Pragmatic Consistency Management in Industrial Requirements Specifications

Jan Scheffczyk, Uwe M. Borghoff
Universität der Bundeswehr München
{jan.scheffczyk,uwe.borghoff}@unibw.de

Andreas Birk, Johannes Siedersleben
sd&m AG
{andreas.birk,johannes.siedersleben}@sdm.de

Abstract

Industrial requirements specifications suffer from consistency problems, particularly in multi-angular or temporal relationships between different specification results. Current consistency management tools generate too many repairs, lack support for temporal relationships, and are poorly integrated into development processes. In this paper we evaluate our consistency management method on how it improves quality of industrial specifications: We formalize (temporal) consistency rules and generate a few domain-specific repairs for inconsistencies. We demonstrate our method using an example specification. Since the effort for formalization is tunable to specific applications and our prototype shows satisfactory performance, we are confident that our contributions scale to an industrial setting.

1. Introduction

Well-structured and semantically consistent requirements specifications are crucial for the success of software projects. Various studies have shown that quality of requirements engineering work products and ability to keep work products up to date are important factors for the success of software projects [11, 10, 7, 6].

Usually, consistency of requirements specifications must be ensured using reviews and inspections. This consumes considerable amounts of time and effort, is a complex and potentially error-prone activity, and tends to be neglected during project planning and resources allocation.

sd&m is a software house specialized in Central Europe that develops custom software solutions for large industrial enterprises. It is a company of Capgemini with 950 staff (2004) and a turnover of € 125 million (2004). Projects range up to 80 persons team size and can have several years of duration. Software specifications usually consist of a set of documents of several hundreds of pages total volume. They may include hundreds of business processes, use cases, data models, dialog designs, and various other parts.

sd&m has established so-called Specification Modules [22]. Specification modules are a collection of tailorable document templates with associated recommended good practice. They have grown out of past project experience and facilitate efficient requirements analysis and specification. They also help ensuring high quality of work products. More than hundred sd&m projects have built their specification documents in accordance with the specification modules. We have chosen specification modules as the basis of our consistency management method. This gives us the opportunity to reuse analysis infrastructure across projects.

Inconsistencies between specification results are natural in a multi-author development process [15, 21]. Ignoring them, however, severely impacts the quality of the specification and hence the quality of the developed system. Therefore, it is necessary to be aware of inconsistencies, their impacts, and the costs of possible repairs.

The Consistent Document Engineering Toolkit\(^1\) (CDET) is a general-purpose semantic consistency management approach [20, 19] that (1) provides automated consistency checking on heterogeneous documents, (2) tolerates inconsistencies, (3) suggests prioritized domain-specific repairs, and (4) gives authors full control of scheduling the repair actions. CDET can check semantic consistency at various granularity levels and integrates fully with authors’ established practices and their day-to-day project work. Documents are controlled by an arbitrary revision control system (RCS),\(^2\) which supports process integration, efficient consistency checking, and temporal consistency rules (which restrict the development of documents over time). That way, we integrate flexible consistency management into the day-to-day work of requirements engineers without requiring any adaptations to the practices used before.

In this paper we show how semi-formal consistency management by CDET can improve the quality of industrial requirements specifications. In particular, we demonstrate the usefulness of domain-specific repairs. First, we discuss related work in Sect. 2. Sect. 3 presents prereq-

\(^1\)see www2-data.informatik.unibw-muenchen.de/cde.html

\(^2\)Throughout, we use the abbreviation RCS as a general term, not to be confused with the specific tool that has the same name.
uisites for pragmatic consistency management in industrial settings. We introduce specification modules as a means for structuring specifications in Sect. 4 and informal consistency requirements between specification results in Sect. 5. Sect. 6 includes a summary of CDET; for technical details see [20, 19]. We define formal consistency rules in Sect. 7. Sect. 8 demonstrates consistency management using an example specification, which includes typical inconsistencies. In Sect. 9 we evaluate our method and give advice to software engineers about how they can benefit from our method.

