of another document and persistent URLs [26] raise 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 exist many key resolvers, the actual names of which are hidden from authors. To ensure consistency we require that (1) (two-step) links 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 $k$, or 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.

This paper is organized as follows: Sect. 2 shows how we integrate our work into document engineering processes that are based on DMS. Sect. 3 describes how consistency rules are formalized. Sect. 4 sketches our static type system. Sect. 5 shows our tolerant semantics and how it helps to precisely detect inconsistencies. Notes on our implementation and results from our experiments can be found in Sect. 6. In Sect. 7 we discuss some related work. A summary and directions for future research are given in Sect. 8.

### 2. USING CONSISTENCY RULES IN DOCUMENT ENGINEERING

We consider formalizing consistency rules a complex process. In order to handle this additional complexity, we divide formalization into different tasks and supply tools to every stakeholder (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 rights, version control, and repository backup and distribution. We design our consistency checker in a way that makes only few assumptions about the DMS. We require (1) a facility to access past and present document versions and (2) a locking mechanism that prevents check-ins during a consistency check. We make no assumptions about the document model of the DMS; so our extensions also apply to revision control systems like CVS [4].

A *rule designer* formalizes consistency rules. Rules define what it means for the repository to be consistent – they reflect wishes from the “administrator” perspective. We check the repository for consistency w.r.t. the rules formalized at given events, e.g., document check-in or the end of a development phase. Checking the repository for consistency 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 strict rule is violated the system will reject 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.

Consistency rules use function and predicate symbols from a domain specific language. This makes the rules completely 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. If, e.g., a predicate symbol $=$ requires two arguments of the same type the careless application of $=$ to a number and a string 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.

A *language designer* declares valid symbols and their types in a signature, and implements symbol semantics in the
3. FORMALIZING CONSISTENCY RULES

Our examples show that we need a very expressive language to formalize consistency rules. We require a temporal component as well as complex functions and predicates. Besides being precise rule design should be comfortable: Once defined, we want to reuse functions and predicates as often as possible. This is why we have decided to formalize consistency rules in a full first-order temporal predicate logic.\(^2\) We allow polymorphic (even higher-order) functions and predicate to facilitate comfort and reuse. The challenge is, therefore, to smoothly combine a first-order logic with higher-order functions and predicates. Our type checker, sketched in Sect. 4, provides significant support for this.

3.1 Overview

Consistency rules are statements about repository states, which we call timestamps because they represent given points in time. Similar to our architecture, 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 symbols (that can be used in rules) in a signature and implements symbol semantics in Haskell.

Rule designers formalize consistency rules in a variant of two-sorted temporal first-order logic with linear time and equality [1, 11]. 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. Fully temporal predicate or function symbols only have temporal arguments (and results). Quantifiers iterate over variables of the sort Time.

\(^2\)For example, the language designer would declare \( = \) to have the polymorphic type \( \forall \alpha. \alpha \times \alpha \) (\( \alpha \) denotes a type variable, \( \times \) separates argument types). A Haskell function then defines the meaning of \( = \), e.g., via instances of the type class Eq.

\(^3\)We employ a first-order logic only because based on our experiments we have not seen the need for higher-order logic.

(1) At each time links must be valid:

\[ \forall t, t_1, t_2 \exists \text{resStates} \exists k \exists \text{refs}(x) \exists d \exists \text{repResDs}(t) \text{ such that } \] 
\[ (x, k) \in \text{refs}(x) \land k \in \text{key}(d) \land \text{repResDs}(t) \] 
\[ \exists m \in \text{repManDs}(t) \] 
\[ \text{dName}(m) = \text{kId}(d) \land \text{kind}(m) = \text{kKind}(d) \] 

(2) Names and kinds of manuals are invariant over time:

\[ \forall t, t_1, t_2 \exists \text{repStates} \exists \text{m1} \exists \text{m2} \] 
\[ \text{m1} \in \text{repManDs}(t_1) \land t_1 \leq t_2 \Rightarrow \] 
\[ \exists m_2 \in \text{repManDs}(t_2) \] 
\[ \text{dName}(m_1) = \text{dName}(m_2) \land \text{kind}(m_1) = \text{kind}(m_2) \] 

Figure 3: Example consistency rules (for function and predicate symbols see Fig. 4)

too. We use the two-sorts approach because (1) timestamp variables make temporal logic more expressive, (2) rule designers do not need to learn temporal connectives, and (3) the introduction of types makes the two-sorts approach straightforward. In our setting Time is a type.

