erschienen in: ***Informatik: Forschung und Entwicklung*** **19(1):17–29, 2004**

Formale Konsistenzsicherung in informellen Software-Spezifikationen

Jan Scheffczyk[1], **Christiane Stutz**[2], **Uwe M. Borghoff**[1], **Johannes Siedersleben**[2]

[1] Institut für Softwaretechnologie, Universität der Bundeswehr München, Werner-Heisenberg-Weg 39, 85577 Neubiberg
ist.unibw-muenchen.de e-mail: {jan, borghoff}@informatik.unibw-muenchen.de
[2] sd&m AG, software design & management, Carl-Wery-Str. 42, 81739 München
www.sdm.de e-mail: {christiane.stutz, johannes.siedersleben}@sdm.de

The date of receipt and acceptance will be inserted by the editor

Zusammenfassung Die Spezifikation ist die Grundlage für den Erfolg eines Software-Projekts. Der Praktiker konzentriert sich hier i. d. R. auf die größte Herausforderung: die vollständige und inhaltlich korrekte Erfassung aller Anforderungen an das zu erstellende Software-System. Dass Spezifikationen für große Systeme aus vielen Dokumenten verschiedener Ausprägung in Form und Inhalt bestehen, tritt dabei meist in den Hintergrund. Ihre Konsistenz wird meist mit hohem manuellen Aufwand sichergestellt. Der Erstellung formal korrekter und konsistenter Software-Spezifikationen widmen sich zahlreiche theoretische Arbeiten. In der Praxis sind sie jedoch meist nicht mit der gewohnten Arbeitsweise vereinbar.

In diesem Artikel stellen wir einen Mittelweg vor: Wir nutzen die Spezifikationsbausteine von sd&m für die Spezifikation, die als Ergebnisse Dokumente in natürlicher Sprache sowie semi-formale Darstellungen umfasst. Für Spezifikationen, die nach diesen Bausteinen erstellt wurden, definieren wir Konsistenz durch formale zeitbehaftete Konsistenzregeln. Ein von uns entwickeltes Auswertungswerkzeug ermittelt Inkonsistenzen präzise. Unterstützt von einem solchen Werkzeug kann sich der Software-Ingenieur wieder ganz auf das Hauptanliegen der Spezifikation konzentrieren: ihre inhaltliche Korrektheit und Vollständigkeit.

Schlüsselwörter Software-Spezifikation, Spezifikationsbausteine, Konsistenz, Temporale Logik, Heterogene Repositories

Abstract Specification is the base for the success of a software project. Usually, practitioners concentrate on completely and correctly capturing all requirements for the specified system. In practice, a serious problem often fails to gain attention: Specifications for large systems consist of many documents with heterogeneous structure and content. Their consistency is often achieved by time-consuming manual effort. On the other hand, many theoretical approaches aim at formally correct and consistent software specifications. In practice, however, these approaches fail to be integrated into the every day work of software engineers.

This article is located at the boundary between theory and practice: For specifying software we use analysis modules developed at sd&m, the results of which are documents in natural language or semi-formal documents. For a specification using our analysis modules we define consistency via formal temporal consistency rules. Our newly developed tool pinpoints inconsistencies precisely. Supported by our tool, software designers can concentrate on their actual work: the technical correctness and completeness of a specification.

Keywords software specification, analysis modules, consistency, temporal logic heterogeneous repositories

CR Subject Classification D.2.1, D.3.1, H.3.1, I.7.1

1 Einführung & Motivation

Wie schreibt man eine Software-Spezifikation? Trotz aller Erfahrungen in der Durchführung von Software-Projekten steht diese Frage immer wieder am Anfang, denn stets stehen neue Aspekte im Vordergrund der Analyse.

Forschung und Praxis haben uns hier bereits viele gute Antworten gegeben: Zu den ältesten Analysemethoden gehört die Strukturierte Analyse [12], während seit den 1990ern die Objektorientierte Analyse (OOA) [6] und die Beschreibung von Use Cases [18] an Bedeutung gewinnen. Die Unified Modeling Language (UML) [7] vereint diese Ansätze und hat sich als Beschreibungssprache in der Spezifikation weithin etabliert.

Bei sd&m haben wir untersucht, welche Methoden in welchem Maße angewandt werden. Diese Ergebnisse wurden in den *Bausteinen zur Spezifikation* oder kurz *Spezifikationsbausteinen* (vgl. [31]) zusammengefasst, die im folgenden Abschnitt vorgestellt werden.

Wir sehen an den Spezifikationsbausteinen eine Reihe von Eigenschaften, die generell auf Software-Spezifikationen in großen Software-Projekten zutreffen, unabhängig von der angewandten Spezifikationsmethode:

- Jede Spezifikation setzt sich aus verschiedenen Ergebnistypen zusammen, die unterschiedliche Aspekte des zu analysierenden Systems beschreiben.
- Für die unterschiedlichen Ergebnistypen werden verschiedene Medien eingesetzt wie formale (UML-) Modelle, informelle Grafiken, strukturierte oder informelle Texte.
- Die unterschiedlichen Medien werden i. d. R. mit verschiedenen Werkzeugen erstellt.

Diese Aussagen treffen auch auf neue Vorgehensmodelle wie agile Prozesse [10] zu. Zwar wird hier keine explizite Spezifikationsphase vorgesehen, dennoch werden Anforderungen erfasst und strukturiert, z. B. mit *Storyboards* wie bei Extreme Programming (XP) [4].

Die Bausteine zur Spezifikation zeigen auch, dass sich werkzeugorientierte Ansätze bisher nicht durchgesetzt haben, in denen alle Ergebnisse in einem (CASE-)Werkzeug [3] erstellt werden. Ein solches Vorgehen wird beispielweise im Rational Unified Process (RUP) [20] empfohlen. In den von uns untersuchten Projekten gab es stets Ergebnisse, die sich nicht adäquat über ein CASE-Werkzeug darstellen ließen.

Die Menge und Vielfalt der Spezifikationsergebnisse birgt das Risiko von Redundanzen und Inkonsistenzen. Damit stehen wir vor einem weiteren Problem: Wie stellt man die Konsistenz einer Spezifikation sicher? Bisher war dies nur manuell möglich.

In diesem Artikel beschreiben wir ein allgemeines Verfahren zur Überprüfung der Konsistenz in heterogenen Dokumentmengen, die sich z. B. bei der Erstellung von Dokumentationen, Büchern oder Software-Spezifikationen ergeben. Unser Ansatz [29,30] umfasst folgende Teile, die wir am Beispiel der Spezifikationsbausteine von sd&m vorstellen:

- Formalisierung von Konsistenzbedingungen zu *Regeln* (Abschn. 3). Bisher nur natürlich-sprachlich formulierte Bedingungen werden so präzisiert. Die von uns gewählte Sprache zur Formalisierung erlaubt einen sehr reichhaltigen Regelumfang.
- Prüfung von inhaltlicher und zeitlicher Konsistenz heterogener Dokumente (Abschn. 4). Wir ermitteln exakt, wann, wo und warum Inkonsistenzen aufgetreten sind. Im Gegensatz zu Datenbanken tolerieren wir Inkonsistenzen, sodass die effiziente inkrementelle Prüfung der Konsistenz zu einer wesentlichen Herausforderung wird.
- Integration dieser Prüfung in die bewährte Arbeitsweise im Team unter Nutzung eines Versionskontrollsystems *ohne* die generelle Vorgehensweise zu beeinflussen (Abschn. 5). Dies unterscheidet unseren Ansatz von rein formalen Ansätzen zur Software-Spezifikation [35,25,2,27].

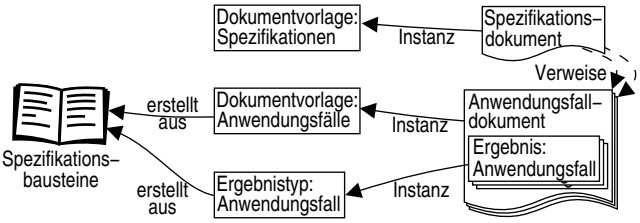

Abb. 1 Zusammenhang zwischen Ergebnistyp, Ergebnis, Dokumentvorlage und Dokument am Beispiel des Spezifikationsbausteins Anwendungsfälle

Wir gehen in diesem Artikel davon aus, dass die Anforderungen für das Software-Projekt bereits erfasst wurden und konzentrieren uns auf das Thema der Strukturierung dieser Anforderungen in der Spezifikation. Für die Herausforderungen der Anforderungsanalyse verweisen wir auf andere Publikationen (z. B. [26]).

2 Spezifikationsbausteine im Überblick

In diesem Artikel nutzen wir die bei sd&m entwickelten Spezifikationsbausteine als Methode der Spezifikation. Diese Methode fasst in der Praxis bewährte Ansätze zusammen. So spiegeln sich darin z. B. auch das Volere-Template [26] oder das Inhaltsverzeichnis eines Pflichtenhefts wider.

Jeder Spezifikationsbaustein beschreibt einen *Ergebnistyp* der Spezifikation. Im Software-Projekt werden die Ergebnistypen zu *Ergebnissen* instantiiert. Zu einem Ergebnistyp kann es in einem Projekt mehrere Ergebnisse geben. So gibt es z. B. zum Ergebnistyp „Anwendungsfall“ i. d. R. mehrere Ergebnisse, also einzelne konkrete Anwendungsfälle. Die Spezifikation umfasst alle Ergebnisse zu den benötigten Ergebnistypen (vgl. Abb. 1).

Meist hat eine Spezifikation einen so großen Umfang, dass sie auf mehrere *Dokumente* [19] aufgeteilt werden muss. Ein Dokument enthält Ergebnisse desselben Ergebnistyps. Es wird nach einer festen Dokumentvorlage für diesen Ergebnistyp erstellt.

Jeder Baustein ist eine Anleitung zur Erstellung eines spezifischen Ergebnistyps der Spezifikation. Er erläutert, was Teil des konkreten Ergebnisses ist, wie bei der Erstellung vorgegangen wird, und was besonders zu beachten ist. Der Baustein wird durch gelungene Projektbeispiele und Dokumentvorlagen ergänzt. Insgesamt haben wir nicht weniger als 17 Spezifikationsbausteine gefunden (vgl. Abb. 2). Natürlich braucht nicht jedes Projekt jeden Baustein und es muss nicht jeder Baustein in großer Ausführlichkeit beschrieben werden. Vielmehr dienen die Bausteine als Checkliste „an was alles gedacht werden muss“. Aus dieser Checkliste werden die für das Projekt relevanten Aspekte ausgewählt.

Im Folgenden stellen wir die Spezifikationsbausteine vor und geben einen Einblick in die zwischen ihnen bestehenden Konsistenzbedingungen, die für spätere Bezüge mit einer Bezeichnung R n ($n = 1, \ldots$) versehen sind.