2. Related Work

Due to space restrictions, we compare our method to consistency management approaches in software engineering only. For discussion of other related work see [20, 19].

CASE tools significantly aid software development. Usually, these tools check a fixed set of consistency rules, e.g., referential integrity constraints or naming conventions, see, e.g., [3]. Even state-of-the-art CASE tools are limited to the documents they maintain, e.g., UML models [12]. It is hardly imaginable that all results of a specification can be maintained by one CASE tool. In contrast, CDET supports documents of heterogeneous content and structure.

Naturally, formal specification approaches [24, 17, 2] provide powerful means for consistency management. As yet, these approaches require enormous effort for formalization, which often means to implement a new software development process and thus hinders application to industrial specifications. Even recent advances towards executable specifications [9] cannot eliminate these efforts. Therefore, formal approaches are limited to critical parts of software systems. In contrast, CDET does not require a formal specification. Without major adaptations, our method can be integrated into arbitrary development processes.

In UML [4], constraints can be defined using the Object Constraint Language (OCL) [23]. CDET’s consistency rule syntax shares some similarities with OCL invariants. Thus, although large part of a specification cannot be defined in UML [22] and often OCL appears too implementation orientated, OCL syntax could be used to formalize consistency rules. We regard, however, syntax issues a matter of personal preference (OCL invariants can be translated to CDET’s syntax).

Probably, the consistency management tool xlinkit [14] is closest to CDET. Xlinkit derives complete repairs for distributed documents and is, therefore, well suited for distributed authoring, e.g., for developing components [13, 8]. Due to the lack of version control, xlinkit cannot check temporal consistency rules. Also, interaction of repairs for different rules and compatibility of repairs w.r.t. the document structure are neglected. We focus on consistency management for development processes and collaborative work in teams. Therefore, we integrate consistency checking into RCS and support temporal rules. Major contributions of CDET are repair reduction and support for domain-specific repairs. As implementation is concerned, xlinkit is a mature product, whereas CDET is a prototypical implementation that explores new research directions in semi-formal consistency management and guides projects towards evolving their existing infrastructures.

3. Pragmatic Consistency Management

To our experience, consistency management in industrial settings must adhere to several requirements in order to work effectively and become accepted by requirements management and software development teams. The requirements below may appear trivial at first sight; we found, however, that many of them are neglected by consistency management approaches.

Most importantly, consistency management must help requirements engineers. It must integrate well with established work processes of software development teams. The consistency management approach must be designed so that it imposes minimum overhead effort on requirements engineers. Requirements engineers must be able to select applicable consistency rules on their own.

The consistency management approach must scale for large collections of high-volume documents. Consistency analysis must work at various granularity levels with informal, semi-structured, and heterogeneous work products.

The consistency management approach must allow requirements engineers to schedule repair actions in accordance with their overall task assignments. Repair actions must be easily comprehensible to requirements engineers and as specific as possible. For flexible inconsistency resolution, domain knowledge should be incorporated into consistency management. Also, resolution strategies may change as the documents evolve over time.

4. Specification by Specification Modules

At sd&m 17 specification modules [22] have been identified for structuring industrial specifications. Here we concentrate on the functional requirements of a specification, namely business processes, use cases, and dialogs. For further details see [22].

Usually, industrial specifications are partitioned into multiple documents, each of which contains specific specification results, e.g., use cases or business processes. Specification modules are templates for these documents. A specification consists of a specification document, which references the result documents. Multiple specification documents support to distinguish different reader species, e.g.,
customers or the test team. All documents are stored in a repository, managed by an RCS (like CVS or subversion [5]), in order to coordinate collaborative work of multiple requirements engineers. The state of the repository corresponds to the development state of the specification.