EXAMPLE 2. The rule designer defines consistency rules like those shown in Fig. 3. If a partially temporal symbol does not depend on time we can omit its temporal parameter (like \( \text{refs}(x) \) in \( \phi_1 \)). We call such applications “unsaturated.” Predicate symbols are written in infix notation.

We formalize rule \( \phi_1 \) by first quantifying over all states in the repository, provided by \( \text{repStates} \). Then, for each state \( t \) we need the current documents \( x \), obtained from \( \text{repDs}(t) \), and the referenced keys \( k \) therein, computed via \( \text{refs}(x) \). Since \( \text{refs} \) is independent of time we omit its temporal parameter. For every \( k \) there must exist a key definition \( d \) such that the key defined by \( d \) (computed via \( \text{key}(d) \)) equals the referenced key \( k \). Furthermore, there must exist a manual \( m \) whose name equals the identifier mentioned by \( d \) (we get the current manuals via \( \text{repManDs}(t) \)). In addition, the kind of \( m \) must equal the kind mentioned by \( d \). We get the current key definitions \( d \) via \( \text{concatMap}(t, \text{kDefs}, \text{repResDs}(t)) \). This computes the current resolver documents \( \text{repResDs}(t) \) and extracts the key definitions from the resolver documents. So finally, \( d \) iterates over all key definitions inside all resolvers. Essentially, \( \text{concatMap} \) behaves like a universal quantifier here. Assume we wanted to neglect case when comparing manual kinds. Then we simply use a different predicate symbol, say \( = \). This is beyond the possibilities of XLink.

In rule \( \phi_2 \) we quantify twice over time because we have to relate different versions of manuals, where one version (at state \( t_1 \)) is “older” than the other (at state \( t_2 \)).

The rules in Fig. 3 appear more complex than the vague “ideas” in our introductory example. Formal rules are much more precise than informal guidelines and give no room for misinterpretations. We consider this an important feature of formalization: When formalizing consistency rules we have to decide what we really want. In our experience this contributes to a common understanding of what consistency actually means. This is vital for any collaborative work.
3.2 Abstract Syntax

Fig. 5 summarizes the abstract syntax of consistency rules – the rule designer toolset. More often than not, formulae are standard first-order formulae. A quantifier Q introduces a bound variable x, restricted by a term e that evaluates to the domain of x. We use terms for quantifier restrictions to easily identify variable domains. We call closed formulae consistency rules and non-empty finite rule sets rule systems. Typically, a rule contains additional metadata, e.g., weakness annotations and its priority.

Our notion of terms corresponds to Nordlander’s Haskell extension O’Haskell [16], 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; construct a record of type R in order to guide type inference. Both case constructs take an additional temporal argument e, fixing the time of evaluation. An optional variant type constructor V uniquely identifies the type of e, thus avoiding ambiguities that may arise from subtyping. In the logic used here it is undecidable whether a formula is satisfiable. This, however, is no issue in our setting because rules are evaluated against concrete repositories. What we are really interested in are concrete inconsistency pointers generated from complex consistency rules! We, therefore, 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 in predicate logic, which has been proven to be undecidable (see, e.g., [13]). Thus, we only perform analyses that detect some cases of rule contradiction and implication [6].