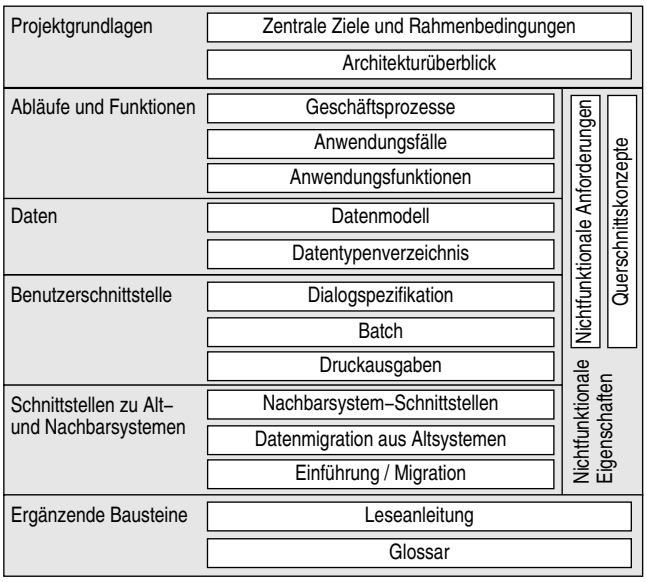

Abb. 2 Bausteine zur Spezifikation im Überblick

2.1 Projektgrundlagen

Im Baustein *Zentrale Ziele und Rahmenbedingungen* werden Zweck und Ziele des Projekts sowie des zu erstellenden Systems festgelegt. Dabei werden die Randbedingungen für das Projekt und die wichtigsten Anforderungen identifiziert. Das zugehörige Dokument umfasst nur wenige Seiten und dient als Einstieg in die Spezifikation für alle Projektbeteiligten.

Im *Architekturüberblick* wird die Ist- und die Soll-Anwendungslandschaft beschrieben. Dadurch wird das neue System in die Anwendungslandschaft des Auftraggebers eingeordnet und die wichtigsten Schnittstellen identifiziert. Diese Schnittstellen werden im Baustein *Schnittstellen* (vgl. Abschn. 2.5) detailliert.

2.2 Abläufe und Funktionen

Geschäftsprozesse beschreiben den Ablauf des Kundengeschäfts. Teile der Geschäftsprozesse werden durch das neue System unterstützt. Die Geschäftsprozesse sind wichtig für das Verständnis des mit dem neuen System zu lösenden Problems.

Geschäftsprozesse enthalten Aktivitäten. Systemunterstützte Aktivitäten werden als *Anwendungsfälle* beschrieben. Anwendungsfälle sind das Herzstück der Spezifikation. Sie beschreiben die funktionalen Anforderungen an das System aus Sicht des Anwenders. Ein Anwendungsfall kann durchaus mehrere Aktivitäten umsetzen.

R 1 Jeder Anwendungsfall setzt mindestens eine Aktivität um. Jede umgesetzte Aktivität ist in einem Geschäftsprozess definiert.

Natürlich liegen auch innerhalb der Anwendungsfälle selbst Konsistenzanforderungen vor:

R 2 Die Vorbedingungen eines Anwendungsfalls sind entweder organisatorische Ereignisse (z. B. Eingang eines Schreibens) oder sie werden durch die Nachbedingungen anderer Anwendungsfälle erfüllt (diese Anwendungsfälle sollten zeitlich davor liegen).

Komplexe Systemfunktionen ohne Interaktion eines Benutzers werden in Form von *Anwendungsfunktionen* beschrieben, die aus den Anwendungsfällen aufgerufen werden. So bleiben die Anwendungsfälle kurz und lesbar, ohne dass wichtige Informationen verloren gehen. Beispielsweise liegen der Bonitätsprüfung eines Kunden komplexe Regeln zugrunde, die in der Spezifikation erfasst und verstanden werden müssen.

R 3 Zu jeder Anwendungsfunktion gibt es mindestens einen Anwendungsfall, der diese Funktion aufruft.

R 4 Die in Anwendungsfunktionen geänderten oder erzeugten Daten sind im Datenmodell beschrieben.

2.3 Daten

Im *Datenmodell* wird eine fachliche Sicht auf die persistenten Daten des Systems erstellt. Es werden Entitätstypen, Attribute und Beziehungen beschrieben. Das Datenmodell wird in fachliche *Komponenten* unterteilt.

R 5 Eine Entität hat mehr Beziehungen zu Entitäten derselben Komponente als zu Entitäten anderer Komponenten.

Daneben gibt es noch viele weitere Bedingungen, die ein „sauberes“ Datenmodell erfüllen muss:

R 6 Beziehungen enthalten Kardinalitäten und Rollen.

R 7 Für jeden Entitätstyp sind die Attribute markiert, die zusammen den fachlichen Schlüssel ergeben.

R 8 Dieser fachliche Schlüssel ist eindeutig.

R 9 Alle Daten des Datenmodells besitzen eine nachweisbare Herkunft. Wir unterscheiden zwischen: (1) Dialogen, in denen die Daten eingegeben werden, (2) Anwendungsfunktionen, in denen die Daten berechnet werden und (3) Schnittstellen, über die die Daten ins System gelangen.

Das *Datentypenverzeichnis* enthält die fachlichen (informellen) Datentypen der Anwendung. Sie werden für die Attributtypen verwendet und legen zusätzliche Prüfungen auf diesen Typen fest.

R 10 Jeder im Datenmodell verwendete Datentyp ist im Datentypenverzeichnis definiert.

2.4 Benutzerschnittstelle

Der Benutzer kann über Dialoge, Batch-Programme oder Druckausgaben mit dem System interagieren.

In der *Dialogspezifikation* wird das Aussehen und das Verhalten der Dialoge beschrieben. Zu einem Dialog wird definiert, welche Daten angezeigt, geändert und schließlich persistiert werden. Hierdurch wird ein direkter Bezug zum Datenmodell hergestellt.

R 11 Zu jedem Dialog gibt es mindestens einen Anwendungsfall, in dem dieser Dialog verwendet wird.

Analoges gilt für Batch-Programme und Druckausgaben.

Batch-Programme werden im Baustein *Batch* dokumentiert. Ein Batch-Programm realisiert eine eigenständige Verarbeitung ohne direkten Benutzereingriff während des Ablaufs. Die fachlichen Funktionen eines Batch-Programms werden in den Anwendungsfällen beschrieben, die verarbeiteten Daten im Datenmodell. Ausgaben werden in Form einer Druckausgabe spezifiziert.

Der Baustein *Druckausgaben* beschreibt das Layout und den Inhalt von Drucklisten, Serienbriefen, etc. Druckausgaben werden durch das Druck-Layout sowie Referenzen auf Entitäten und Attribute des Datenmodells spezifiziert.

2.5 Schnittstellen zu Alt- und Nachbarsystemen

Der Baustein *Nachbarsystem-Schnittstellen* beschreibt die Schnittstellen des spezifizierten Systems zu Nachbarsystemen, mit denen dieses System kommuniziert, z. B. über Batch-Programme.

Im Baustein *Einführung / Migration* wird der organisatorische Prozess des Übergangs von der alten in die neue Anwendungslandschaft beschrieben.

Die fachlichen und technischen Aspekte der Migration enthält der Baustein *Datenmigration aus Altsystemen*. Er beschreibt, wie bei Ablösung eines Altsystems die Daten aus dem Altsystem in das neue System überführt werden müssen.

R 12 Für alle Daten des Altsystems muss sichergestellt werden, dass sie im Neusystem vorhanden sind oder nicht mehr gebraucht werden.

2.6 Nicht-funktionale Eigenschaften

Während die bisher genannten Inhalte der Spezifikation überwiegend fachliche Eigenschaften des neuen Systems betreffen, gibt es daneben noch eine Reihe von Eigenschaften, die nicht fachlicher Natur sind und einen baustein-übergreifenden Charakter haben.

Im Baustein *Nicht-funktionale Anforderungen* werden alle nicht-funktionalen Anforderungen, wie Datenqualität, Performance, Flexibilität, Robustheit oder Benutzbarkeit beschrieben. Diese stellen übergreifende Forderungen an einzelne Spezifikationsergebnisse dar (z. B. Benutzbarkeit der Dialoge) oder müssen durch die Architektur des Systems erfüllt werden (z. B. Robustheit). Auf nicht-funktionale Anforderungen kann aus allen anderen Bausteinen verwiesen werden. Hieraus ergibt sich eine Reihe von Regeln zur referentiellen Integrität in den Ergebnissen, die wir hier weglassen. Nicht-funktionale Anforderungen werden durch ein Schema beschrieben, das eine kurze Beschreibung der Anforderung, ein Testkriterium und eine Bewertung enthält.

Für bestimmte Aspekte des Verhaltens oder Aussehens des Systems ist diese kurze Form nicht ausreichend, weil es sich um komplexe Anforderungen handelt, die die Architektur des Gesamtsystems beeinflussen. Typische Beispiele dafür sind Mandantenfähigkeit, Berechtigungen und Historisierung, die i. d. R. eine ausführliche Erläuterung erfordern. Hierfür ist der Baustein *Querschnittskonzepte* vorgesehen, in dem sich eine Sammlung von Gliederungen und Beispielen für die am meisten benötigten Querschnittskonzepte findet.

2.7 Ergänzende Bausteine

Die folgenden ergänzenden Bausteine sollten Teil jeder Spezifikation sein:

Die *Leseanleitung* beschreibt den Aufbau und die Methodik der Spezifikation. Für jeden in der Spezifikation enthaltenen Baustein wird angegeben, welche Lesergruppe diesen Baustein lesen muss. Das *Glossar* erläutert anwendungsspezifische und technische Fachbegriffe. Dabei bestehen oft Überschneidungen zum Datenmodell.

R 13 Die Begriffe des Glossars sind mit den Beschreibungen im Datenmodell konsistent.

R 14 Die Definition eines Begriffs im Glossar verändert sich über die Zeit gesehen nicht wesentlich.

2.8 Medien und Werkzeuge

Während wir bisher nur erläutert haben, was alles Teil einer Spezifikation ist, steht in den Spezifikationsbausteinen auch, *wie* die Ergebnisse erstellt werden sollen. Ein Großteil der Spezifikation besteht aus *Text*. Dabei handelt es sich selten um freien, sondern überwiegend um stark strukturierten Text. Informelle Grafiken dienen der Erläuterung. Formale Grafiken werden für eine stärkere Strukturierung verwendet, z. B. UML-Aktivitätsdiagramme für Abläufe oder ein UML-Klassendiagramm für das Datenmodell.