Specification modules include the usual means for capturing functional requirements, e.g., business processes, use cases, and dialogs. A business process describes central activities in the customer business. The specified system supports some activities of a business process, which are described by use cases in detail. A use case describes functional requirements of the specified system from the user perspective. For a use case, pre- and postconditions may be listed. Dialogs specify the look and feel of the system; they are associated to use cases. In use cases and dialogs, complex processes and computations without user interaction can be described by analysis functions. These functions are called by use cases and dialogs, respectively. The description of an analysis function contains pre- and post-conditions, effects, and messages. Fig. 4 illustrates the relationships between use cases, dialogs, and analysis functions. Large software systems are partitioned into technical components; they provide a view orthogonal to the specification modules. Components, which appear as sections in documents, apply to use cases and dialogs.

Specifications are directed to multiple reader species. Hence, for each reader species we have a specification document, which references documents intended for this reader species. This supports to reuse specification results on the one hand and to hide results on the other hand. For example, customers are not bored with analysis functions, which are, however, needed by the test team. For a specification, we distinguish between study, coarse specification, and fine specification; technically, these are special repository states.

5. Informal Consistency Requirements

Specification modules induce a range of consistency requirements at different granularity levels (we have identified more than 100). Simple requirements just claim the existence of some specification result (part) if some other result is present. More complex requirements involve sophisticated content analysis, like satisfiability of preconditions (see below). Temporal requirements relate specification results at different development states. The following requirement kinds are of major importance for the functional requirements of a specification. For each kind, we present an example consistency requirement including its strength and importance. Strong requirements must be adhered to, weak requirements may be violated.

Satisfiability of preconditions of use cases and analysis functions. We express pre- and postconditions by boolean logic formulae.

R 1 The preconditions of a use case should be implied by the postconditions of all other use cases.

strength: weak  importance: medium

Validity of triangular relationships between use cases, dialogs, and analysis functions.

R 2 An analysis function called by a dialog $dlg$ should also be called by each use case associated with $dlg$.

strength: weak  importance: high

R 3 A dialog should be associated to use cases from its own component only.

strength: weak  importance: medium

Requirements for specifications directed to specific reader species. R 4 permits lazy inconsistency resolution during specification development. For the final specification, however, we make R 4 strong, which prohibits inconsistent specifications:

R 4 A specification intended for a reader species should contain documents for this reader species only.

strength: weak  importance: high

R 5 A finished specification intended for a reader species must contain documents for this reader species only.

strength: strong  importance: high

Temporal requirements relate specification results at different development states. R 6 ensures that in the fine specification no activities get lost. Sometimes, however, activities are dropped intentionally. Our consistency management method documents these design decisions automatically.

R 6 A system-supported activity in a business process of the coarse specification is also included in the fine specification and is also supported by the system.

strength: weak  importance: high
6. Consistency Management by CDET

Fig. 2 illustrates the CDET consistency management approach. In their day-to-day work, requirements engineers check out documents containing specification results, edit them, and check them in again. Consistency requirements are formalized via consistency rules in a variant of full first-order logic. Rules can be customized for specific projects. In order to ensure well-formedness, rules are type-checked against the functions and predicates they use. Types also include document types, corresponding to specification modules. Document types abstract from the actual document format, which may change without affecting the rules.

At a check-in, CDET checks the repository for consistency and generates repairs, from which requirements engineers can choose. Recall that we tolerate inconsistencies in the repository. Exponential complexity of repair enumeration techniques requires a two-step approach: (1) during consistency checking CDET generates for each rule an S-DAG (Suggestion carrying Directed Acyclic Graph); (2) on demand CDET derives one repair collection from all S-DAGs. The repository is locked during step (1) only, which is performed incrementally. An S-DAG visualizes inconsistencies and plausible repair actions for one rule. S-DAGs do not include all possible repairs but only a few repairs that require the least changes to the repository. S-DAG reduction is a major benefit in practice. Since the repair collection can grow large (hundreds of alternatives), it is sorted w.r.t. a user-defined preference metric.