The language designer declares valid symbols for rule definition in a signature. Each symbol has a unique name. Partially temporal predicate symbols in P have a timestamp as first parameter; their meaning may change over time. Fully temporal predicate symbols in P have only temporal parameters. A similar distinction holds for function symbols. We require a predicate symbol ≤, interpreted by a total ordering relation to facilitate linear time logic. We further require a function symbol * which is “plugged” into unsaturated applications prior to type checking. The type structure Ω(T) contains all types properly constructed from the type constructors in T (see below). In contrast, the temporal type structure Ω(T) ⊆ Ω(T) contains only temporal types. These are constructed from Time using nonatomic type constructors in T, e.g., [Time], [Time] ∈ Ω(T). The type constructor set T also contains document type constructors, resembling document templates in the repository. A subtype theory S contains subtype axioms about record and variant types, respectively. Record type constructor definitions R induce a record constructor K and a label environment Π containing record labels and their types. For example, the label environment Π contains kDefs : Time × ResD → [Item], dName : Time × ResD → String.

There exist decidable subsets in temporal predicate logic [11]. A restriction to these subsets, however, would severely limit the range of applications using our approach.

Figure 5: Formal consistency rule syntax

Example 3. For the rules formalized the language designer defines symbols and types like those shown in Fig. 4, where < g denotes an explicit subtype relation (called subtype axiom) between two types. Partially temporal symbols that do not really depend on time are marked with *. They are subject to unsaturated application.

The variant list type [a] is declared as usual in functional programming – the variant constructors are [] (empty list) and (:) (cons). The record type Doc stands for a formal document, holding a name (of type String) and a check-in time.

concatMap :: (a -> b) -> (Time x [a] -> [b]) = Haskell concatMap

Function symbols in F

concatMap : ∀α, β. Time x (Time x α) → β

Figure 4: Example types and symbols

Type definitions in T

String [a] = {[], (:) x [a]} lists

Doc = Doc [dName :: String, dTime :: Time] documents

ManD <a  = Man [kind :: String] manuals

ResD <a  = Res [KDefs :: kDefs] key resolvers

KDefs = KDef [key, kId, kKind :: String] key definition

Predicate symbol definitions in P

e = ∀α. Time x α x α equality

≤ = Time x Time timestamp ordering

Function symbol definitions in F

concatMap : ∀α, β. Time x (Time x α) → β

Haskell concatMap

repDs : Time → [Doc] get documents

repManDs : Time → [ManD] get manuals

repResDs : Time → [ResD] get resolvers

refS* : Time x Doc → [String] referenced keys

repStates : [Time] all repository states
The system supports first-order logic properties: we do not quantify over functions. In contrast to most functional programming languages, we restrict ourselves to full application of a type constructor to argument types and full application of functions and predicates to argument terms. To simplify notation, we use uncurried syntax in function and predicate types. Furthermore, we distinct between ground types of functions and predicates to argument terms. To simplify the type system, we can easily model document subsets and read: “In the subtype theory $S$, under subtype constraints $C$, variable assumptions $\Gamma$, and symbol assumptions $\Delta$ the term $e$ has the type scheme $\sigma$.” Rule judgments for formulae $C$, $\Gamma$, $\Delta \vdash s \phi$ ensure that $\phi$ is well-typed. The context $\Delta$ holds the types of all symbols in a signature $\Sigma$, and $\Gamma$ holds the types for variables introduced by quantifiers.

The well-typedness rule TypSymApp models valid function symbol application. If a symbol $s$ has the function type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_0$ and each term $e_i$ ($i \in \{1, \ldots n\}$) has the type $\tau_i$ then we can apply $s$ to $e_1$ through $e_n$ and $s(e_1, \ldots, e_n)$ has type $\tau_0$.

The well-typedness rule TypSub covers subsumption due to subtyping. If a term $e$ has type $\tau$ and $\tau$ is a subtype of $\tau'$ (inferred through subtyping rules) then $e$ also has type $\tau'$.