Weil erfahrungsgemäß nicht Modelle, sondern Textdokumente den Hauptteil der Spezifikation ausmachen, verfolgen wir einen *dokumentenorientierten* Ansatz für die Spezifikation. Dies steht im Gegensatz zum werkzeugorientierten Ansatz, der z. B. im Rational Unified Process (RUP) und von vielen Herstellern von CASE-Werkzeugen empfohlen wird. Stets ist ein Textdokument der Einstiegspunkt in die Spezifikation.

Dies sieht man auch bei den Werkzeugen: Wir nutzen Textverarbeitungsprogramme deutlich stärker als Modellierungswerkzeuge. Damit möchten wir den Nutzen von Modellierungswerkzeugen nicht leugnen. Wir sehen deren Stärken jedoch eher in der Konstruktionsphase als in der Spezifikationsphase.

2.9 Dokumentversionen und zeitliche Abhängigkeiten

Im Laufe der Spezifikationsphase entstehen mehrere, aufeinander folgende Versionen eines Dokuments. Wir gehen davon aus, dass in Software-Projekten ein Versionskontrollsystem verwendet wird, das den Vergleich von Versionen, paralleles Arbeiten mehrerer Projektmitglieder und die Wiederherstellung alter Versionen erlaubt. Einen derart versionierten Dokumentbestand bezeichnen wir als *Repository*.

Jedes Dokument hat einen Lebenszyklus, der über einen Status ausgedrückt wird. Wir verwenden hier den folgenden einfachen Lebenszyklus, es kann jedoch ein beliebiger Lebenszyklus definiert werden: Während der Erstellung ist das Dokument im Status „in Arbeit“. Ist das Dokument fertig, so muss es noch der Qualitätssicherung (QS) unterzogen werden und kommt in den Status „in QS“. Ist der Kunde mit dem Inhalt des Dokuments einverstanden, so erhält es den Status „abgenommen“.

R 15 Wird ein abgenommenes Dokument geändert, so kommt es wieder in den Status „in Arbeit“. Das Dokument muss in der Folge die QS nochmals durchlaufen.

Innerhalb eines Projekts kann es mehrere abgenommene Versionen eines Dokuments geben. Dies ist z. B. der Fall, wenn zuerst eine Grobspezifikation erstellt wird, der eine Feinspezifikationsphase folgt. Dokumente, die im Rahmen der Grobspezifikation abgenommen wurden, werden dann in der Feinspezifikation verfeinert. Hieraus ergeben sich temporale Konsistenzbedingungen, die verschiedene Dokumentversionen in Beziehung setzen, z. B.:

R 16 Jede systemunterstützte Aktivität eines Geschäftsprozesses der Grobspezifikation ist auch in der Feinspezifikation vorhanden und systemunterstützt.

2.10 Konsistenz

Das Zusammenspiel der Spezifikationsbausteine wird durch eine Reihe von Konsistenzregeln bestimmt, von denen wir beispielhaft bereits einige genannt haben. Insgesamt haben wir rund 140 Regeln identifiziert. Dabei gibt es unterschiedliche Arten von Regeln:

- Regeln innerhalb eines Ergebnisses (R 6, 7).
- Regeln zwischen unterschiedlichen Ergebnissen eines Ergebnistyps. Durch die Zusammenfassung mehrerer Ergebnisse in einem Dokument bzw. Aufteilung auf mehrere Dokumente kann es sich dabei um Regeln innerhalb eines Dokuments oder zwischen mehreren Dokumenten handeln (R 2, 5, 8).
- Regeln zwischen verschiedenen Ergebnistypen, also mehreren Dokumenten (R 1, 3, 4, 9, 10, 11, 12, 13).
- Temporale Regeln, die zwischen unterschiedlichen Versionen eines oder verschiedener Dokumente bestehen (R 14, 15, 16).

Der Umfang der Konsistenzregeln und der Spezifikationsdokumente macht eine manuelle Konsistenzprüfung in einer Spezifikation im Normalfall unmöglich.

Im Folgenden beschreiben wir unseren allgemeinen Ansatz zur Konsistenzüberwachung heterogener Repositories [29] am Beispiel der Spezifikationsbausteine. Viele weitere Anwendungsgebiete sind denkbar, z. B. technische Dokumentationen, Bücher, Internet-Auftritte oder sogar Personal-Beurteilungen. Wir zeigen, wie Konsistenzregeln formal definiert (Abschn. 3), ausgewertet (Abschn. 4) und in ein Versionskontrollsystem integriert werden können (Abschn. 5). Auf diese Weise erreichen wir eine automatisierte Konsistenzprüfung, die in die Arbeitsweise in der Spezifikation nahtlos eingefügt wird. Aufwändige manuelle Konsistenzprüfung wird überflüssig. In der QS kann man sich somit wieder auf das Wesentliche konzentrieren: die fachliche Korrektheit und Vollständigkeit der Spezifikation.

Wir wollen Konsistenz bzgl. der aufgestellten Regeln nicht erzwingen. Vielmehr soll präzise dargelegt werden, *wann*, *wo* und *warum* Inkonsistenzen aufgetreten sind, um flexibel reagieren zu können. Wir erlauben also Inkonsistenzen und betrachten sie als Hilfsmittel! Eine zusätzliche Gewichtung der Regeln erlaubt Aussagen über das Ausmaß von Inkonsistenzen. Die Vorteile dieses toleranten Ansatzes haben sich bereits in vielen verwandten Themengebieten gezeigt [15, 23]. Beispielsweise wird es Ausnahmen zu Konsistenzregeln oder temporäre Inkonsistenzen aufgrund unsynchronisierter Schreibzugriffe geben. Weiterhin können Inkonsistenzen Entwurfsänderungen darstellen, die unser Ansatz dokumentiert.

3 Konsistenzregeln definieren

In diesem Abschnitt werden wir Konsistenzregeln *formal* erfassen. Nach einem kurzen Überblick definieren wir zunächst einige Symbole, die in den Regeln benutzt werden, um danach zu den Regeln selbst zu kommen. Die Formalisierung von Konsistenzbedingungen ist mit einigem technischen Aufwand verbunden. Eine automatische Erkennung von Inkonsistenzen rechtfertigt diesen Aufwand jedoch in vielen Fällen. Oft verschafft erst die Formalisierung von Regeln – und die damit einhergehende Konkretisierung – Klarheit über die eigentlichen Konsistenzanforderungen.

3.1 Überblick

Konsistenzregeln machen Aussagen über die Dokumente in einem Repository zu bestimmten Zeitpunkten. Statische Regeln (R 1 – R 13) sollen zu jedem Zeitpunkt erfüllt sein, während temporale Regeln (R 14 – R 16) verschiedene Zeitpunkte in Beziehung setzen. Unsere Beispiele zeigen deutlich, dass zur Formalisierung derartiger Konsistenzregeln eine temporale Komponente sowie

komplexe Funktionen (z. B. für den Zugriff auf Dokumentinhalte) und Prädikate (z. B. für semantische Ähnlichkeit) benötigt werden. Wir können nicht davon ausgehen, dass herkömmliche Versionskontrollsysteme die von uns verwendeten Dokumenttypen direkt unterstützen. Ein statisches Typsystem erleichtert den Formalisierungsprozess und sorgt dafür, dass nur syntaktisch korrekte Konsistenzregeln aufgestellt werden – eine essentielle Voraussetzung bei komplexen Dokumenttypen.

Konsistenzregeln werden in einer Variante der zweisortigen temporalen Prädikatenlogik erster Stufe mit linearer Zeit und Funktionssymbolen definiert [1].[1] Damit ist zwar nicht mehr entscheidbar, ob eine Konsistenzregel allgemeingültig oder unerfüllbar ist. Dies ist für unsere Anwendung jedoch nicht relevant, da wir Konsistenzregeln bzgl. *konkreter* Dokumente auswerten.[2] Anstatt temporale Operatoren wie sometimes oder always zu nutzen, wird beim zweisortigen Ansatz explizit über Zeitpunkte quantifiziert. Hierzu wird eine Sorte Time eingeführt. Wir nutzen diesen zweisortigen Ansatz, da er ausdrucksstärker ist als der Ansatz mit temporalen Operatoren und keine neuen Operatoren benötigt. Weiterhin führen wir *Typen* ein, die Sorten verallgemeinern. Bei unserem Ansatz ist Time ein Typ, der als Zeitpunkt interpretiert wird. Jeder Zeitpunkt definiert einen Repository-Zustand.

Konkret definieren wir Konsistenzregeln und die in ihnen benutzten Funktions- und Prädikatensymbole in einer XML-Syntax. Regeln arbeiten nicht direkt auf den Dokumenten, sondern auf formalen Datenstrukturen. Die dafür notwendigen Typen werden ebenfalls in einer XML-Syntax definiert. Eine grafische Benutzeroberfläche vereinfacht die Eingabe und Verwaltung der Regeln, Symbole und Typen. In diesem Artikel verwenden wir eine gekürzte formale Notation und stellen die konkrete XML-Syntax nur exemplarisch vor. Symbole müssen mit einer *Semantik* versehen („ausprogrammiert“) werden. Wir nutzen hierzu die streng typisierte funktionale Programmiersprache Haskell [24], deren Typsystem weitgehend kompatibel mit unserem Typsystem ist. Weiterhin bietet Haskell gute Voraussetzungen für den Einsatz inkrementeller Techniken zur schnellen Regelauswertung, wie wir in Abschn. 4.3 sehen werden.

3.2 Symbole und Typen

Zunächst definieren wir Funktions- und Prädikatensymbole, die in Regeln erlaubt sind. Wir benötigen z. B. Funktionssymbole, die Dokumente des Repositories in eine geeignete Datenstruktur umwandeln. Um Zeitpunkte zu vergleichen, definieren wir ein Prädikatensymbol $\leq$ und interpretieren es als lineare (zeitliche) Ordnung. Weiterhin benötigen wir (Dokument-)Typen und die Typisierung der einzelnen Symbole. Unser Typsystem unterstützt Record- und Varianten-Typen, wobei wir in weiten Teilen einer Erweiterung des Haskell-Typsystems folgen [22]. Subtypbeziehungen zwischen Record- bzw. Varianten-Typen erweisen sich als sehr nützlich.

Funktionssymbole (sd&m spezifisch):
`repD_Sp` : Time → [D_Sp] Spezifikationsdok. im Rep.
`repD_GP` : Time → [D_GP] Geschäftsprozessdok. im Rep.
`repD_AF` : Time → [D_AF] Anwendungsfalldok. im Rep.