7. Formal Consistency Rules

In order to check consistency across document revisions, CDET employs a variant of temporal predicate logic, which explicitly quantifies over repository states. That way, we can also formalize requirements for transitions between development phases in process models. Annotations guide repair generation by supplying domain knowledge. Formalizing consistency rules requires technical effort, which one should not underestimate: (1) document structures have to be well defined in order to retrieve document content, (2) document parsers have to be developed (for XML documents, parsers can be derived from DTDs), (3) consistency rules must be formalized, and (4) additional functions and predicates are necessary for content evaluation. To our experience, the advantages of our method outweigh these

---

3 Currently, the rules themselves are not checked for satisfiability, which is in general undecidable for full first-order predicate logic. We sacrifice satisfiability for expressiveness.

4 If XML is used, document types can be derived from DTDs.

5 A repository state represents the accumulated check-ins up to a given point in time.

6 In general, CDET can check any computable property of a document.
costs, which appear only once. Document authoring considerably benefits from clear document structures. Formal rules give precious insights and lead to fruitful discussions about what “consistency” actually means.

Fig. 3 shows a simplified excerpt of the document types for specification modules. Cardelli-style subtypes [1] have proven practical for document management. We model documents by the type Doc, which carries the document name and check-in state. All other document types are subtypes of Doc and prefixed by D_. A use case document D_Uc carries a list of use case components [Comp Uc]. The type D_Uc is a subtype of Doc (indicated by <); thus, D_Uc inherits the record labels id and state. Record labels serve as normal functions, which can be used in rules. For components, we define a recursive record type Comp α, parameterized over the type parameter α. A use case Uc carries references to analysis functions, and pre- and postconditions, given by boolean logic formulae. The variant type Logic is a supertype of the standard type Bool (indicated by >). Since for variant types inheritance is reversed, Logic inherits the standard boolean constructors True and False. Specification documents reference documents containing specification results, e.g., use case documents. Functions shown in Fig. 3 access documents at a given repository state and convert them into appropriate structures. Consistency rules work on these structures and are, therefore, independent from concrete document formats.

We now formalize the consistency requirements R 1, R 2, R 3, and R 6; R 4 and R 5 are omitted for brevity.

R 1 At all repository states t we require for all use case components c and all use cases u therein: The postconditions of all other use cases imply the preconditions of u.

\[ \forall t \in \text{repStates} \] 
\[ \forall c \in \text{contents(u)} \] 
\[ \text{map}(\text{post}(\text{filter}(\{\text{name}(u), \text{name}, \text{cMap}, \ldots\})) \Rightarrow \text{pre}(u) \]

R 2 At all repository states t we require for all dialogs dlg and all use cases u the following: If the dialog dlg is for u then all analysis functions fun called by dlg are also called by u.

\[ \forall t \in \text{repStates} \] 
\[ \forall \text{dlgCHG} \in \text{cMap(contents, cMap(flatten, cMap(digComp, repD_Dlg(t))))} \] 
\[ \forall u \in \text{cMap(contents, cMap(flatten, cMap(ucComp, repD_Uc(t))))} \]

0 \Rightarrow 
\[ \{\text{True: dlg.ucs \sim uucs(dlg) \& \text{name}(u)}\} \Rightarrow 
\[ \forall \text{fun} \in \text{Map(actFun, actions(dlg))} \] 
\[ \forall \text{fun} \in \text{funs}(u) \Rightarrow \{\text{False: u.funs \sim fun : (fun(u))} \}