SubConstr \hspace{1cm} \text{SubTrans}
\begin{align*}
\text{TypSymApp} & \hspace{1cm} C, \Gamma, \Delta \vdash s : \tau_1 \times \ldots \times \tau_n \rightarrow \tau_0 \quad C, \Gamma, \Delta \vdash s_1 : \tau_1 \\
& \hspace{1cm} C, \Gamma, \Delta \vdash s(s_1, \ldots, s_n) : \tau_0
\end{align*}

We have seen that rule design is a complex task. Higher-order functions can simplify this task. But if used without caution they can result in subtle evaluation errors. Thus we provide the rule designer with a type checker that helps him to decide whether the rules formalized can make sense.
Well-typedness rules for formulae are straightforward. The rule TypQuant corresponds to TypLet in [16] except that the term e, defining the domain for x, must have a list type. Since we understand e as a domain, e must have some kind of “container” type.9 If e has the type [r] and the formula φ is well-typed under Γ, extended by the new variable assumption x:τ, then the quantified formula Q x ∈ e. φ is well-typed. Without loss of generality, we allow only rectified formulae, in which each quantifier binds a different variable. Thus Γ cannot contain an assumption for x.

\[ \text{TypQuant} \quad C, \Gamma, \Delta \vdash S : [r] \quad C \cup \{x: \tau\}, \Delta \vdash S \quad \phi \]

**Example 4.** In rule φ1 we would like to derive the type String for the term dName. First, our preprocessor extends dName(m) to dName(*, m). To infer dName(*, m): τ0 the rule TypSymApp requires: dName: τ1 × τ2 → τ0, *: τ1, and m: τ2. From our example signature Δ derives *; Time and dName: Time × Doc → String. Quantifiers push the types of their bound variables into Γ such that m: ManD ∈ Γ. Due to subtyping we also obtain m: Doc (TypSub). Finally, we derive the typing dName(*, m): String. □

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 [16] 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 (see, e.g., [27,19]). Whereas complete subtype inference is NP-hard [12], 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. Furthermore, we use explicit record constructors and an annotated case construct, providing further guidance. For brevity we omit a detailed description of our type inference algorithm and refer the reader to [23].

A lot of background support is provided by our tools to ensure that only well-typed rules are evaluated. We consider type checking a key ingredient in our framework because syntactical well-typedness of consistency rules is a vital precondition for their evaluation, which we discuss next.

5. **EVALUATING CONSISTENCY RULES**

In this section we show a semantics that allows to pinpoint the trouble spots making a repository inconsistent w.r.t. a rule system. We do not restrict ourselves to finding out whether a repository fails to satisfy certain rules. Instead, we want to know when, where, and why documents in the repository are inconsistent.

Classic set theoretic semantics provides a boolean result only and is, therefore, not sufficient for our purposes. Evaluation of a predicate logic formula, however, assigns concrete values to quantified variables. The basic idea behind our tolerant semantics is to exploit these assignments to indicate inconsistencies. We have designed our evaluation algorithm to provide compact information that precisely characterize inconsistencies. Our tolerant semantics follows xlinkit [15], which calculates a set of links, each containing a consistency

9We use lists instead of sets because lists are more familiar to Haskell programmers – our language designers.

flag and a value set that makes the formula true and false, respectively. A major difference is that we also return those atomic formulae that make a rule true or false. Authors need a more detailed report when consistency rules grow large and only some of many predicates falsify a rule.

5.1 **Overview**

For each consistency rule we generate a consistency report containing a boolean result (representing boolean truth semantics) and a diagnosis set. A diagnosis (C, as, ps, ps) reads: “The processed formula is satisfied (Consistent) for variable assignments as due to true atomic formulae ps and false atomic formulae ps.” A diagnosis (IC, as, ps, ps) indicates that the processed formula is violated (InConsistent). The assignment as maps variables to concrete values, e.g., timestamps or documents. Due to temporal quantification the variable assignment as tells when and where a rule is violated. The sets ps and ps describe why a rule is violated.

**Example 5.** For brevity we omit Haskell sources of function and predicate symbol implementations. Fig. 7 shows a small repository containing a documentation doc1.txt, a key resolver keys.xml, and a manual man1.xml the kind of which changed during the transition from state 1 to state 2. This introduces two inconsistencies: State 2 is inconsistent w.r.t. rule φ1 because the link kaA1c3 is invalid at state 2. States 1 and 2 are inconsistent w.r.t. rule φ2 because the kind of man1.xml has changed.

