Hierarchische Familien formaler Anforderungen

Jan Bredereke
Universität Bremen, FB Informatik · Postfach 33 04 40 · D-28334 Bremen
jan.bredereke@topmail.de · www.tzi.de/jan.bredereke

Zusammenfassung

Wir beschäftigen uns mit der Wartung von formalen Anforderungsdokumenten. Ziel ist die Vermeidung bzw. die Erkennung von Feature-Interaktions-Problemen in Telefonvermittlungssystemen. Dazu behandeln wir die vielen Varianten von Anforderungsdokumenten gemeinsam als eine Familie von Anforderungen und bieten dafür Sprachunterstützung. Außerdem ordnen wir die optionalen Leistungsmerkmale in einer expliziten Hierarchie an. Durch das explizite Dokumentieren von wichtigen Abhängigkeitsinformationen wird die Erkennung von (bestimmten) Feature-Interaktions-Problemen durch eine Typüberprüfung möglich.

1 Einleitung

Digitale Telefonvermittlungssysteme umfassen bereits hunderte von Leistungsmerkmalen (Features), und der Markt drängt die Hersteller, immer weitere neue Dienste und Leistungsmerkmale dazuzufügen. In den letzten Jahren sind dabei *Feature-Interaktions-Probleme* ein ernsthaftes Hindernis geworden [1, 2, 3, 4, 5]. Das Hinzufügen eines weiteren Leistungsmerkmals, selbst wenn es für sich genommen wie gewünscht funktioniert, kann zusammen mit anderen Leistungsmerkmalen in unerwünschtem Verhalten resultieren, selbst wenn diese für sich genommen ebenfalls wie gewünscht funktionieren. Die Anzahl der möglichen Kombinationen von Leistungsmerkmalen ist dabei inzwischen so groß geworden, daß man nicht mehr alle Kombinationen überprüfen kann.

Ein Standardbeispiel für eine solche Feature-Interaktion entsteht zwischen dem Leistungsmerkmal Terminating Call Screening (TCS) und dem Leistungsmerkmal Call Forwarding (CF). TCS erlaubt einem Abonnenten C eine Liste von Anrufern anzugeben, von denen er niemals angerufen werden möchte. CF erlaubt einem Abonnenten B ein Ziel anzugeben, zu dem alle Anrufe für B weitergeleitet werden sollen. Nehmen wir nun an, daß C TCS abonniert hat und A auf seine Schwarze Liste gesetzt hat. Nehmen wir weiterhin an, daß B CF abonniert hat und alle Anrufe zu C weiterleitet. Und es versuche A, B anzurufen. Wird das Telefon von C klingeln? Sollte es das? Dies hängt sehr davon ab, *wie wir den Begriff des „Anrufers“ definieren.* Wenn CF mit zwei „zusammengeklebten“ Anrufen realisiert wird, dann ist B der Anrufer für C, die Blockierbedingung ist nicht erfüllt, und das Telefon klingelt. Wenn wir berücksichtigen, daß C wohl nicht von A belästigt werden möchte, sollte das Telefon nicht klingeln. Zave hat solche „Geheimnisse der Anrufweiterschaltung“ ausführlich diskutiert [6]. Wir werden ebenfalls auf dieses Beispiel noch zurückkommen.

Das Beispiel zeigt, daß die Feature-Interaktionen bereits in den Anforderungsdokumenten zu finden sind. Und sofern diese Dokumente eine *vollständige* Beschreibung der Verhalten (und anderer interessierender Eigenschaften) enthalten, dann sind sogar *alle* Feature-Interaktionen zumindest inhärent in den Anforderungsdokumenten vorhanden. Deshalb sollten sie auch bereits dort behandelt werden. (Natürlich müssen wir immer noch sicherstellen, daß eine Implementation korrekt ist bezüglich ihrer Anforderungsspezifikation. Aber dies ist etwas anderes als die unerwünschten Interaktionen von Leistungsmerkmalen, die für sich genommen wie gewünscht funktionieren.)

Der Begriff der vollständigen Anforderungen führt uns zu formalen, d.h. präzisen Anforderungen. Hierzu gibt es bereits umfangreiche Arbeiten. Ein Aspekt, der dabei allerdings mehr Beachtung verdient, ist die Wartung von formalen Anforderungsdokumenten. Was macht man, wenn sie im Laufe der Zeit oft geändert werden müssen, und wie organisiert man eine große Anzahl von Varianten für solche Dokumente, wenn verschiedene Kunden eines erfolgreichen Produkts stets leicht verschiedene Wünsche haben?