Typdefinitionen (sd&m spezifisch außer `Doc`):
Dokument-Status (Varianten-Typen)
`Status` = {INARB | INQS | ABGEN} Status-Indikator
`Phase` = {GROB | FEIN} Spezifikationsart
Dokumente (Record-Typen)
`Doc` = {id : String; time : Time}
Dokument (Name, Eincheck-Zeitpunkt)
`D_` < `Doc` = {status : Status} sd&m Dokument
`D_Sp` < `D_` = {art : Phase; gpD, afD : [Doc]}
Spezifikationsdokument
(Art, Referenzen auf Geschäftsprozess- und Anwendungsfalldokumente)
`D_GP` < `D_` = {prozesse : [E_GP]}
Geschäftsprozessdokument
`D_AF` < `D_` = {anwFaelle : [E_AF]}
Anwendungsfalldokument
Ergebnisse (Record-Typen)
`E_` = {name, beschreibung : String}
benanntes Ergebnis mit Beschreibung
`E_GP` < `E_` = {akts : [E_GP_Akt]}
Geschäftsprozess mit Aktivitäten
`E_GP_Akt` < `E_` = {system : Bool}
Aktivität eines Geschäftsprozesses
`E_AF` < `E_` = {fuer : [R_GP]; pre, post : [String]}
Anwendungsfall (umgesetzte Aktivitäten, Vor- und Nachbedingungen)
`R_GP` = {rGP, rAkt : String}
Referenz auf Geschäftsprozess
(Name, Aktivität)

Abb. 3 Typen, Funktions- und Prädikatensymbole für Konsistenzregeln in Abb. 4 und 5. Time, `String`, `Bool` und der Listentypkonstruktor [_] sind in der Basissprache Prelude definiert. Felder eines Record-Typs stehen in geschweiften Klammern und sind durch „:“ von ihren Typen getrennt.

Die Definitionen von Typen, Funktions- und Prädikatensymbolen sind zu *Sprachen* zusammengefasst. Die Basissprache Prelude definiert bereits einige grundlegende Typen und Funktionen. Für die jeweilige Anwendung wird meist noch eine eigene Sprache definiert, die z. B. Dokumenttypen und Zugriffsfunktionen auf Dokumente enthält. Abb. 3 zeigt wesentliche Teile der anwendungsspezifischen Sprache SDM, die wir für die Spezifikationsbausteine definiert haben. Zunächst benötigen wir einige Funktionssymbole, die Dokumentinhalte in formale Datenstrukturen überführen. Das Funktionssymbol `repD_Sp`

[1] Uns sind in unserem Umfeld keine Beispiele bekannt, die den Einsatz höherstufiger Logik rechtfertigen würden.

[2] Es wäre sicher interessant zu wissen, ob eine Konsistenzregel allgemeingültig ist und wir sie demnach nicht zu prüfen brauchen. Eine Beschränkung auf entscheidbare Regeln wäre jedoch mit deutlichen Einbußen ihrer Ausdruckskraft verbunden [17].

erhält einen Zeitpunkt (Typ Time) als Parameter und gibt die im Repository zur angegebenen Zeit vorhandenen Spezifikationsdokumente (als formale Datenstruktur des Typs D_Sp) in einer Liste zurück. Analog verhalten sich die Funktionssymbole repD_GP und repD_AF für Geschäftsprozess- bzw. Anwendungsfalldokumente.

Für den Status eines Dokuments definieren wir einen Varianten-Typ Status mit den Ausprägungen INARB, INQS und ABGEN. Der Variantentyp Phase repräsentiert die Spezifikationsart, also Grob- oder Feinspezifikation. Dokumente werden durch den Record-Typ Doc repräsentiert, der den Namen id und den Eincheck-Zeitpunkt time eines Dokuments enthält. Wir verlangen, dass alle anderen Dokumenttypen Subtyp von Doc sind. Der Typ D_ ist für Dokumente von sd&m vorgesehen. Er ist Subtyp von Doc (symbolisiert durch $<$) und enthält zusätzlich zu den Feldern id und time ein Feld status. Die Typen für konkrete Dokumente sind wiederum Subtypen von D_. Ein Spezifikationsdokument D_Sp enthält neben seiner Art Referenzen auf Geschäftsprozess- (gpD) bzw. Anwendungsfalldokumente (afD), welche die entsprechenden Ergebnisse enthalten. Jeder Ergebnistyp ist Subtyp von E_, der Name und Beschreibung eines Ergebnisses enthält. Geschäftsprozesse E_GP enthalten nur ihre Aktivitäten. Aktivitäten E_GP_Akt haben zusätzlich ein Feld system, das angibt, ob sie vom System unterstützt werden. Anwendungsfälle E_AF enthalten Referenzen auf Aktivitäten in Geschäftsprozessen und je eine Liste von Vor- und Nachbedingungen (pre bzw. post). Man kann sich die jeweiligen Bedingungen mit *und* verknüpft vorstellen.

Der folgende Auszug aus der XML-Definition von SDM zeigt die Definition des Typs Status.

```
<language id="SDM">
 <importlanguage path="Prelude.xml"/>
 <typedefs>
  <typedef id="Status">
   <desc>Status eines sd&amp;m-Dokuments</desc>
   <variant><vcon id="INARB"/>
            <vcon id="INQS"/>
            <vcon id="ABGEN"/></variant>
  </typedef> ...
 </typedefs> ...
</language>
```

3.3 Regeln

Die zuvor definierten Symbole werden in Konsistenzregeln benutzt. Regeln sind geschlossene prädikatenlogische Formeln mit Quantoren über explizit angegebenen Wertebereichen. Betrachten wir hierzu die Regeln R 1 und R 2 in Abb. 4.

Regel R 1 ist eine typische referentielle Integritätsforderung. Sie wirkt etwas komplex, da in Anwendungsfällen zwar auf Aktivitäten von Geschäftsprozessen referenziert wird, aber nicht bekannt ist, in welchem *Dokument* sich

R 1: „Zu jeder Zeit t gilt für alle Anwendungsfälle a, dass sie mindestens eine Aktivität umsetzen und für jede ihrer Aktivitäten akt ein Geschäftsprozess g existiert, der die Aktivität akt definiert."

$$\forall\, t \in \texttt{repStates} \bullet \forall\, aD \in \texttt{repD\_AF}(t) \bullet$$
$$\forall\, a \in \texttt{anwFaelle}(aD) \bullet$$
$$\texttt{fuer}(a) \neq [] \;\wedge$$
$$\quad \forall\, akt \in \texttt{fuer}(a) \bullet$$
$$\quad \exists\, g \in \texttt{concatMap}(\texttt{prozesse}, \texttt{repD\_GP}(t)) \bullet$$
$$\quad \texttt{rGP}(akt) = \texttt{name}(g) \;\wedge\; \texttt{rAkt}(akt) \in \texttt{map}(\texttt{name}, \texttt{akts}(g))$$

R 2: „Zu jeder Zeit t gilt für alle Anwendungsfälle a, dass für jede ihrer Vorbedingungen pre ein anderer Anwendungsfall a_2 existiert, dessen Nachbedingungen pre enthalten."

$$\forall\, t \in \texttt{repStates} \bullet \forall\, aD \in \texttt{repD\_AF}(t) \bullet$$
$$\forall\, a \in \texttt{anwFaelle}(aD) \bullet \forall\, pre \in \texttt{pre}(a) \bullet$$
$$\exists\, a_2 \in \texttt{concatMap}(\texttt{anwFaelle}, \texttt{repD\_AF}(t)) \bullet$$
$$pre \in \texttt{post}(a_2) \;\wedge\; \texttt{name}(a) \neq \texttt{name}(a_2)$$

Abb. 4 Statische Konsistenzregeln R 1 und R 2.

ein entsprechender Geschäftsprozess befindet. Wir präzisieren daher die ursprüngliche Forderung, dass „jede umgesetzte Aktivität in einem Geschäftsprozess definiert ist". Der erste Allquantor iteriert über alle Zeitpunkte (Zustände) t des Repositories, die durch repStates in einer Liste geliefert werden. Zu jedem Zeitpunkt t ermittelt repD_AF(t) die aktuellen Anwendungsfalldokumente aD, aus denen mittels anwFaelle(aD) die jeweiligen Anwendungsfälle a extrahiert werden. Die Anwendung eines Record-Feldes auf einen Record selektiert das entsprechende Feld.[3] Mittels fuer(a) $\neq []$ stellen wir sicher, dass mindestens eine Aktivität durch den Anwendungsfall a umgesetzt wird. Da ein Anwendungsfall mehrere Aktivitäten umsetzen kann, müssen wir über sie quantifizieren. Die Variable akt iteriert über alle jemals in einem Anwendungsfall referenzierten Aktivitäten. Für sie muss es einen Geschäftsprozess g geben, dessen Name mit rGP(akt) übereinstimmt und der eine Aktivität mit gleichem Namen wie rAkt(akt) enthält. Die zur Zeit t aktuellen Geschäftsprozesse werden durch den Term concatMap(prozesse, repD_GP(t)) bestimmt: concatMap selektiert die Geschäftsprozesse aller Geschäftsprozessdokumente zum Zeitpunkt t (mittels prozesse) und fasst die entstehenden Listen zusammen (jede Selektion erzeugt eine eigene Liste von Geschäftsprozessen). Die Liste der Namen aller Aktivitäten in g bestimmen wir mittels map(name, akts(g)): map selektiert mittels name die Namen aller Aktivitäten in akts(g).[4]

In Regel R 2 gehen wir von der einfachen Formulierung der Vor- und Nachbedingungen eines Anwendungsfalls aus, die durch den Typ E_AF festgelegt wurde. Daher

[3] Record-Felder sind gleichzeitig Funktionen. Dadurch kann man sie in Funktionen höherer Ordnung nutzen.

[4] concatMap und map sind Funktionen höherer Ordnung. Das Typsystem garantiert, dass wir auf Logikebene im Kontext erster Stufe bleiben, also *nicht* über Funktionen iterieren.