In R 2 we annotate the atomic formula name(u) \in uucs(dlg) by a collection of hints, which guides repair generation by providing domain knowledge. Each hint set in this collection is an alternative, within a hint set all hints are evaluated simultaneously. A hint indicates how the truth value of an atomic formula can be inverted: b: x.f \sim e c means to change the label f of the variable x to the term e if the atomic formula evaluates to the boolean result b; this costs c. The hint True: dlg.ucs \sim uucs(dlg) \& \text{name}(u) \Rightarrow 3 removes from the list of all referenced use cases uucs in the dialog dlc the name of the use case u. CDET evaluates this hint if the atomic formula name(u) \in uucs(dlg) is fulfilled. The hint False: u.funs \sim fun : (fun(u)) 1 adds the analysis function fun to the list of referenced analysis functions in the use case u. In our experiments we have estimated the cost of a repair based on our experience about its real time effort and potential negative impacts. In addition, we can annotate a quantified variable x by KEEP (no repairs are generated for x) or by CHG (only repairs to change x are generated).

R 3 For all dialog documents dlgD, their components c, and all dialogs dlg therein we require that there exists a component cu containing use cases such that we have: The components c and cu have equal names and for all use cases referenced by dlg there exists a corresponding use case in the component cu.

\[ \forall t \in \text{repStates} \] 
\[ \forall \text{dlgCHG} \in \text{cMap(contents(c)}) \] 
\[ \exists \text{cu} \in \text{cMap(contents(c)}) \] 
\[ \text{name}(c) = \text{name}(cu) \] 
\[ \{\text{False: cu.name \sim name(c) 10}\} \] 
\[ \forall u \in \text{ucs(dlg)} \& \exists u \in \text{contents(c)}) \] 
\[ u\text{ref} = \text{name}(u) \Rightarrow \{\text{False: u.ref \sim name(u)} 1\}

\[ ^7\text{cMap}(f, xs) \) maps the function f over the list xs and concatenates the result lists, as concatMap in Haskell [16].

\[ ^8\text{filter}(ys, f, xs) \) retains all elements x of the list xs for which f(x) \notin ys holds. In contrast, filter(ys, f, xs) retains all elements x of xs for which f(x) \in ys holds.

\[ ^9\text{for non-record variables like u.ref in rule R 3 we omit the label f.}\]
If R 3 is violated we propose to rename the use case component \(c_i\) towards the name of the dialog component \(c\) and to change the name of the use case reference \(u_{ref}\) to the name of the use case \(u\), which is of course cheaper than changing the name of the component.

Hints are the basis for high-level domain-specific repairs: Repair strategies can change dynamically depending on the repository state and the repository documents. Hints provide direct access to the repository state at which an inconsistency occurred – thus, we can react flexibly to violations depending on the development state of a specification.

R 6 For every finished fine specification \(s_f\) we require for all corresponding finished coarse specifications \(s_c\), the following:

For each business process \(bp_c\) in \(s_c\) there exists a corresponding business process \(bp_f\) in \(s_f\) such that for each activity \(act_c\) in \(bp_c\) we have a corresponding activity in \(bp_f\) supported like \(act_c\).

\[
\forall \text{KEEP} \in \text{repStates} \bigwedge \forall \text{KEEP} \in \text{repStates} \bigwedge
\left[ \forall \text{KEEP} \in \text{repStates} \bigwedge \right.
\left. \forall \text{KEEP} \in \text{repStates} \right]
\]

\[
\left( \forall \text{KEEP} \in \text{repStates} \bigwedge \forall \text{KEEP} \in \text{repStates} \right)
\]

\[
\left( \forall \text{KEEP} \in \text{repStates} \bigwedge \forall \text{KEEP} \in \text{repStates} \right)
\]

8. Demonstration of Consistency Management

In this section we demonstrate the CDET consistency management approach using an example specification. Our full experiments can be found in [19].

8.1. Example Specification: The Ski School

The ski school is structured like a typical sd&m specification. For brevity, we only consider one coarse specification (repository state 13) and one fine specification (repository state 23); each including one specification document, two business processes, two use cases, and one dialog. In particular these are: the business process Register customer in the document BPRegCust.xml; the business process Assign teacher in the document BPAssign.xml; the use cases Register customer and Assign students in the document UC.xml; and the dialog Assign student in the document Dig.xml. Components are defined in the fine specification.