In Kapitel 2 geben wir zunächst einen kurzen Überblick über die von uns verwendete Formale Beschreibungstechnik. In Kapitel 3 beschreiben wir anhand eines Beispiels, wie wir Anforderungen hierarchisch strukturieren, um zu einem System neue Leistungsmerkmale hinzufügen zu können, und in Kapitel 4 zeigen wir, wie wir eine ganze Familie von formalen Anforderungen organisieren. Kapitel 5 behandelt die automatische Erkennung von Feature-Interaktionen, und Kapitel 6 faßt die Ergebnisse zusammen.

2 Der Formalismus CSP-OZ

Als formale Beschreibungstechnik zur Spezifikation von Anforderungen verwenden wir CSP-OZ [7, 8, 9], das einen ausgearbeiteten Verfeinerungsbegriff bereits mitbringt. CSP-OZ basiert auf der Prozeßalgebra CSP (Communicating Sequential Processes) [10, 11] und erlaubt daher, die Steuerungs- und Kommunikationsaspekte eines verteilten Systems zu spezifizieren. Der CSP-Teil wird mit Object-Z [12] kombiniert, einer objektbasierten Erweiterung von Z [13, 14]. Object-Z und Z sind eine Formalisierung der Mengentheorie, und sie sind besonders geeignet, um Datenaspekte eines Systems zu spezifizieren. Zum Beispiel bieten sie die Konzepte der abstrakten Datentypen und der Zustandsinvarianten, die es in CSP nicht gibt.

CSP-Spezifikationen sind über Ereignissen definiert. Man kann Prozesse mit Operatoren wie z.B. der Parallel- und der sequentiellen Komposition, der Alternative und dem Hiding kombinieren. Die Semantik eines Systems besteht aus seinen Traces, Failures und Divergenzen. Es gibt zahlreiche algebraische Gesetze über die Verhaltensverfeinerung. Eine Object-Z-Klasse besteht aus der Spezifikation eines Zustandsraumes und aus Operationen über ihm. Es gibt zahlreiche algebraische Gesetze über die Datenverfeinerung. In der Semantik von CSP-OZ werden die Operationen des Object-Z-Teils als CSP-Ereignisse interpretiert und machen zusätzliche Einschränkungen für das Verhalten des CSP-Teils.

Es gibt Werkzeugunterstützung für CSP, insbesondere den mächtigen, kommerziellen Model-Checker FDR (Failures-Divergence Refinement) [15]. An der automatischen Übersetzung einer Teilmenge von CSP-OZ nach FDR wird zur Zeit gearbeitet [16, 17], und es gibt bereits das Werkzeug cspozTC [18] zur Typüberprüfung für CSP-OZ, das wir auch für unsere Fallstudie eingesetzt haben.

3 Hierarchische Anforderungsspezifikation

Telefonvermittlungssysteme werden ständig erweitert und geändert, und mit ihnen die Anforderungsdokumente. Dabei setzten die Änderungen früher üblicherweise immer auf der letzten Version auf. Inzwischen gibt es einige Bestrebungen, ein Basissystem zu definieren, relativ zu dem dann alle Änderungen definiert werden. Ein Beispiel ist der ITU-T-Standard für das Intelligente Netzwerk (IN) [19, 20].

Der letztere Ansatz erleichtert zwar die Einführung neuer Leistungsmerkmale, aber es kommt trotzdem zu Feature-Interaktions-Problemen. Ein Beispiel haben wir bereits in der Einleitung beschrieben, die unerwünschte Interaktion zwischen TCS und CF. Der Grund für diese Interaktion ist, daß die Anrufweiterschaltung CF die Art erweitert, auf die das System den Benutzer identifiziert, mit dem es den Anrufer verbinden soll, und diese Art damit ändert. Alle Leistungsmerkmale, die diese Definition verwenden, müssen daher aufbauend auf CF definiert werden, genauer gesagt aufbauend auf einen erweiterten Begriff der Benutzeridentifikation. Im Intelligenten Netzwerk ist eine solche geschichtete Architektur nicht möglich, da jedes Leistungsmerkmal zwar die Schnittstelle des Basissystems nutzen kann, aber nicht selbst wieder eine Schnittstelle zur Verfügung stellen kann.

In einer Fallstudie [21, 22] haben wir die Anforderungen an ein Telefonvermittlungssystem mit CSP-OZ spezifiziert. Die Spezifikation ist *hierarchisch* aufgebaut, mit Hilfe der in CSP-OZ vorhandenen Konzepte der Klasse und der Vererbung können komplexe Eigenschaften aus einfachen aufgebaut werden. Wir demonstrieren die Struktur am Beispiel von Ausschnitten aus zwei einfachen Leistungsmerkmalen und aus dem Basis-Telefonsystem.