R 16: „Zu jeder abgenommenen Feinspezifikationen s_f muss es eine Grobspezifikation s_g geben. s_g muss vor s_f abgenommen worden sein. Für alle in der Spezifikation s_g referenzierten Geschäftsprozessdokumente gD_{r_g} muss es jeweils ein Geschäftsprozessdokument gD_g zum angegebenen Zeitpunkt geben. Gleiches gilt für die Spezifikation s_f. Zu jedem Geschäftsprozess g_g in einem Geschäftsprozessdokument gD_g muss es einen passenden Geschäftsprozess g_f in einem Geschäftsprozessdokument gD_f geben, sodass alle Aktivitäten a_g des Geschäftsprozesses g_g auch in g_f enthalten sind und vom System genauso unterstützt werden wie a_f."

$$
\begin{array}{l}
\forall\, t_f \in \texttt{repStates} \bullet \forall\, s_f \in \texttt{repD\_Sp}(t_f) \bullet \\
(\texttt{art}(s_f) = \texttt{FEIN} \wedge \texttt{status}(s_f) = \texttt{ABGEN}) \;\Rightarrow \\
\quad \exists\, t_g \in \texttt{repStates} \bullet \exists\, s_g \in \texttt{repD\_Sp}(t_g) \bullet t_g < t_f \wedge \\
\quad \texttt{id}(s_g) = \texttt{id}(s_f) \;\wedge\; \texttt{art}(s_g) = \texttt{GROB} \;\wedge\; \texttt{status}(s_g) = \texttt{ABGEN} \wedge \\
\quad\quad \forall\, gD_{r\_g} \in \texttt{gpD}(s_g) \bullet \exists\, gD_g \in \texttt{repD\_GP}(\texttt{time}(gD_{r\_g})) \bullet \\
\quad\quad \texttt{id}(gD_g) = \texttt{id}(gD_{r\_g}) \;\wedge \\
\quad\quad\quad \forall\, gD_{r\_f} \in \texttt{gpD}(s_f) \bullet \exists\, gD_f \in \texttt{repD\_GP}(\texttt{time}(gD_{r\_f})) \bullet \\
\quad\quad\quad \texttt{id}(gD_f) = \texttt{id}(gD_{r\_f}) \;\wedge \\
\quad\quad\quad\quad \forall\, g_g \in \texttt{prozesse}(gD_g) \bullet \exists\, g_f \in \texttt{prozesse}(gD_f) \bullet \\
\quad\quad\quad\quad \texttt{name}(g_g) = \texttt{name}(g_f) \;\wedge \\
\quad\quad\quad\quad\quad \forall\, a_g \in \texttt{akts}(g_g) \bullet \exists\, a_f \in \texttt{akts}(g_f) \bullet \\
\quad\quad\quad\quad\quad \texttt{name}(a_g) = \texttt{name}(a_f) \;\wedge\; \texttt{system}(a_g) = \texttt{system}(a_f)
\end{array}
$$

Abb. 5 Temporale Konsistenzregel R 16.

reicht es aus, zu prüfen, ob die Vorbedingung *pre* in den Nachbedingungen $\texttt{post}(a_2)$ enthalten ist.

Regel R 16 fordert, dass „Jede systemunterstützte Aktivität eines Geschäftsprozesses der Grobspezifikation auch in der Feinspezifikation vorhanden und systemunterstützt ist." Da verschiedene Versionen des Spezifikationsdokuments und der Geschäftsprozessdokumente in Beziehung gesetzt werden, ist R 16 temporal. Die Formalisierung von R 16 präzisiert die obige noch recht „schwammige" Forderung (vgl. Abb. 5):

- Grob- und Feinspezifikation bezeichnen dasselbe Dokument in verschiedenen Versionen.
- In den Spezifikationen referenzierte Dokumente müssen zum angegebenen Zeitpunkt existieren.
- Geschäftsprozesse der Grobspezifikation müssen auch in der Feinspezifikation vorhanden sein.
- Aktivitäten eines Geschäftsprozesses der Grobspezifikation müssen im *gleichen* Geschäftsprozess der Feinspezifikation vorhanden sein.
- Nicht unterstützte Aktivitäten der Grobspezifikation sind auch in der Feinspezifikation nicht unterstützt.

Wir sehen, dass die Formalisierung von Regeln zu hoher *Präzision* führt: implizite Annahmen werden explizit ausgedrückt. Dies verhindert eine Reihe von Missverständnissen, die sonst immer wieder auftreten.

Weiterhin geben wir zu jeder Regel an, ob sie verletzt werden darf und wie wichtig sie ist. Im folgenden Auszug wird die Regel R 1 definiert. Sie darf verletzt werden und erhält die Gewichtung 0.7. Somit wird unser System Inkonsistenzen bzgl. R 1 erlauben (vgl. Abschn. 5). Die Gewichtung erlaubt eine Bewertung aufgetretener Inkonsistenzen.

```
<rules>
 <importlanguage path="SDM.xml"/>
 <importlanguage path="Prelude.xml"/>
 <rule name="R 1" weak="true" weight="0.7">
  <desc>Zu jeder Zeit t gilt für alle ...</desc>
  <formula><forall id="t">
            <term><apply id="repStates"/></term>
            <formula><forall id="aD"> ...
            </forall></formula>
   </forall></formula>
 </rule> ...
</rules>
```

4 Konsistenzregeln auswerten

Bevor wir die aufgestellten Konsistenzregeln formal auswerten, wollen wir an einem Beispiel unsere Erwartungen an eine automatische Auswertung demonstrieren.

4.1 Beispielspezifikation: Skischule

Wir betrachten die Spezifikation des Planungssystems einer Skischule.

Zu Beginn der Saison wird eine erste Planung erstellt, welche Kurse angeboten werden. Der Planer wählt hierbei einen von der Skischule angebotenen Standardkurstyp aus. In diesem Schritt werden den Kursen bereits Skilehrer entsprechend ihrer Verfügbarkeit zugeordnet. Für die Kurse gibt es eine maximale Belegung, die vom Kurstyp abhängt. Während der Saison werden Teilnehmer zu Kursen angemeldet. Sofern im Kurs noch Platz ist, werden die Teilnehmer direkt zugeordnet, andernfalls werden sie auf eine Wunschliste gesetzt. Dann muss die Kursplanung den Anmeldungen angepasst werden, so dass möglichst alle Kunden den gewünschten Kurs besuchen können und gute Auslastung besteht. Je nach Nachfrage fallen Kurse aus bzw. müssen zusätzliche Kurse aufgenommen werden. Dies erfordert eine Neuzuordnung der Skilehrer und Neuverteilung der Teilnehmer.

Zu Beginn der Spezifikation (Zeitpunkt 1) wird der Geschäftsprozess „GP_[Kurse planen]" erstellt, in dem je nach Bedarf neue Kurse angelegt werden. Abb. 6 zeigt einen Ausschnitt. Weiterhin wurden Anwendungsfälle für die Aktivitäten „neuen Kurs anlegen" und „angemeldete Teilnehmer zuordnen" spezifiziert. Die Struktur des folgenden XML-Dokuments `AF_1.xml` entspricht dem Dokumenttyp `D_AF`.

```
<D_AF status="INARB">
 <anwFall name="AF_[Neuer Kurs]">
  <fuer rGP="GP_[Kurse planen]"
        rAkt="neuen Kurs anlegen"/>
  <pre>Standardkurstyp angelegt</pre>
  <post>Kurs angelegt</post>
```

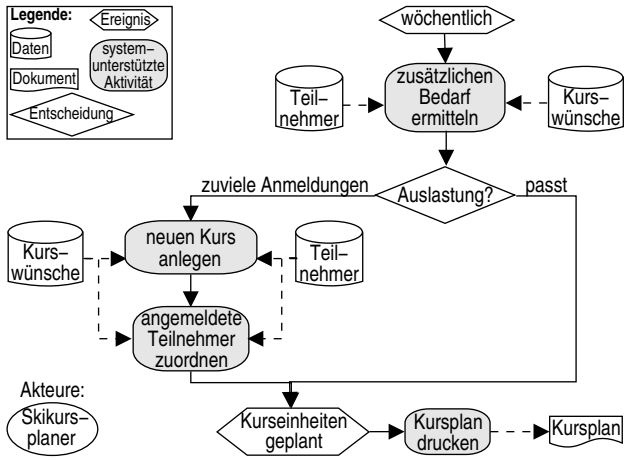

Abb. 6 Geschäftsprozess „GP_[Kurse planen]“

```
  <ablauf> ... (Ablauf wird für Konsistenzprüfung
           noch nicht verwendet) ... </ablauf>
 </anwFall>
 <anwFall name="AF_[Teilnehmer Kurs zuordnen]">
  <fuer rGP="GP_[Kurse planen]"
        rAkt="angemeldete Teilnehmer zuordnen"/>
  <pre>Kurs angelegt</pre>
  <pre>Kundendaten erfasst</pre>
  <post>Kurse haben Teilnehmer</post>
  <ablauf> ... </ablauf>
 </anwFall>
</D_AF>
```

Falls `AF_1.xml` das einzige Anwendungsfalldokument ist, so ist es inkonsistent bzgl. der Regel R 2: Für die Vorbedingungen „Standardkurstyp angelegt“ und „Kundendaten erfasst“ gibt es keinen Anwendungsfall, der eine entsprechende Nachbedingung hat (diese Anwendungsfälle wären noch zu definieren).

Zum Zeitpunkt 2 wird im XML-Dokument `AF_2.xml` ein weiterer Anwendungsfall spezifiziert, der für die Eingabe von Standardkurstypen vorgesehen ist. Er bezieht sich auf den noch nicht vorhandenen Geschäftsprozess „GP_[Kursangebote festlegen]“:

```
<D_AF status="INARB">
 <anwFall name="AF_[Standardkurstypen eingeben]">
  <fuer rGP="GP_[Kursangebote festlegen]"
        rAkt="Standardkurstypen aktualisieren"/>
  <post>Standardkurstyp angelegt</post>
  <ablauf> ... </ablauf>
 </anwFall>
</D_AF>
```

Weiterhin wird ein Spezifikationsdokument erstellt, das auf die zur Spezifikation gehörenden Dokumente inklusive ihrer Version verweist.

```
<D_Sp status="INARB" art="GROB">
 <gpD id="GP_1.xml" time="2"/>
 <afD id="AF_1.xml" time="2"/>
 <afD id="AF_2.xml" time="2"/>
</D_Sp>
```

Eine der beiden Inkonsistenzen ist nun beseitigt, da der neue Anwendungsfall die Nachbedingung „Standardkurstyp angelegt“ hat. Jedoch ergibt sich eine neue Inkonsistenz bzgl. der Regel R 1: Der referenzierte Geschäftsprozess „GP_[Kursangebote festlegen]“ ist nicht spezifiziert.

4.2 Diagnoseberichte

In unserem stark vereinfachten Beispiel sind die aufgetretenen Inkonsistenzen und deren Ursachen noch manuell feststellbar. Die Skischulspezifikation umfasst jedoch zehn Geschäftsprozesse mit 40 Anwendungsfällen. Hier ist eine automatische Konsistenzprüfung zwingend notwendig. Wie also können wir die oben aufgeführten Inkonsistenzen automatisiert feststellen?