The automatically generated reports shown in Fig. 8 reflect these inconsistencies. The report for rule φ1 lacks assignments to d and m – the repository is inconsistent w.r.t. φ1 for all possible assignments to d and m, respectively. The report lacks the atomic formulae k = key(d) and dName(m) = kind(d) – the key resolver contains kaA1c3, 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. The report for rule φ2.
shows that for the manual man1.xml at state 1 no manual at state 2 could be found with the same kind. The report lacks @name(m1) = @name(m2) because there exists a manual man1.xml at state 2 – only its kind is wrong.

We can react in various plausible ways to the above reports; e.g., we might change the key resolver to point at a field manual or inform the author of man1.xml about the inconsistencies. □

Next, we present our algorithm that generates consistency reports.

5.2 Generating Reports Automatically

As usual, we interpret temporal formulae in first-order temporal structures \( A = (\mathcal{T}, I) \), consisting of (1) a time-line TL interpreting temporal types, fully temporal predicate symbols, and fully temporal function symbols; and (2) a function \( I \) associating a time \( w \) with a non-temporal structure \( A = I(w) \) interpreting non-temporal types, partially temporal predicate symbols, and partially temporal function symbols. Neglecting their temporal parameter, record labels and variant constructors have the usual built-in semantics: A record label selects a field from a record, a variant constructor builds a variant. The domain of Time is the set of natural numbers \( \mathbb{N} \), representing repository states. We interpret \( \leq \) via the natural “less equal” ordering. Formally, a consistency report is defined as follows:

Definition 2. A consistency report \( (b, D) \) contains a boolean value \( b \) (representing boolean truth semantics) and a set of diagnoses \( \mathbb{D} \). A diagnosis \( (c, a, ps, p_\phi) \in \mathbb{D} \) contains a consistency flag \( c \in \{\mathcal{C}, \mathcal{I}\} \), a variable assignment \( a \), and sets of atomic formulae \( ps \) and \( p_\phi \).

The function \( R_\mathcal{A}[\phi] \eta \) generates the consistency report for a formula \( \phi \) w.r.t. the temporal structure \( A \) and the variable assignment \( \eta \) – see Fig. 9. For auxiliary functions used by \( R \) see [23]. Initially, \( R \) is applied to an empty variable assignment. Depending on their truth value we push atomic formulae into \( ps \) and \( p_\phi \), respectively. We store the complete application to identify predicate symbols that occur more than once in a formula. For a conjunction \( \phi \land \psi \) we retain the report of subformulae that are responsible for the final truth value – this shortens our reports. If \( \phi \) and \( \psi \) have the same truth value we compute the cartesian product of their reports via the auxiliary function \( \varpi \). Otherwise, we retain the report of the false subformula, because the false subformula is already sufficient for making the conjunction false. For disjunctions we use a similar approach but join reports via \( \odot \).

For a formula \( \forall x \in e \bullet \phi \) we first evaluate the domain term \( e \) giving a list of values. For each value \( v \) in this list we compute \( \phi \)'s reports w.r.t. the assignment extension \( \eta[x \mapsto v] \). If \( \phi \) is violated for an assignment extension then the boolean value of the corresponding report is false. In this case we push the current variable assignment \( x \mapsto v \) into the variable assignments of each diagnosis in \( \phi \)'s report. Finally, \( \bar{\varpi} \) joins the resulting reports in \( F \). If \( F \) is empty then \( \phi \) is satisfied for any assignment extension. So we join the reports in \( T \), containing the reports of satisfied \( \phi \) without \( x \mapsto v \). The new variable assignment is superfluous here because \( \phi \) is true for every \( \eta[x \mapsto v] \). In a consistency report, we understand omitted variable assignments as universally quantified. For existentially quantified formulae we use a similar approach. The roles of \( T \) and \( F \) are, however, “reversed” because an existentially quantified formula is already satisfied if its child formula is true for one assignment extension.

Note that variable assignments inside diagnoses can only contain quantified variables. So we must be careful when replacing quantifiers by applications of \texttt{concatMap}.