3.1 Das Basis-Telefonsystem

Am untersten Ende der Hierarchie stehen die Basisbegriffe der Telefonie. Das Basis-Telefonsystem führt sie ein. Die Klasse HumanEss führt die Menge der Menschen ein, modelliert als Variable humans. Klassen werden in CSP-OZ syntaktisch durch einen Kasten dargestellt, in dessen Kopf der Klassenname steht. Der Zustandsraum der Klasse wird in einem darin verschachtelten, namenlosen Kasten spezifiziert. Die Klasse DeviceEss führt die Menge der Terminalgeräte devices ein, sowie die Zuordnung von Menschen zu Geräten possessedDevices, und die Klasse UserEss führt die Menge der Benutzerrollen users von Menschen ein, sowie die Zuordnung von Menschen zu Benutzerrollen userRoles:

HumanEss

...

$\mathsf{humans} : \mathbb{F}\,\mathsf{humanIds}$

UserEss

`inherit` HumanEss

...

$\mathsf{users} : \mathbb{F}\,\mathsf{userIds}$
$\mathsf{userRoles} : \mathsf{humanIds} \to \mathbb{F}\,\mathsf{userIds}$

DeviceEss

`inherit` HumanEss

...

$\mathsf{devices} : \mathbb{F}\,\mathsf{deviceIds}$
$\mathsf{possessedDevices} : \mathsf{humans} \to \mathbb{F}\,\mathsf{deviceIds}$

Für ein einfaches, traditionelles Telefonvermittlungssystem wird zunächst spezifiziert, daß die Elemente dieser Mengen alle in Eins-zu-Eins-Relationen stehen, und daß diese Mengen statisch sind. Zum Beispiel soll die Zuordnung zwischen Menschen und Geräten sich niemals ändern:

DevAssocNoRem

`inherit` HumanEss DeviceEss ...

$$\forall\, h : \mathsf{humans} \bullet \quad \mathsf{possessedDevices}(h) = \mathsf{initialPossessedDevices}(h)$$

Weiterhin beschreibt die Spezifikation, wie eine Verbindung auf- und abgebaut wird. (Aus Platzgründen lassen wir die recht umfangreiche Beschreibung davon fort und bringen nur Beispiele einfacher Eigenschaften.) Als Voraussetzung für eine Verbindung wird u.a. die Verfügbarkeit verschiedener Ressourcen spezifiziert. Eine dieser Ressourcen ist, daß der Ziel-User verfügbar ist, was folgendermaßen spezifiziert ist:

ConEssUserAvailable

`inherit` UserEss

$\mathsf{userAvailableType} ::= \mathsf{userAvail} \mid \mathsf{userNotAvail}$

$\mathsf{userAvailable} : \mathsf{userIds} \times \mathsf{userIds} \to \mathsf{userAvailableType}$

3.2 Leistungsmerkmal Schwarze Liste

Bis zu diesem Punkt haben wir ein Telefonsystem ohne besondere Leistungsmerkmale. Aber das soll sich ändern. Als nächstes führen wir die grundsätzliche Möglichkeit ein, daß Verbindungen nach irgendwelchen Kriterien verhindert werden können. Im folgenden beschreibt die Variable connectionScreen der Klasse ConScreenEss die User-Paare, die niemals verbunden werden sollen. Wir spezifizieren das Screening dabei als eine Form von fehlender Ressource:

ConScreenEss

`inherit` ...

connectionScreenType ::= connectionScreenOk | connectionScreenBlock

$$\text{connectionScreen} : \text{userIds} \times \text{userIds} \to \text{connectionScreenType}$$

$$\forall\, u, v : \text{users} \bullet$$
$$(\text{connectionScreen}(u, v) = \text{connectionScreenBlock}) \Rightarrow (\text{userAvailable}(u, v) = \text{userNotAvail})$$

Etwas umfangreicher ist wiederum die Definition einer Schwarzen Liste unerwünschter Anrufer, einschließlich des Setzens und Löschens von Einträgen darin. Daher lassen wir sie hier fort. Interessant ist allerdings, wie die Schwarze Liste blackListUsr an das allgemeine Screening angekoppelt wird:

BlackListUsrOnly

`inherit` ConScreenEss

$$\forall\, u, v : \text{users} \bullet$$
$$((\text{connectionScreen}(u, v) = \text{connectionScreenBlock}) \Leftrightarrow ((u, v) \in \text{blackListUsr}))$$

Um später die Eigenschaften aus allen Klassen des Leistungsmerkmals der Schwarzen Liste zusammen verwenden zu können, definieren wir per Konvention eine Klasse, die nur genau diese Klassen erbt. Genauer gesagt definieren wir *zwei* Klassen, da wir zwischen den Anforderungen unterscheiden, die das System erfüllen muß, und denen, auf die es sich verlassen kann, weil seine Umgebung sie erfüllen muß:

BlackListOfUsersFeatureSys

`inherit` BlackListUsrOnly ...