Die klassische Semantik der Prädikatenlogik sagt nur aus, *ob* eine Konsistenzregel erfüllt ist. Sie ist also für das Aufzeigen von Inkonsistenzen unbrauchbar. Während der Auswertung prädikatenlogischer Formeln werden Variablen mit Werten belegt, z. B. die Variable aD in R 1 mit dem Anwendungsfalldokument `AF_1.xml` zum Zeitpunkt 1. Anhand dieser konkret im Repository bzw. in den Dokumenten vorhanden Werte können wir Inkonsistenzen präzise beschreiben. Anstatt nur einen Wahrheitswert b zu berechnen, generieren wir einen *Diagnosebericht*, der neben b eine Menge von Diagnosen enthält. Jede Diagnose besteht aus einem Konsistenzindikator, einer Variablenbelegung, einer Menge erfüllter Prädikate und einer Menge nicht erfüllter Prädikate. Eine Diagnose $(\mathsf{IC}, vs, ps_t, ps_f)$ entspricht der Aussage: „Die Konsistenzregel ist verletzt für die Variablenbelegung vs, da die Prädikate ps_t erfüllt sind und ps_f nicht.“ Die Variablenbelegung vs bildet quantifizierte Variablen auf Werte ab. Da wir über Zeitpunkte und Dokumente quantifizieren, legt eine Variablenbelegung präzise dar, welche Dokumente wann inkonsistent bzgl. einer Konsistenzregel sind. Die Prädikatmengen ps_t und ps_f decken die Gründe für Inkonsistenzen auf. Diagnoseberichte werden ähnlich wie der boolesche Wert bei der klassischen Semantik durch Rekursion über den Formelaufbau berechnet. Wir bewahren jedoch mehr Informationen aus den einzelnen Teilformeln, wie Variablenbelegung und Informationen über verletzte bzw. erfüllte Prädikate. Ausführlichere Informationen sind in [29] zu finden.

Für obige Skischulspezifikation generiert unser System zum Zeitpunkt 1 den in Abb. 7 dargestellten Diagnosebericht für Regel R 2.[5] Die erste Diagnose besagt, dass im Anwendungsfalldokument `AF_1.xml` zum Zeitpunkt 1 der Anwendungsfall „AF_[Neuer Kurs]“ die Vorbedingung „Standardkurstyp angelegt“ enthält und für alle Anwendungsfälle a_2 das Prädikat $pre \in \mathtt{post}(a_2)$ verletzt ist. Für die Vorbedingung „Standardkurstyp angelegt“ gibt es also keine passende Nachbedingung in irgendeinem Anwendungsfall. Variablen, die nicht in der Variablenbelegung einer Diagnose enthalten sind, können

[5] Für die anderen beiden Regeln erhalten wir $(\mathsf{True}, \emptyset)$.

$$\left(\mathsf{False}, \left\{ \begin{array}{l} \left(\mathsf{IC}, \left\{ \begin{array}{l} t \mapsto 1, aD \mapsto \texttt{D\_AF}\{\texttt{id} = \mathsf{AF\_1.xml}, \texttt{time} = 1, \ldots\}, \\ a \mapsto \texttt{E\_AF}\{\texttt{name} = \mathsf{AF\_[Neuer\ Kurs]}, \ldots\}, pre \mapsto \mathsf{Standardkurstyp\ angelegt} \end{array} \right\}, \emptyset, \{pre \in \texttt{post}(a_2)\}\right), \\ \left(\mathsf{IC}, \left\{ \begin{array}{l} t \mapsto 1, aD \mapsto \texttt{D\_AF}\{\texttt{id} = \mathsf{AF\_1.xml}, \texttt{time} = 1, \ldots\}, \\ a \mapsto \texttt{E\_AF}\{\texttt{name} = \mathsf{AF\_[Teilnehmer\ Kurs\ zuordnen]}, \ldots\}, pre \mapsto \mathsf{Kundendaten\ erfasst} \end{array} \right\}, \emptyset, \{pre \in \texttt{post}(a_2)\}\right) \end{array} \right\}\right)$$

Abb. 7 Generierter Diagnosebericht für Regel R 2 zum Zeitpunkt 1 (gekürzt)

Bericht für Regel R 1 zum Zeitpunkt 2:

$$\left(\mathsf{False}, \left\{ \left(\mathsf{IC}, \left\{ \begin{array}{l} t \mapsto 2, aD \mapsto \texttt{D\_AF}\{\texttt{id} = \mathsf{AF\_2.xml}, \texttt{time} = 2, \ldots\}, a \mapsto \texttt{E\_AF}\{\texttt{name} = \mathsf{AF\_[Standardkurstypen\ eing.]}, \ldots\}, \\ akt \mapsto \texttt{R\_GP}\{\texttt{rGP} = \mathsf{GP\_[Kursangebote\ festlegen]}, \texttt{rAkt} = \mathsf{Standardkurstypen\ aktualisieren}\} \end{array} \right\}, \right. \right. \\ \left. \left. \emptyset, \{\texttt{rGP}(akt) = \texttt{name}(g), \texttt{rAkt}(akt) \in \texttt{map}(\texttt{name}, \texttt{akts}(g))\} \right) \right\}\right)$$

Bericht für Regel R 2 zum Zeitpunkt 2:

$$\left(\mathsf{False}, \left\{ \begin{array}{l} < \text{Diagnosen für Zeitpunkt 1 (vgl. Abb. 7)} > \\ \left(\mathsf{IC}, \left\{ \begin{array}{l} t \mapsto 2, aD \mapsto \texttt{D\_AF}\{\texttt{id} = \mathsf{AF\_1.xml}, \texttt{time} = 1, \ldots\}, \\ a \mapsto \texttt{E\_AF}\{\texttt{name} = \mathsf{AF\_[Teilnehmer\ Kurs\ zuordnen]}, \ldots\}, pre \mapsto \mathsf{Kundendaten\ erfasst} \end{array} \right\}, \emptyset, \{pre \in \texttt{post}(a_2)\}\right) \end{array} \right\}\right)$$

Abb. 8 Generierte Diagnoseberichte für Regeln R 1 und R 2 zum Zeitpunkt 2 (gekürzt)

wir uns „all-quantifiziert“ vorstellen. Die zweite Diagnose zeigt, dass es zur Vorbedingung „Kundendaten erfasst“ keine entsprechende Nachbedingung gibt.

Zum Zeitpunkt 2 generiert unser System die in Abb. 8 dargestellten Diagnoseberichte für die Regeln R 1 und R 2. Der Bericht für R 1 besagt: Es gibt keinen Geschäftsprozess „GP_[Kursangebote festlegen]“, der eine Aktivität mit dem Namen „Standardkurstypen aktualisieren“ definiert. Gäbe es einen Geschäftsprozess „GP_[Kursangebote festlegen]“ ohne die Aktivität „Standardkurstypen aktualisieren“, so würde im Diagnosebericht das Prädikat $\texttt{rGP}(akt) = \texttt{name}(g)$ fehlen. Der Bericht für R 2 enthält neben den Diagnosen des letzten Berichts eine neue Diagnose, die besagt, dass zum Zeitpunkt 2 die Vorbedingung „Kundendaten erfasst“ nicht erfüllt werden kann.

Insgesamt können wir aus den generierten Diagnoseberichten und den aufgestellten Konsistenzregeln genau erkennen, wann und wo welche Inkonsistenzen aufgetreten sind. Die Typen quantifizierter Variablen tragen dabei sehr zum Verständnis bei. Die Gewichtung der Regeln wird noch nicht formal ausgewertet, erleichtert jedoch schon jetzt die Bewertung von Inkonsistenzen.

4.3 Inkrementelle Konsistenzprüfung

Die Auswertungszeit einer Konsistenzregel steigt (wie bei der booleschen Semantik auch) exponentiell mit der Quantoren-Schachtelung und polynomial mit der Anzahl und der Größe der Dokumente. Wir nutzen sowohl statische Methoden, die Regeln vor der Konsistenzprüfung vereinfachen, als auch dynamische Methoden, die Regeln *inkrementell* auswerten [30]. Die Auswertungszeit einer Regel kann beispielsweise dadurch verringert werden, indem man Quantoren soweit wie möglich in die Regel hineinschiebt, sodass sie über eine „kleinere“ Formel quantifizieren (miniscoping, vgl. [11]). Weiterhin werden nach einem Schreibzugriff auf Dokumente nur diejenigen Regeln ausgewertet, die diese Dokumente betreffen. Für auszuwertende Regeln zieht unsere inkrementelle Konsistenzprüfung soweit möglich nur Dokumente heran, die neu sind oder geändert wurden. Für alte Dokumente greifen wir auf den vorherigen Diagnosebericht zurück. Dies ist notwendig, da wir Inkonsistenzen tolerieren.

Einige unserer Ideen sind aus dem Datenbankbereich bekannt [16,13]. Zu beachten ist jedoch, dass Datenbanken ein *striktes* Konsistenzmodell implementieren: Regeln sind vor einem Schreibzugriff erfüllt. Wesentliche Voraussetzung für unser inkrementelles Vorgehen ist die Verwendung eines Versionskontrollsystems zur Darstellung von Veränderungen am Repository. Weiterhin setzen wir voraus, dass sich der Diagnosebericht einer Regel nicht ändern kann, falls die Wertebereiche ihrer quantifizierten Variablen konstant bleiben. Dies erfordert, dass Funktionen und Prädikate *referentiell transparent*[6] sind, was durch die zu ihrer Implementierung angewandte Programmiersprache jedoch garantiert ist. Ihre Ergebnisse hängen ausschließlich von ihren Parametern ab. Funktionen und Prädikate können nur über die Schnittstelle des Versionskontrollsystems auf Dokumente zugreifen.

5 Konsistenzregeln im Projekt einsetzen

Wir verteilen die Aufgaben unseres formalen Ansatzes auf verschiedene Rollen (vgl. Abb. 9) und geben den Beteiligten Werkzeuge an die Hand:

- Aus Spezifikationsbausteinen werden Dokumentvorlagen erstellt und Konsistenzbedingungen abgeleitet.
- Aus den Vorlagen werden Dokumenttypen erstellt.
- Bedingungen werden zu Regeln formalisiert.

[6] In einem Bindungsbereich hat jeder Ausdruck immer den gleichen Wert, der nur von der Konstruktion des Ausdrucks aus seinen Teilausdrücken und deren Wert abhängt.

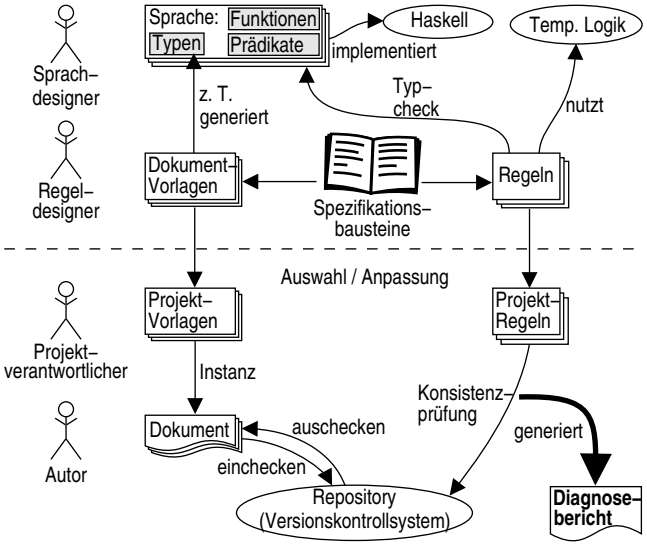

Abb. 9 Einsatz im Projekt. Nutzerdefinierbare Komponenten sind rechteckig, nicht-veränderbare Komponenten oval.