Fig. 4 outlines the business processes of the coarse specification. In the fine specification we drop wish list support.

All use cases belong to the component Course attendees. Pre- and postconditions are added in the fine specification.

Use case name: Assign students
Activities: Assign student (in Register customer)
Preconditions: course created and customer registered
Postconditions: student assigned to course
Analysis functions: Find course

Use case name: Register customer
Activities: Register customer, Register customer (2) (in Register customer)
Postconditions: customer registered
The dialog `Assign student` belongs to the component `Course planning` and is associated to the use case `Assign students`. The dialog calls the analysis function `Check maximum students`.

8.2. Visualizing Useful Repairs by S-DAGs

Inconsistencies in the above specification and repairs are visualized by S-DAGs. The structure of an S-DAG resembles that of a consistency rule. Nodes represent logical connectives or atomic formulae; edges target the subformulae of a connective. Universal nodes \( \forall \) and existential nodes \( \exists \) represent universal and existential quantification, respectively. Outgoing edges carry value bindings to the quantified variable and alternative repair actions. A value represents a repository state, a document, or document content, blamed for inconsistencies. An action proposes to either add a value to (Add), or change a value within (Chg), or delete a value from the domain of the quantifier node (Del). Conjunction nodes \( \land \) stand for conjunctions; disjunction nodes \( \lor \) stand for disjunctions.\(^ {10} \) A predicate leaf contains an atomic formula \( \phi \) that causes an inconsistency, the truth value of \( \phi \), and the collection of evaluated hints. This collection contains alternative sets, each including evaluated hints that indicate how the truth value of \( \phi \) can be inverted.

At state 13 only rule R 2 is violated (use cases lack pre- and postconditions, components are not yet introduced, and there is no fine specification). Fig. 5 shows the corresponding S-DAG. For brevity, we neglect previous repository states. The S-DAG shows two inconsistencies. The lower leaf indicates that the analysis function reference `Check maximum students` in the dialog `Assign student` is inconsistent because it is not referenced by the use case `Assign students`. This use case is, however, associated to the dialog `Assign student`, which we learn from the upper leaf. The bold edge below the disjunction node indicates that repairing the lower leaf is cheaper than repairing the upper leaf. The S-DAG proposes the following repair actions: (1) change the dialog `Assign student` such that it is no longer associated to the use case `Assign students`, (2) change the use case `Assign students` such that it also references the analysis function `Check maximum students`, (3) delete the analysis function reference to `Check maximum students` from the dialog. From an S-DAG, software engineers can choose suitable repair actions interactively. In our example, we add the analysis function reference to the use case `Assign students`, as proposed by action (2) above. CDET applies this action to the S-DAG, which lets the S-DAG collapse. This indicates that action (2) is sufficient for repairing inconsistencies for R 2.

The other rules are violated at state 23 (see Fig. 5).

The S-DAG for R 1 lacks concrete repair actions. The leaf proposes to change the use case `Assign student` such
that the postconditions of all other use cases imply its pre-
conditions course created and customer registered. This ac-
tion does not reflect what a software engineer would do in
this situation, namely to add a new use case ensuring the
postcondition course created. Usually, lack of hints in rules
causes lack of useful repairs.\(^{11}\)

The S-DAG for R 6 expresses repair actions that meet
our expectations: Add the missing activities or downgrade
the specification either to a coarse specification or to a
specification “in progress.” Notice that CDET proposes to
change activities in the business process Register customer
only. CDET does not propose to change the business pro-
cess Assign teacher. Below existential nodes and disjunc-
tion nodes, respectively, CDET selects cheap repairs that
require the least changes to the repository. That way, S-
DAGs are reduced. No matter how many business processes
the coarse specification contains, the S-DAG for R 6 will re-
main the same. In our example, dropping wish list support
is a deliberate design decision, which is documented by the
S-DAG.