BlackListOfUsersFeatureEnv

`inherit` ...

3.3 Leistungsmerkmal Anrufweiterschaltung

Außerdem wünschen wir ein Leistungsmerkmal FollowHumanConnectionForwarding, bei dem Menschen sich von einem Telefongerät zu einem anderen begeben können, um sich dort zu registrieren und anschließend dort alle Verbindungen abzuwickeln. Im wesentlichen ist es dafür nur notwendig, eine der obigen Anforderungen über die statische Zuordnung zwischen Menschen und Geräten aufzuheben. Dies geschieht dadurch, daß eine der Klassen des Basis-Telefonsystems wieder entfernt wird. Der Operator dafür wird im nächsten Kapitel ausführlicher diskutiert, er sieht folgendermaßen aus:

`remove` DevAssocNoRem

Um die Implementation dieser Spezifikation zu vereinfachen, sieht das Leistungsmerkmal außerdem vor, daß das Registrieren nur erfolgen darf, wenn keine Verbindung mit diesem Menschen besteht. (Die Spezifikation dieser Bedingung lassen wir hier aus Platzgründen fort.)

Nachdem wir wiederum die Klassen dieses Leistungsmerkmals in zwei Klassen zusammengetragen haben, fügen wir schließlich eine vollständige Spezifikation aus den einzelnen Teilen zusammen, indem wir per Vererbung die Sammelklassen der einzelnen Leistungsmerkmale einsammeln:

SpecificationCSys

`inherit` ... BlackListOfUsersFeatureSys FollowHumanConnectionForwardingFeatureSys

SpecificationCEnv
`inherit` ... BlackListOfUsersFeatureEnv FollowHumanConnectionForwardingFeatureEnv

Die eigentliche Spezifikation ist dann eine Parallelkomposition aus den Anforderungen an das System und den Anforderungen an die Umgebung dieses Systems. Die sichtbaren Ereignisse sind dabei genau die, die wir per Vererbung von den Leistungsmerkmalen eingesammelt haben; wir schließen sie mit einem speziellen Operator in die Spezifikation ein:

SpecificationC
`inherit-interface` SpecificationCEnv
$main \;=\;$ SpecificationCEnv $\|$ SpecificationCSys

3.4 Diskussion

Wir fassen das Vorgehen in der Fallstudie zusammen: Zum einen legen wir Konzepte im Basissystem bereits an, wenn absehbar ist, daß wir sie für spätere Leistungsmerkmale benötigen werden. Ein Beispiel ist die Trennung von Menschen, Rollen und Geräten. Dabei halten wir die zusätzliche Arbeit so gering wie möglich, indem wir gleich anschließend die änderbare Eigenschaft spezifizieren, daß zunächst kein Unterschied besteht. Trotzdem erlaubt uns dies, im weiteren System bereits jeweils den richtigen dieser Begriffe zu verwenden, so daß wir später nicht mehr die ganze Spezifikation durchgehen müssen, um etwa den Begriff „Telefonnummer" in jeweils den richtigen der präziseren Begriffe umzubenennen. Eine solche Aufspaltung ist erfahrungsgemäß aufwendig, und sie führt leicht zu Feature-Interaktions-Problemen, wenn sie nicht vollständig und korrekt durchgeführt wird. Wir haben dies am Begriff des Anrufers im Beispiel in der Einleitung gesehen. In unserer Fallstudie tritt keine unerwünschte Feature-Interaktion auf, weil Verbindungen zwischen Benutzerrollen hergestellt werden, weil bei einem Wechsel eines Menschen zu einem anderen Gerät auch dessen Benutzerrollen mitwechseln und weil die Schwarze Liste unerwünschte Benutzer(rollen) ausfiltert. Das Telefon wird also nicht klingeln.