- Teilweise komplexe Funktionen und Prädikate werden deklariert und programmiert.
- Für einzelne Projekte werden Dokumentvorlagen und Konsistenzregeln ausgewählt und z. T. angepasst.

Eine zentrale Rolle spielt der *Regeldesigner*. Er erstellt aus den Spezifikationsbausteinen Dokumentvorlagen (z. B. XML-DTDs) und formalisiert Konsistenzregeln. Bei Bedarf passt er die Spezifikationsbausteine, Dokumentvorlagen und Regeln an. Der Regeldesigner benötigt daher umfassende Kenntnisse der Spezifikationsmethodik, der formalen Logik und der einsetzbaren Funktions- und Prädikatensymbole. Konkrete Programmieraspekte kann er hingegen vernachlässigen. Die aufgestellten Regelsätze importieren Sprachen, die bereits vordefiniert sind, z. B. Prelude, oder durch den Sprachdesigner erstellt wurden, z. B. SDM. Ein Typinferenzalgorithmus prüft, ob die Regeln syntaktisch korrekt sind.

In Zusammenarbeit mit dem Regeldesigner erstellt der *Sprachdesigner* Dokumenttypen aus den Dokumentvorlagen. Er deklariert weitere Typen sowie Funktions- und Prädikatensymbole und fasst sie zu Sprachen zusammen, die in verschiedenen Regelsätzen verwendet werden können. Es werden z. B. Einlesefunktionen benötigt, die Dokumente in eine Datenstruktur gemäß eines Dokumenttyps umwandeln. Die deklarierten Symbole werden in der Programmiersprache Haskell implementiert, was deren referentielle Transparenz gerantiert. Die Aufgaben des Sprachdesigners umfassen weitestgehend Programmieraufgaben, jedoch keine Formalisierung von Regeln. Bei Verwendung von XML als Dokumentformat können Dokumenttypen und Einlesefunktionen generiert werden [33].

Konsistenzregeln sind unabhängig vom verwendeten Dokumentformat. Sie arbeiten ausschließlich auf formalen Datenstrukturen, die (Dokument-)Typen entsprechen. Sollte sich das verwendete Dokumentformat ändern, z. B. vom proprietären MS-Office zu XML, müssen lediglich die Einlesefunktionen durch den Sprachdesigner geändert werden. Eine solche Migration zu Standardformaten ist nötig, um Spezifikationen für späteren Zugriff zu archivieren [8] (gute Software kann sehr lange leben).

Der *Projektverantwortliche* wählt Dokumentvorlagen und Konsistenzregeln für sein Projekt aus und passt sie ggfs. an. Unsere bisherigen Erfahrungen zeigen, dass Anpassungen selten vorkommen und eher repräsentativen als inhaltlichen Charakter haben (z. B. Änderung des Firmenlogos). Sollten dennoch größere inhaltliche Anpassungen an den Dokumentvorlagen notwendig sein, zieht dies Anpassungen in den verwendeten Typen, Funktions- und Prädikatensymbolen nach sich. Hierbei erfordern Spezialisierungen von Dokumentvorlagen nur kleinere Sprachanpassungen, da Spezialisierungen durch die Bildung von Subtypen repräsentiert werden können.

Autoren erstellen und bearbeiten Dokumente gemäß der im Projekt benutzten Dokumentvorlagen unter Nutzung eines Versionskontrollsystems. Beim Einchecken von Dokumenten in das Repository prüft zunächst das Versionskontrollsystem, ob Versionskonflikte etc. auftreten. Erlaubt es den Eincheckvorgang, so prüft unser System die Konsistenz des Repositories unter Berücksichtigung der vom Autor durchgeführten Änderungen bzgl. der im Projekt verwendeten Konsistenzregeln und generiert einen Diagnosebericht. Sollten Regeln verletzt worden sein, die erfüllt werden müssen, so verbietet unser System den Eincheckvorgang. Andernfalls reicht es aus, den betreffenden Autoren den Diagnosebericht zu schicken, z. B. per Email.

Das hier beschriebene Vorgehen macht nur geringe Annahmen über den tatsächlichen Software-Entwicklungsprozess: Es wird lediglich ein dokumentenorientierter Ansatz und die Verwendung eines (beliebigen) Versionskontrollsystems vorausgesetzt. Damit lässt sich dieses Vorgehen problemlos in verschiedene Projektmodelle mit unterschiedlicher Methodik integrieren.

Während die Spezifikationsbausteine bereits in vielen Projekten bei sd&m im Einsatz sind, wird mit der hier vorgestellten Methode für die Konsistenzprüfung zunächst nur prototypisch am Beispiel der Skischule gearbeitet. Dieses Beispiel wird bei sd&m für Schulungen zur Spezifikationsmethode verwendet und dabei stetig verbessert. Der Bedarf für ein Werkzeug zur Konsistenzprüfung wird in den Software-Projekten durchaus gesehen. Die Zurückhaltung im praktischen Einsatz liegt hauptsächlich in der bisher recht spröden Anwendung und dem noch zu erbringenden Entwicklungsaufwand z. B. für die Dokumentparser und eine grafische Benutzeroberfläche. Noch werden Spezifikationen überwiegend in MS Word erstellt. Diese Dokumente müssen zu XML transformiert werden bzw. die für die Prüfung notwendigen Merkmale in ein anderes Format migriert werden.

Auch ist das Aufstellen der Regeln zu den Spezifikationsbausteinen noch Experten vorbehalten, die sich mit der Regelsprache bestens auskennen. Hier soll ein bereits in Arbeit befindlicher grafischer Editor die Eingabe erleichtern. Weiterhin werden in Zukunft die Regeln automatisch in natürliche Sprache rückübersetzt, so dass der Regeldesigner seine Formalisierung besser überprüfen kann. Bereits jetzt haben wir aufgrund dieser Arbeit schon Ungenauigkeiten in den Spezifikationsbausteinen entdeckt, die ausgebessert werden.

Für Autoren ist der Diagnosebericht noch schwer interpretierbar. Zwei in Arbeit befindliche Ansätze führen hier zu einer Verbesserung: Analog zu den Regeln werden auch Diagnoseberichte in natürliche Sprache übersetzt. Der Diagnosebericht wird zudem automatisch ausgewertet und Reparaturvorschläge für die betroffenen Dokumente generiert. Eine der wesentlichen Herausforderungen besteht darin, die explosionsartige Vermehrung möglicher Reparaturkombinationen einzudämmen und nur wenige sinnvolle Reparaturvorschläge zu generieren. Weiterhin ist es nicht immer möglich, wirklich alle Inkonsistenzen aufzulösen. Die Gewichtung der Regeln erlangt hier also eine besondere Bedeutung.

Mit diesen Verbesserungen ist der Weg zum praktischen Einsatz geebnet.

6 Vergleich mit weiteren Ansätzen zur Konsistenzüberwachung

Da das Problem von Inkonsistenzen weithin bekannt ist, gibt es eine Reihe verwandter Ansätze, auf die wir aus Platzgründen nur kurz eingehen. Weitere Ansätze werden z. B. in [21,29] ausführlich erläutert.

Speziell für die Software-Spezifikation bieten sich einige CASE-Werkzeuge [3] an, die für die Modellierung eingesetzt werden. In der Regel lassen sich mit diesen Werkzeugen viele referenzielle Integritätsforderungen überprüfen oder die Einhaltung von Namenskonventionen sicherstellen. Dies setzt jedoch voraus, dass alle Spezifikationsergebnisse mit Hilfe des CASE-Werkzeugs erstellt werden. Im Gegensatz dazu verfolgen wir in der Spezifikation wie in der Konsistenzprüfung einen dokumentenbasierten Ansatz, welcher der Heterogenität der Dokumente Rechnung trägt und unabhängig von dem verwendeten Werkzeug ist. Unser Ansatz ist damit offen für weitere Prüfungen wie die zeitlichen Abhängigkeiten. Zudem ist der Ansatz allgemein anwendbar, also nicht nur auf das hier verwendete Beispiel Software-Spezifikationen.

Formale Ansätze zur Softwarespezifikation [35] wie Z [25] oder CASL [2] bieten stärkere Möglichkeiten zur Konsistenzüberwachung [27] als unser Ansatz. Sie erfordern jedoch erheblichen Aufwand bei der Spezifikations-Erstellung, was nicht selten eine völlige Abkehr vom für Praktiker gewohnten Vorgehen bedeutet. Dies hat bisher einen breiten Einsatz formaler Techniken verhindert. Unser Ansatz zur Konsistenzüberwachung lässt sich dagegen in die gewohnte Arbeitsweise des Software-Ingenieurs integrieren und erfordert kein Umdenken im Erstellen der Spezifikation.

Die Object Constraint Language (OCL) [34] dient der Definition von Konsistenzregeln in UML-Modellen. Obwohl der größte Teil einer Spezifikation nicht mit UML erfassbar ist, könnten wir alternativ die OCL-Syntax für Invarianten zur Regel-Definition verwenden. Wir würden dann Dokumenttypen als UML-Klassenmodell realisieren und temporale OCL-Erweiterungen [36] nutzen. OCL erlaubt die Verwendung von seiteneffektfreien Methoden {query}. Da objektorientierte Programmiersprachen immer Seiteneffekte hervorrufen, verwenden wir eine funktionale Programmiersprache, die i. A. Seiteneffektfreiheit *garantiert*. Diese Garantie ist eine Grundvoraussetzung für die inkrementelle Erstellung von Diagnoseberichten. Daher ziehen wir für Konsistenzregeln eine funktionale Syntax gegenüber einer objektorientierten Syntax vor, betrachten Syntax-Debatten jedoch als eine Frage persönlicher Vorlieben (OCL-Invarianten können in unserer Syntax abgebildet werden). Unser wesentlicher Beitrag liegt in der typsicheren Erstellung und inkrementellen Auswertung von Konsistenzregeln, sowie in der Integration von Konsistenzprüfungen in den Dokumentlebenszyklus. Dieser Beitrag wäre für OCL in gleicher Weise zu leisten.

Das Problem der Konsistenzhaltung ist im Bereich der Datenbanken eingehend studiert worden [13]. Einige Gemeinsamkeiten ergeben sich mit der Datenbankprogrammiersprache Thémis [5] – auch hier werden komplexe Funktionen höherer Ordnung in prädikatenlogischen Regeln erster Ordnung zugelassen. Wir erachten das strenge Schema einer Datenbank jedoch als ungeeignet für die Erstellung einer heterogenen Software-Spezifikation und dem beobachteten Vorgehen eines Projekt-Teams eher abträglich. Wir bevorzugen daher einen *dokumenten*orientierten Ansatz. Im Gegensatz zu unserem Ansatz kommt dem Einsatz *entscheidbarer* Logiken im Datenbankbereich eine hohe Bedeutung zu, z. B. in semistrukturierten Datenbanken [9], Content-Management-Systemen [14] oder Wissensdatenbanken [32]. Wir konzentrieren uns jedoch eher auf das präzise Aufzeigen von Inkonsistenzen in einer konkreten Spezifikation und nicht auf eine a-priori Konsistenzgarantie. Zudem waren viele der Konsistenzregeln von sd&m in einer entscheidbaren Logik nicht formalisierbar [17].