The evaluation function \( V_A[e] \eta \) returns the value of a term \( e \) w.r.t. a temporal structure \( A \) and a variable assignment \( \eta \). We calculate values as usual in two-sorted temporal logics; for brevity we omit a formal definition. Applications of partially temporal function symbols \( f(e_0, e_1, \ldots, e_n) \) first evaluate the temporal parameter \( e_0 \), select its associated non-temporal structure \( \tilde{A} \), and apply the interpretation of \( f \) in \( \tilde{A} \) to the evaluations of \( e_i \) through \( e_n \).

We demonstrate the algorithm above by showing part of the report generation for rule \( \phi_1 \).

\[
R_{\mathcal{T}, I}[p(e_0, e_1, \ldots, e_n)] \eta = \begin{cases} (\text{True}, \{\mathcal{C}, \emptyset, \{p(e_0, e_1, \ldots, e_n), \emptyset\}\}) & \text{if } (V_{\mathcal{T}, I}[e_0] \eta, \ldots, V_{\mathcal{T}, I}[e_n] \eta) \in p^A \\ (\text{False}, \{\mathcal{I}, \emptyset, \{p(e_0, e_1, \ldots, e_n)\}\}) & \text{otherwise} \end{cases}
\]

where \( \tilde{A} = I(V_{\mathcal{T}, I}[e_0] \eta) \)

\[
R_{\mathcal{T}, I}[p(e_1, \ldots, e_n)] \eta = \begin{cases} (\text{True}, \{\mathcal{C}, \emptyset, \{p(e_1, \ldots, e_n), \emptyset\}\}) & \text{if } (V_{\mathcal{T}, I}[e_1] \eta, \ldots, V_{\mathcal{T}, I}[e_n] \eta) \in p^A \\ (\text{False}, \{\mathcal{I}, \emptyset, \{p(e_1, \ldots, e_n)\}\}) & \text{otherwise} \end{cases}
\]

where \( \tilde{A} = I(V_{\mathcal{T}, I}[e_1] \eta) \)

\[
R_{\mathcal{A}[\phi \land \psi]} \eta = R_{\mathcal{A}[\phi]} \eta \varpi R_{\mathcal{A}[\psi]} \eta
\]

\[
R_{\mathcal{A}[\phi \lor \psi]} \eta = R_{\mathcal{A}[\phi]} \eta \odot R_{\mathcal{A}[\psi]} \eta
\]

\[
R_{\mathcal{A}[\neg \phi]} \eta = \bar{\varpi} R_{\mathcal{A}[\phi]} \eta
\]

\[
R_{\mathcal{A}[\phi \Rightarrow \psi]} \eta = \begin{cases} R_{\mathcal{A}[\neg \psi]} \eta & \text{if } T \neq \emptyset \\ (\text{True}, \emptyset) & \text{otherwise} \end{cases}
\]

\[
R_{\mathcal{A}[\forall x \in e \bullet \phi]} \eta = \begin{cases} \bar{\varpi}(F) & \text{if } F \neq \emptyset \\ \varpi(T) & \text{if } T \neq \emptyset \end{cases}
\]

\[
R_{\mathcal{A}[\exists x \in e \bullet \phi]} \eta = \begin{cases} \varpi(T) & \text{if } T \neq \emptyset \\ \bar{\varpi}(F) & \text{if } F \neq \emptyset \end{cases}
\]

\[
F = \begin{cases} \{\text{push}(x \mapsto v, R_{\mathcal{A}[\phi]}[\eta[x \mapsto v]]) \mid v \in V_A[e] \eta \text{ and } \text{fst}(R_{\mathcal{A}[\phi]}[\eta[x \mapsto v]]) = \text{False} \} & \text{otherwise} \\ \{\text{push}(x \mapsto v, R_{\mathcal{A}[\phi]}[\eta[x \mapsto v]]) \mid v \in V_A[e] \eta \text{ and } \text{fst}(R_{\mathcal{A}[\phi]}[\eta[x \mapsto v]]) = \text{True} \} & \text{if } T \neq \emptyset \\ \{\text{push}(x \mapsto v, R_{\mathcal{A}[\phi]}[\eta[x \mapsto v]]) \mid v \in V_A[e] \eta \text{ and } \text{fst}(R_{\mathcal{A}[\phi]}[\eta[x \mapsto v]]) = \text{False} \} & \text{if } F \neq \emptyset \end{cases}
\]

\[
R_{\mathcal{T}, I}[p(e_0, e_1, \ldots, e_n)] \eta
\]