Wir spezifizieren constraint-orientiert. Das heißt, daß wir Variablen und Ereignisse zunächst mit maximalem Nichtdeterminismus einführen und dann schrittweise Einschränkungen machen. Ein Beispiel ist die Eigenschaft im Basissystem, daß eine Verbindung nur zustande kommen kann, wenn die notwendigen Ressourcen vorhanden sind. Welche Ressourcen dies konkret sind, wird dabei erst im nächsten Schritt festgelegt. Die Variable userAvailable stellt also eine Schnittstelle des Verbindungsaufbau-Moduls dar, an der andere Module ansetzen können.

Wir wenden diesen Ansatz mehrstufig an: Als wir erkannten, daß ein Eintrag in einer Schwarzen Liste als eine Art von fehlender Ressource angesehen werden kann, führten wir eine allgemeine Screening-Ressource durch die Variable connectionScreen in der Klasse ConScreenEss ein. Sie stellt eine neue Schnittstelle für Screening-Leistungsmerkmale dar, in unserem Fall setzt die Schwarze Liste unerwünschter Benutzer darauf auf. Wichtig ist dabei, rechtzeitig das neue allgemeine Konzept zu erkennen und getrennt vom aktuellen Leistungsmerkmal zu spezifizieren.

Diese Trennung der Spezifikation von Grundkonzepten von der ihrer aktuellen Ausprägungen sowohl im Basissystem als auch in den aktuell vorhandenen Leistungsmerkmalen setzen wir so hierarchisch fort. Auf die vorhandenen Schnittstellen der Klassen werden per Vererbung hierarchisch neue Klassen mit neuen Schnittstellen aufgesetzt. Im Intelligenten Netzwerk (s.o.) ist dies im Gegensatz dazu nicht möglich.

4 Familien von Anforderungen

Im vorigen Kapitel haben wir gezeigt, wie neue Leistungsmerkmale auf bestehenden Anforderungen aufbauen können. In diesem Kapitel beschreiben wir, wie wir die vielen Varianten von Anforderungen organisieren.

Damit man bei der Wartung und Fortschreibung der Anforderungen den Überblick behält, halten wir alle Varianten der Anforderungen in *einem* zusammenhängenden Dokument. Zur Strukturierung bekommt jede Variante ihr eigenes Kapitel. Immer wenn wir ein neues Leistungsmerkmal, d.h. eine neue Variante der bisherigen Anforderungen definieren, hängen wir ein neues Kapitel an das Dokument an. Immer wenn ein konkretes System aus den Varianten zusammengestellt werden soll, geben wir die Liste der gewünschten Leistungsmerkmale in einen automatischen *Generator* ein, der aus dem Hauptdokument die eigentliche Anforderungsspezifikation in CSP-OZ generiert. Die vielen möglichen Varianten, die das Hauptdokument anbietet, stellen also eine implizite, große Familie von sehr vielen generierbaren CSP-OZ-Spezifikationen dar.

> Wir definieren eine *Familie von Anforderungen* als eine Menge von Anforderungsdokumenten, die so viele gemeinsame Eigenschaften haben, daß es sich lohnt, die gemeinsamen Eigenschaften zu studieren, bevor man die einzelnen Mitglieder analysiert.

Dieser Begriff ist verwandt mit dem der Familien von Programmen, der von Parnas [23] geprägt wurde. Parnas beschäftigte sich mit Mengen von Programmen, deren Mitglieder sehr ähnlich sind, zum Beispiel, weil im Laufe der Zeit immer weitere Varianten eines Ursprungsprogramms implementiert werden müssen. Er schlug vor, auch die einzelnen Zwischenschritte bei der Entwicklung des ersten Programms zu dokumentieren und aufzubewahren. Wenn dann für eine Variante einzelne Entwurfsentscheidungen anders getroffen werden müssen, kann man alle Schritte bis zu diesem Punkt direkt wiederverwenden. Der zweite wichtige Vorschlag dabei ist, die Zwischenschritte nicht chronologisch zu dokumentieren, sondern nach inhaltlichen Abhängigkeiten in einem Baum. Der Ansatz ist als *Geheimnisprinzip* bekanntgeworden, und die dokumentierten Zwischenschritte als *Information-Hiding-Module*. Ein solches Modul kapselt eine Entwurfsentscheidung als ein Geheimnis, und andere Module außerhalb dieses Moduls sind unabhängig von möglichen Änderungen dieser Entscheidung, da sie nur auf die Schnittstelle des Moduls zugreifen dürfen, an der sein äußeres Verhalten beschrieben ist.