Mit `xlinkit` [21] können verteilte Dokumente auf ihre Konsistenz hin untersucht werden. Konsistenzregeln werden in einer nicht-temporalen und nicht-typisierten Prädikatenlogik erster Stufe ohne Funktionssymbole definiert. Anwendungsspezifische Prädikate werden in JavaScript implementiert. Im Gegensatz zu `xlinkit` nutzen wir ein Repository, um Dokumente zu verwalten. Dadurch wird es möglich, (1) temporale Konsistenzregeln zu formulieren, (2) *inkrementelle* Konsistenzprüfungen in die Arbeit mit einem Versionskontrollsystem zu inte-

grieren und zu automatisieren und (3) den Zugriff auf Dokumente zu kontrollieren (z. B. wird das Repository während der Konsistenzprüfung gesperrt). Unser statisches Typsystem garantiert, dass nur syntaktisch korrekte Regeln ausgewertet werden. Haskell ermöglicht wesentlich flexiblere und wiederverwendbarere Implementierungen von Funktionen und Prädikaten als JavaScript.

7 Zusammenfassung und Ausblick

In diesem Artikel haben wir die Spezifikationsbausteine von sd&m zur strukturierten Erstellung von Software-Spezifikationen verwendet. Die Spezifikationsbausteine bilden einen umfassenden und flexiblen Baukasten, den wir bereits in verschiedenen Projekten erfolgreich eingesetzt haben. Eine große Herausforderung bei diesem eher praxisorientierten und damit informellen Ansatz zur Software-Spezifikation ist die Sicherstellung der inhaltlichen Konsistenz. Wir haben Konsistenz durch Konsistenzregeln definiert, die in einer mächtigen Regelsprache formalisiert sind. Unser Ansatz ermöglicht die Formalisierung von Regeln über Dokumentgrenzen hinweg für unterschiedliche Versionen unabhängig vom verwendeten Dokumentformat. Ein von uns entwickeltes Werkzeug prüft die Konsistenz automatisch und zeigt Inkonsistenzen präzise auf. An einem Beispiel wurde dargestellt, wie unsere neue Form der Regelauswertung exakt darlegt, wo, wann und warum Inkonsistenzen in einer Software-Spezifikation aufgetreten sind. Die generierten Diagnoseberichte erleichtern die Bewertung dieser Inkonsistenzen. Die Integration der Prüfung in ein Versionskontrollsystem beschleunigt den Prozess der Prüfung durch ein inkrementelles Vorgehen und stellt eine ideale Basis für eine arbeitsteilige Problemlösung im Projektteam dar.

Wir sehen die Stärke unseres Ansatzes in der Kombination tatsächlich angewandter Praktiken mit einem formalen Verfahren zur Konsistenzsicherung. So kann die gewohnte Vorgehensweise beibehalten werden und die Projektbeteiligten können sich auf die inhaltlichen Aspekte der Spezifikation konzentrieren. Da die formalen Konsistenzregeln auf die Spezifikation „aufgesetzt“ werden, ist der Grad der Formalisierung individuell beeinflussbar. Diagnoseberichte schränken Software-Entwickler nicht ein, da wir Inkonsistenzen tolerieren – sie werden lediglich aufgezeigt. Die Reaktion auf Inkonsistenzen bleibt dem Software-Entwickler überlassen. Wir denken, dass dieser Ansatz wesentlich bessere Aussichten auf einen breiten praktischen Einsatz hat als rein informelle, rein werkzeugorientierte oder rein formale Ansätze.

Daneben erwarten wir von unserem Ansatz Verbesserungen auf Seiten der Methodik. Während bisher Konsistenzbedingungen nur in natürlicher Sprache und verteilt über alle Spezifikationsbausteine formuliert waren, werden sie nun formalisiert und in einem Regelwerk konzentriert. Dies führt zu einer Kontrolle der Bausteine und ihrer Abhängigkeiten. Erkannte Inkonsistenzen und Redundanzen werden korrigiert. Die Gewichtung der Regeln leitet den Software-Ingenieur auf kritische Punkte der Spezifikation. Durch Feedback aus den Projekten können wir feststellen, welche Regeln besonders oft verletzt werden. Damit ergeben sich wichtige Anhaltspunkte zur weiteren Verbesserung unseres Vorgehensmodells.

Wir haben unseren Ansatz bereits vollständig implementiert. Unser Prototyp arbeitet mit dem Versionskontrollsystem darcs [28] zusammen und generiert die dargestellten Diagnoseberichte. Anbindungen an andere Systeme, z. B. CVS, sind geplant. Derzeit entwickeln wir Werkzeuge, die dem Software-Ingenieur den praktischen Einsatz unseres Ansatzes erleichtern werden: ein grafischer Editor, Bibliotheken für Zugriffsfunktionen sowie Interpreter, die Regeln und Diagnoseberichte in natürliche Sprache übersetzen. Präzise Diagnoseberichte sind nur ein erster Schritt zu mehr Konsistenz im Software-Entwicklungsprozess. Daher erweitern wir unser Werkzeug um die Generierung konkreter Vorschläge, wie Inkonsistenzen aufgelöst werden können. Hierbei nutzen wir insbesondere die Gewichtungen der Regeln. Zudem wird es bei sd&m nicht nur beim Einsatz in der Spezifikation bleiben. Die erste Version der *Konstruktions*bausteine liegt bereits vor. Eine enge Verzahnung dieser beiden Baukästen für die Phasen Spezifikation und Konstruktion ist in Arbeit. Es werden nicht nur analoge Regeln für die Konsistenz der Konstruktionsbausteine benötigt, sondern auch Regeln für den Übergang von der Spezifikation zur Konstruktion.

Durch schrittweise Erweiterungen erwarten wir einen umfassenden Ansatz, der zu einer Verbesserung des Software-Entwicklungsprozesses und der Software-Qualität insgesamt führt. Unsere bisher erzielten Ergebnisse untermauern diese These.[7]

Literatur

1. S. Abiteboul, L. Herr, and J. van den Bussche. Temporal versus first-order logic to query temporal databases. In *ACM Symposium on Principles of Database Systems*, pages 49–57, Montreal, Canada, 1996. ACM.
2. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
3. H. Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, volume 2. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.
4. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, Reading, MA, 2000.
5. V. Benzaken and A. Doucet. Thémis: A database programming language handling integrity constraints. *VLDB Journal*, 4(3):493–518, 1995.

[7] Weitere Informationen finden Sie auf der WWW-Seite www2-data.informatik.unibw-muenchen.de/cde.html

6. G. Booch. *Objektorientierte Analyse und Design*. Addison Wesley, Reading, MA, 1994.
7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.
8. U. Borghoff, P. Rödig, J. Scheffczyk, and L. Schmitz. *Langzeitarchivierung: Methoden zur Erhaltung digitaler Dokumente*. dpunkt.Verlag, Heidelberg, Germany, 2003.
9. P. Buneman, W. Fan, and S. Weinstein. Path consistency rules in semistructured databases. *Journal of Computer and System Sciences*, 61(2):146–193, 2000.
10. A. Cockburn. *Agile Software Development*. Addison Wesley, Reading, MA, 2001.
11. D. de Champeaux. Subproblem finder and instance checker, two cooperating modules for theorem provers. *Journal of the ACM*, 33(4):633–657, 1986.
12. T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.
13. M. e Silva. Dynamic integrity constraints definition and enforcement in databases: a classification framework. In *Proc. of the IFIP TC-11 Working Group 11.5 First Working Conf. on Integrity and Internal Control in Information Systems*, pages 65–87, 1997.
14. M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Verifying integrity consistency rules on web sites. In *IJCAI*, pages 614–619, 1999.
15. A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Proc. of the 11th Int. Conf. on Database and Expert Systems Applications (DEXA)*, pages 1–5, London, UK, 2000. Springer.
16. A. Gupta, Y. Sagiv, J. Ullman, and J. Widom. Constraint checking with partial information. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 45–55, Minneapolis, MS, 1994. ACM.
17. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.
18. I. Jacobson, M. Christerson, P. Johnson, and F. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, Wokingham, England, 1992.
19. U. Kampffmeyer and B. Merkel. *Dokumentenmanagement: Grundlagen und Zukunft*. Project Consult, 1999.
20. P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 3rd edition, 2003.
21. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. In *ACM Transactions on Internet Technology*, pages 151–185, 2002.
22. J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers Tekniska, Högskola, 1999.
23. B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.
24. S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
25. B. Ratcliff. *Introducing Specification Using Z: A Practical Case Study Approach*. McGraw Hill Book Co Ltd, 1994.
26. J. Robertson and S. Robertson. *Mastering the Requirements Process*. Addison Wesley, 1999.
27. M. Roggenbach and L. Schröder. Towards trustworthy specifications I: consistency checks. In *Recent Trends in Algebraic Development Techniques, 15th Int. Workshop, WADT'01*, volume 2267 of *LNCS*, pages 305–327. Springer, 2001.
28. D. Roundy. Darcs: David's advanced revision control system, 2004. see www.abridgegame.org/darcs/.
29. J. Scheffczyk, U. Borghoff, P. Rödig, and L. Schmitz. Consistent document engineering. In *Proc. of the 2003 ACM Symp. on Document Engineering*, pages 140–149, Grenoble, France, 2003. ACM Press.
30. J. Scheffczyk, U. Borghoff, P. Rödig, and L. Schmitz. Efficient (in-)consistency management for heterogeneous repositories. In *Proc. of the 4th Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 370–377, Lübeck, Germany, 2003. ACIS.
31. J. Siedersleben and W. Krug. Bausteine der Spezifikation. In J. Siedersleben, editor, *Softwaretechnik*. Hanser, 2. edition, 2002.
32. A. I. Vermesan and F. Coenen, editors. *Validation and Verification of Knowledge Based Systems – Theory, Tools and Practice, Papers from EUROVAV '99, 5th European Symp. on Validation and Verification of Knowledge Based Systems, Oslo, Norway*. Kluwer, 1999.
33. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the Fourth ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 1999. ACM Press.
34. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.
35. M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675–788. Elsevier Science Publishers, 1990.
36. P. Ziemann and M. Gogolla. An extension of OCL with temporal logic. In J. Jürjens et al., editors, *Critical Systems Development with UML – Proc. of the UML'02 workshop*, pages 53–62, Munich, Germany, 2002.