Figure 9: Rule evaluation algorithm.
Example 6. In the rule $\phi_3$ the topmost quantifier $\forall t \in \text{repStates} \cdot$ first evaluates \text{repStates} to $[1, 2]$ and calls $\mathcal{R}$ with two variable assignments, $\mathcal{R}_A[\{0\}[t \mapsto 1] \land \mathcal{R}_A[\{0\}[t \mapsto 2]$. The former results in a true report $(True, 0)$. The latter reports the following inconsistency:

\[
\begin{pmatrix}
\text{False}, \{x \mapsto \{dName = doc1.txt, dTime = 1\}, \{k \mapsto kaA1c3\}, \{kind(m) = kKind(d)\}\}
\end{pmatrix}
\]

We push the assignment $t \mapsto 2$ into the above report via $\text{push}(t \mapsto 2, \mathcal{R}_A[\{0\}[t \mapsto 2])$. Since $F$ contains exactly one report $\exists(F)$ results in the final report, shown in Fig. 8.

Speed is a key issue when generating reports: Number and size of documents, and the number of repository states contribute a polynomial factor; the nesting of quantifiers an even exponential behavior. We, therefore, evaluate consistency rules incrementally; for details see the companion paper [24].

6. IMPLEMENTATION

This section sketches our implementation and presents some lessons learned from our experiments.

6.1 Prototype Implementation

How do we perform a consistency check in practice? Our consistency checker is implemented in Haskell. Currently, we interface the revision control system darcs [21]. Assume an author checks in a document. Then, darcs first performs some tests to ensure that the changes submitted may be applied to the repository. If this test succeeds darcs calls our consistency checker via a system call. We use a system call because of its simplicity. During the run of our consistency checker the repository is locked, i.e., no check-ins can be performed. In case all strict rules are satisfied our consistency checker finishes normally and darcs 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. The next step is to type-check the rules against the functions and predicates they use. Usually, the language used is not changed between consistency checks. In this case we dynamically link auxiliary Haskell modules to the running consistency checker (the modules have been compiled during a previous consistency check). These Haskell modules contain implementations of functions, predicates, and types. Haskell lacks subtyping. We, therefore, resolve subtype relationships through (1) coercion functions, which coerce a subtype to its supertype, and (2) marshalling functions, which convert a Haskell value into a value as required by our consistency checker. Now, we can call the functions supplied by the language designer in order to check the repository for consistency. Consistency checking uses the algorithm from Sect. 5. If, however, the language used has changed since the last consistency check we generate the above Haskell modules and compile them by the Haskell compiler GHC.

The GHC also type-checks functions and predicates implemented by the language designer. All this is performed in the background.

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 repository type, e.g., darcs or CVS. The interface consists of five Haskell functions: \text{repStates} returns all states of a given repository. \text{repHeadState} returns the repository head, i.e., the current state. \text{repDocs} returns the documents in the repository that are current at a given time and match a regular expression. For example, \text{repDocs repo 2 ".xml"} returns all XML documents current at state 2. \text{repFolders} behaves similarly to \text{repDocs} but returns directories instead. \text{parseDoc} accesses a given document in the repository. One parameter is a function that parses the document content and converts it into an appropriate Haskell data structure. For XML documents these parser functions can be generated. The interface functions above can be used by the language designer.