Im vorigen Kapitel haben wir bereits gesehen, wie wir auch für die Spezifikation von Anforderungen die Vererbung von CSP-OZ-Klassen einsetzen können, um Zwischenschritte bei der Anforderungsspezifikation zu dokumentieren. Die Klassen sind durch die Vererbungsrelation im der Vererbungshierarchie angeordnet und dokumentieren damit die inhaltlichen Abhängigkeiten explizit.

Diese Zwischenschritte können wir nun auch für die Anforderungsspezifikation wiederverwenden, indem wir Teile der Klassenhierarchie, wiederum durch Vererbung, zu einem konkreten, kompletten System zusammensetzen. Diesen Schritt führen wir für jedes konkrete Anforderungsdokument aus, das wir realisieren wollen.

Bei der Zusammenstellung eines Systems wollen wir nicht mit einer unüberschaubaren Zahl von CSP-OZ-Klassen hantieren, von denen wir jede dafür benutzen, nur genau eine in sich abgeschlossene Eigenschaft des Systems zu spezifizieren. Stattdessen gruppieren wir die Klassen zu Leistungsmerkmalen, und wir wählen nur ganze Leistungsmerkmale bei der Zusammenstellung an oder ab. Hierfür haben wir einige neue Sprachkonstrukte definiert, die festlegen, wie die Klassen zu Leistungsmerkmalen gruppiert sind [24].

Die Konstrukte `feature` und `feature_end` klammern jeweils ein Leistungsmerkmal ein. Da wir jedem Leistungsmerkmal ein eigenes Kapitel widmen, lassen wir das Schlüsselwort `feature` in der Kapitelüberschrift erscheinen, z.B. für das neunte Kapitel: „9 `feature` BlackListOfUsers“. Dieses Schlüsselwort bewirkt gleichzeitig die formale Definition des Namens des Leistungsmerkmals. Der automatische Generator wird daraufhin entweder das gesamte Kapitel in die konkrete Anforderungsdefinition einschließen oder es insgesamt weglassen. So wird auch gleichzeitig der natürlichsprachliche Text, der diesen Teil der formalen Spezifikation in Bezug zur Realität setzt, geeignet ein- oder ausgeschlossen.

Ein weiterer Grund für die Unterscheidung von Klassen und Leistungsmerkmalen ist, daß neue Leistungsmerkmale oft *nicht-monotone Erweiterungen* sind. Dies bedeutet, daß in den Anforderungen für ein neues Leistungsmerkmal eine oder mehrere Eigenschaften des bisherigen Systems nicht mehr enthalten sind. Ein Beispiel haben wir bereits gesehen: Im Basis-Telefonsystem gibt es eine statische Eins-zu-Eins-Zuordnung zwischen Menschen und Geräten. Für das Leistungsmerkmal der Anrufweiterleitung müssen wir diese Eigenschaft entfernen und durch eine weniger restriktive und außerdem dynamische Zuordnung ersetzen.

Da wir beim Spezifizieren jeweils nur eine in sich abgeschlossene Eigenschaft in einer CSP-OZ-Klasse formalisieren, muß ein neues Leistungsmerkmal also ggf. eine oder mehrere solcher Klassen der bisherigen Spezifikation „entfernen". Dies realisieren wir wiederum mit Hilfe unserer Generatorfunktion. Außer CSP-OZ-Klassen kann ein Leistungsmerkmal-Kapitel auch `remove`-Anweisungen enthalten. Diese geben an, welche Klassen ersatzlos entfallen müssen, wenn dieses Leistungsmerkmal verwendet wird. Im generierten Dokument werden sie nicht enthalten sein, auch wenn der Rest des Leistungsmerkmals, zu dem sie gehören, eingeschlossen wird.

Durch diese Ausdrucksmöglichkeit droht selbstverständlich eine große Gefahr: Unerwünschte Feature-Interaktionen. Ein zweites Leistungsmerkmal kann ein erstes Leistungsmerkmal verändern, und zwar in einer Weise, die der Spezifizierer des ersten Leistungsmerkmals nicht vorhersehen konnte. Schlimmer noch, der Spezifizierer des zweiten Leistungsmerkmals wird das „alte" erste Leistungsmerkmal weniger gut kennen und die Folgen einer Veränderung weniger gut abschätzen können.

Aus diesem Grunde verlangen wir bereits vom ursprünglichen Spezifizierer des ersten Leistungsmerkmals, daß er explizit dokumentiert, was zum unverzichtbaren Kern dieses Leistungsmerkmals gehört, und welche weiteren Eigenschaften er nur hinzugefügt hat, um die Beschreibung vollständig und deterministisch zu machen. Auf diese Weise können insbesondere auch Punkte explizit dokumentiert werden, die eine Schnittstelle für spätere Erweiterungen darstellen. Formal tun wir dies alles, indem wir alle CSP-OZ-Klassen eines Leistungsmerkmals in *änderbare* und in *essentielle* Eigenschaften einteilen.