8.3. One Repair Collection for All Rules

Clearly, the visualizing power of S-DAGs is limited to
a rather small number of inconsistencies. Also, in favor
of incremental generation of S-DAGs, interactions between re-
pairs are neglected; e.g., a repair can resolve inconsistencies
for multiple rules. Therefore, from all S-DAGs, CDET de-

erives one repair collection on demand. The repair collection
contains alternative repair sets; within each set, all repairs
together resolve all inconsistencies for all rules. CDET
guarantees that (1) each repair set is a real alternative that is
not expressed by other repair sets and (2) the repairs within
each set do not contradict each other. Repairs proposing to
change content are compatible to the document structure,
which we achieve by static type checking.\(^{12}\)

Fig. 6 shows the top-ranked repair sets for states 13 and
23, respectively. For the rules R 1, R 2, R 3, and R 6 at state
13 CDET derives three repair sets, each containing one re-
pair; at state 23 CDET derives three repair sets, containing
between three and four repairs. A repair consists of five
components: (1) affected rules and variables (a repair can
resolve inconsistencies for multiple rules), (2) the domain
term of the variables (as given by the rules), (3) variable
bindings necessary to calculate the domain of the repair,
(4) the proposed repair action, and (5) the repair rating.
A rating indicates how many inconsistencies are resolved,
which rules might be broken, and the repair cost. Naturally,

---

\(^{11}\)If we modelled pre- and postconditions by conjunctions only (e.g., by
a list of conditions), R 1 would have a “repair-friendlier” structure. Our
formalization of R 1 demonstrates the boundaries for repair generation.

\(^{12}\)Static type checking cannot ensure that insertions or deletions are
compatible to document structures. This requires dynamic type checking.

---

**Figure 6. Top-ranked repair sets**

repairs could violate consistency, which we can determine
only after a new consistency check.

The repair for state 13 applies to the variable \(u\) in rule
R 2. The domain is given by cMap(contents, . . . ()), where \(t\) is bound to the repository state 13. That means:
The use case \(u\) is part of a use case component at state 13.
The repair proposes to change the use case Assign students
such that it references the analysis functions Check maxi-
mum students and Find course. Applying the repair resolves
one inconsistency for a high priority rule and costs 1. The
rules R 1, R 2, and R 3 might be broken. The repair is
ranked top because our example preference metric prefers
changing document content to adding or deleting document
content.

At state 23 the first repair represents an inconsistency
due to missing use case postconditions. It is, however, un-
clear how this inconsistency should be repaired (indicated
by two question marks). The last two repairs propose to add
the missing activities to the business process Register cus-
tomer. Both repairs together are cheaper than downgrading
the specification.
9. Lessons Learned

In our full case study [19] we formalized 15 consistency rules and checked them against the specification outlined in Sect. 8.1. Only one rule was strong, 14 rules were weak. The specification contained 18 documents including (among others) four specification documents, three business processes, ten use cases, twelve analysis functions, and two dialogs. The specified system was subdivided into three components. There were 24 check-ins to the repository, defining 24 development states.

Overall, CDET reported 493 inconsistencies. Most of them resulted from “uncoordinated” check-ins of different requirements engineers, which is natural in multi-author environments. S-DAGs were generated in less than 14 seconds on a 800 MHz PIII laptop. In contrast, repair derivation took up to three minutes. Temporal rules are expensive to check: CDET needed up to six seconds to check the temporal rule R 6. Transitional rules, which relate documents at two consecutive repository states, can be checked almost as fast as static rules, which relate documents at one repository state only. Repairs appeared reasonable from the requirements engineering point of view, provided that rules were annotated by appropriate hints. At each state, we found a good repair set among the top five repair sets.