6.2 Lessons Learned

In order to test the feasibility and the practical relevance of our approach, we currently run a project together with sdm

\[\text{see www.sdm.de}\]

\[\text{\\\\\text{10}}\]
Programming language environments [20] evaluate semantic rules of the underlying programming language on abstract syntax trees of source code documents. Later work on software engineering environments [28] provides consistency checks across different documents. Rules are, however, limited to a subset of a (non-temporal) first-order logic. Various database systems [7] employ integrity constraints. Our approach shares some ideas with Thémis [2]: Higher-order complexity is encapsulated in functions and thus hidden from first-order rules. Recent works on semistructured databases [3] and integrity constraints for web sites [9] 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 allow violations of database integrity constraints. The database “corset” is, however, too strict and limited to document metadata only. Complex rules, e.g., “referenced sections should keep similar over time,” require to inspect document content via information retrieval techniques [32]. Metadata impinging consistency may change, which would require to adapt the database schema. For smaller projects the database approach appears too heavy weight. Of course, our database-independent approach can still use databases for fast metadata access.

In software engineering [10, 17] tolerating inconsistency is considered more profitable than enforcing consistency. Many-valued logics help to model inconsistencies and their consequences [8]. We argue, however, that classic logic is easier to understand for users and that our approach is powerful enough to manage consistency. Our priority levels for rules can be considered as 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 [30], in software engineering [10, 17], and for analyzing consistency between documents [15]. While for knowledge bases decidability of the implication problem is crucial, we find close relationships in the context of distributed documents. xlinkit [15] statically checks distributed documents against user-defined consistency rules and implements tolerant consistency. Rules are formalized in an untyped non-temporal first-order logic employing user-defined predicates the semantics of which is implemented in a Javascript “blackbox.” This hinders the reuse of already available algorithms. We share many ideas with xlinkit but use a repository, which has a lot of advantages: (1) DMS and repositories are widely used and provide useful management mechanisms, e.g., document locking; (2) Internet access to a repository already supports distribution; and (3) history information (already stored by DMS) is a precondition for temporal consistency rules and incremental evaluation. Furthermore, 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 to provide consistency management.

As far as document types are concerned, we might also have used finite tree automata [29]. 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.

8. CONCLUSION AND OUTLOOK

We have presented an approach to flexible consistency management in heterogeneous repositories by explicit formal consistency rules. Formalization provides a common understanding of consistency, which is vital for any collaborative document engineering process. We have sketched how our approach can be integrated into existing DMS. A rule designer defines domain specific rules in a full first-order temporal logic with linear time. In contrast to decidable subsets, full temporal first-order logic provides an expressive rule designer toolset and supports formalizing practically relevant consistency rules. Higher-order complexity is hidden from the rule designer; it is encapsulated in functions and predicates defined by the language designer. Our type system significantly supports the formalization process. In the presence of subtyping records and variants provide expressiveness to describe document structures, comparable to the XML Schema standard. Automatically generated consistency reports precisely pinpoint inconsistencies within documents w.r.t. the rules defined. This allows for flexible inconsistency handling strategies. We have implemented our consistency checker interfacing the revision control system darcs [21].

Because of space restrictions we omit speed issues (see [24]) and our concrete rule syntax, which is XML-based. Currently, a graphical user interface is developed (see Fig. 10) that greatly simplifies formalizing and maintaining consistency rules, and is a key ingredient for ensuring user acceptance. The correctness of rules covers another aspect, not yet dealt with in this paper: Do the rules formalized reflect the rule designer’s intentions? For this purpose we plan to re-translate formal consistency rules into natural language. The translation can be compared to the initial intentions. Finally, just pinpointing inconsistencies is only a first step towards flexible consistency management. We are developing strategies to suggest compact and reasonable repair actions, in order to resolve inconsistencies. Repair actions will be generated from enriched consistency reports.

We are confident that the contributions of this paper can significantly improve consistency management in document engineering processes. Early tests in the field of software engineering confirm this assumption.
Acknowledgements

We would like to thank Christiane Stutz from sd&m for many fruitful discussions and practical examples, Wolfram Kahl for valuable suggestions concerning the formal basics, and Tobias Uwe Kuhn for developing the GUI. Special thanks go to the anonymous reviewers of previous drafts of this paper.

9. REFERENCES