Syntaktisch geschieht dies durch zwei verschiedene Vererbungsoperatoren. Am Ende eines jeden Leistungsmerkmal-Kapitels definieren wir zwei besondere CSP-OZ-Klassen. In der einen sammeln wir per Vererbung alle Anforderungen an das System, und in der anderen ebenfalls per Vererbung alle Anforderungen an die Umwelt des Systems, auf die sich der Implementierer verlassen darf. Zum Erben der essentiellen Eigenschaften verwenden wir den gewöhnlichen CSP-OZ-Vererbungsoperator `inherit`. Zum Erben der änderbaren Eigenschaften nehmen wir dagegen den neuen Operator `default_properties`.

Der Generator ersetzt `default_properties` einfach syntaktisch durch `inherit`, so daß die beiden Operatoren auf CSP-OZ-Ebene die gleiche Semantik haben. Allerdings unterdrückt die Generatorfunktion in der Liste der mit `default_properties` geerbten Klassennamen alle diejenigen, die an anderer Stelle von einer `remove`-Anweisung eines eingeschlossenen Leistungsmerkmals entfernt werden. So gehen diese „entfernten" Klassen nicht mehr in die Gesamtanforderungen ein.

Im Rest dieses Kapitels machen wir unseren Ansatz durch ein Beispiel anschaulicher. In unserer Fallstudie [21] haben wir zunächst ein einfaches Basis-Telefonsystem spezifiziert. Da auch das Basis-System mehrere unabhängige Aspekte umfaßt, haben wir es in drei „Leistungsmerkmale" aufgeteilt. Dabei definiert das Leistungsmerkmal UserSpace die grundlegenden Begriffe des Menschen, der Benutzer(-Rolle) und des Gerätes. Alle Eigenschaften aus Kapitel 3.1 gehören dazu, außer der letzten. Das Leistungsmerkmal BasicConnection liefert den grundlegenden Begriff der Verbindung zwischen Benutzern, und wie eine solche auf- und abgebaut werden kann. Die letzte Eigenschaft aus Kapitel 3.1 gehört dazu. Das Leistungsmerkmal VoiceChannel liefert schließlich den Begriff eines Sprachkanals mit zwei Enden, der auf dem Begriff der Verbindung aufbaut.

Aus diesen drei Leistungsmerkmalen können wir eine erste Anforderungsspezifikation SimpleSpecification generieren. Der Generator erzeugt ein entsprechendes Dokument und hängt ein Kapitel an, das die Eigenschaften formal komponiert:

```
SimpleSpecificationSys
inherit
    UserSpaceFeatureSys
    BasicConnectionFeatureSys
    VoiceChannelFeatureSys
```

```
SimpleSpecificationEnv
inherit
    UserSpaceFeatureEnv
    BasicConnectionFeatureEnv
    VoiceChannelFeatureEnv
```

```
SimpleSpecification
inherit-interface SimpleSpecificationEnv
 main  =  SimpleSpecificationEnv || SimpleSpecificationSys
```

Anschließend erweitern wir unser Dokument um eine Liste unerwünschter Anrufer. Für das Leistungsmerkmal BlackListOfUsers hängen wir entsprechend ein Kapitel an. Allerdings erkennen wir, daß auch andere Formen von Schwarzen Listen interessant werden könnten, daher teilen wir es in zwei Kapitel auf. Das erste Kapitel ScreeningBase führt die Idee der Schwarzen Liste ein, ohne daß es das Verhalten des Basissystems verändert, während das zweite Kapitel BlackListOfUsers die erste Ausprägung der Schwarzen Liste beschreibt. Die erste Eigenschaft aus Kapitel 3.2 dieses Aufsatzes gehört zum Kapitel ScreeningBase, der Rest dort zum Kapitel BlackListOfUsers.

Ein weiteres Kapitel umfaßt das Leistungsmerkmal zur Anrufweiterschaltung FollowHumanConnectionForwarding. In diesem Aufsatz wird es in Kapitel 3.3 skizziert. Die `remove`-Anweisung dort gehört ebenfalls in dieses Kapitel. Schließlich generieren wie eine neue Anforderungsspezifikation SpecificationC, die auch die beiden neuen Leistungsmerkmale umfaßt. Sie ist am Ende von Kapitel 3.3 skizziert.