The cost of the CDET consistency management approach is influenced mostly by determining the actual consistency requirements informally and defining document structures. Clearly, formalizing consistency rules requires some technical effort. Rules are, however, formalized only once, which is subject to experts.

As yet, we have not studied large collections of high-volume documents. The properties of our method let us, however, expect good scalability. The ski school represents a typical sd&m specification. The effort for formalization mostly depends on finding out consistency requirements, which is influenced by the nature of the project but not by the number of documents. CDET makes no assumptions to particular document formats. All it requires is an RCS, for which the generic repository interface is instantiated; currently, CDET supports DARCS [18], subversion [5], and the normal file system. The repository is locked during S-DAG generation only. Repair derivation is done on demand. The times needed for S-DAG generation and repair derivation, respectively, confirm that CDET’s two-step approach is viable.

For applying pragmatic consistency management, we suggest the following steps:

1. Identify the document kinds that are part of the development process. What are their goals and scopes?
2. Explore informal consistency requirements within and between documents. Investigate document structures.

To our experience step (1) and (2) take long time; they give, however, precious insights into the consistency requirements actually needed, which leads to a good understanding of work flows beyond consistency management. Step (3) is an iterative process that includes technical details, which is subject to experts.

10. Conclusion and Outlook

In this paper we apply our CDET consistency management approach to industrial requirements specifications by using specification modules developed at sd&m. In order to support collaborative work, we identify useful consistency requirements, formalize (temporal) consistency rules, and integrate consistency management into the work with an RCS. In particular, we demonstrate that CDET can generate domain-specific repairs, provided that consistency rules are annotated by appropriate hints.

How does the CDET consistency management approach meet the requirements from Sect. 3? We do not hinder the work of requirements engineers – their day-to-day habits are not influenced because we tolerate inconsistencies. CDET is smoothly integrated into established work processes that use an RCS. Admittedly, formalization of consistency rules is subject to experts in the field of logic; the rules can, however, be reused in a number of projects. Requirements engineers can select and customize consistency rules. Rules work on heterogeneous documents and are independent from particular document formats. If the document format changes, all we need is a new parser. Since functions and predicates are implemented in a full programming language, it is possible to employ sophisticated information retrieval techniques and to check consistency at different granularity levels. CDET generates repairs in two steps, where the repository is locked during S-DAG generation only, which is done incrementally. So far CDET has shown satisfactory performance. Support from the used RCS is, however, poor since most RCSs ignore the document format (diffs are line based only). Obviously, we can improve performance for XML repositories by extending RCSs by XML revision control approaches, which we currently evaluate. Since CDET tolerates inconsistencies, requirements engineers can schedule repair actions with their overall task assignments. Hints are a powerful means for flexible reaction to inconsistencies. By hints we can incorporate domain knowledge in order to customize repairs. Since the repository state of a rule violation is known, hints can be used to change repair strategies dynamically according to the development state of a software project.

Based on the results of our experiments, we transfer our method into industrial projects along six lines of action:
(1) We reduce its formal overhead (e.g., formalization of hints, a systematic approach to assign costs, and definition of preference metrics). (2) We plan to improve usability of CDET’s output (e.g., converting repairs to natural language or enhancing the visual power of S-DAGs by using interactive graph viewing tools such as uDraw(Graph)).

(3) We extend our experiments towards architecture design, which includes temporal consistency requirements between requirements analysis and architecture design. (4) We have started applying our method in projects that already have high degrees of requirements formality (e.g., product data management). (5) We gradually augment project infrastructures to become suitable for automated document management and analysis (e.g., by improving document structures). (6) We feed back lessons learned on work product consistency to projects (e.g., by enhancing inspection checklists).

So we are confident that this pragmatic approach will evolve our understanding of practical consistency management and also improve quality of software engineering work products. Further information about our ongoing work can be found on the CDET WWW site www2-data.informatik.unibw-muenchen.de/cde.html.

References


13 See www.informatik.uni-bremen.de/uDrawGraph/en/home.html