Es gibt noch weitere Leistungsmerkmale, die wir hier nicht mehr vorstellen können. So beschreibt das Leistungsmerkmal BlackListOfDevices eine andere Art von Schwarzer Liste, die nicht Benutzer abwehrt, sondern nur Anrufe von bestimmten Geräten, und die daher leichter zu implementieren ist. Das Leistungsmerkmal TransferUserRoleToAnotherHuman erlaubt eine andere Art der Anrufweiterschaltung. Hier geht es nicht darum, denselben Menschen an einem anderen Gerät zu erreichen, sondern eine Benutzerrolle an einen anderen Menschen zu delegieren, wie es z.B. bei dem Durchverbinden von einem Sachbearbeiter einer Firma zu einem anderen geschieht. Aus einer solchen Menge von Leistungsmerkmalen können wir uns jeweils die vom Kunden gewünschten heraussuchen und eine maßgeschneiderte Anforderungsspezifikation generieren lassen.

5 Automatische Erkennung von Feature-Interaktionen

Wie wir gesehen haben, hilft das hierarchische Schichten von Leistungsmerkmalen, Feature-Interaktionen zu vermeiden. Trotzdem können Spezifizierer noch Fehler machen. Bei deren Erkennung helfen die syntaktischen Regeln unserer neuen Konstrukte zur Spezifikation von Familien. Diese syntaktischen Regeln werden durch unser Generator-Werkzeug genFamMem überprüft.

Die erste dieser Typ-Prüfungen nutzt aus, daß wir die Abhängigkeiten zwischen den Teilen der Anforderungsspezifikation als Vererbung von Klassen explizit gemacht haben. In der generierten Spezifikation fehlen viele Klassen, weil entweder ihr Leistungsmerkmal nicht selektiert wurde, oder weil sie explizit von einem anderen selektierten Leistungsmerkmal per `remove`-Anweisung entfernt wurden. Auf diese Weise fehlende Klassen werden nicht implizit wieder eingeschlossen, sondern als Fehler gemeldet.

Eine Klasse kann mit einem von zwei Vererbungsoperatoren in eine zweite Klasse vererbt werden. Wenn sie dabei durch den `inherit`-Operator als essentiell gekennzeichnet wird, dann ist es ein syntaktischer Fehler, falls es irgendwo eine `remove`-Anweisung gibt, die sie entfernt. Und auch wenn eine Klasse niemals entfernt wird, gibt es bereits eine Warnung, wenn sie an verschiedenen Stellen mit verschiedenen der beiden Vererbungsoperatoren geerbt wird, weil sie so nicht eindeutig als essentiell oder änderbar gekennzeichnet ist.

Eine weitere Hilfe für die Wartung der Spezifikation ist die Option des Generators, die vollständige Hierarchie der Vererbung von Klassen auszugeben. Der Spezifizierer kann sich so einen Überblick über die Abhängigkeiten der Klassen und Leistungsmerkmale verschaffen.

Schließlich gibt es noch syntaktisch zulässige Situationen, die aber heuristisch als wahrscheinliche Fehler zu Warnungen führen: Es ist zulässig, daß zwei verschiedene Leistungsmerkmale mit je einer `remove`-Anweisung die gleiche Klasse entfernen. Aber es ist wahrscheinlich, daß jedes dieser Leistungsmerkmale eine Ersatzklasse definiert. Und diese können leicht zu einer widersprüchlichen Spezifikation führen, zu der es keine Implementation gibt. Ein Beispiel kommt in unserer Fallstudie zwischen den beiden verschiedenen Arten von Schwarzen Listen vor. Dort gibt es die Klasse BlackListUsrOnly, die wir oben in Kapitel 3.2 vorgestellt haben, und die im wesentlichen besagt, daß außer durch die Schwarze Liste unerwünschter Benutzer kein weiteres Screening vorkommt. Und es gibt eine ganz ähnliche Klasse für die Schwarze Liste unerwünschter Geräte. Aufgrund des Warnhinweises des Generators erkennt man schnell, daß es in der Tat zu Widersprüchen kommt, sobald Einträge in die Listen gemacht werden. Die Auflösung ist dann einfach: Wir definieren ein zusätzliches Leistungsmerkmal, das nur genau diesen Fall behandelt, indem es die beiden fraglichen Klassen entfernt und durch eine passende gemeinsame Klasse ersetzt. Da wir jeweils immer nur eine Eigenschaft pro Klasse spezifiziert haben, ist der Aufwand dafür nicht groß.

6 Zusammenfassung und Diskussion

Wir beschäftigen uns mit der Wartung von formalen Anforderungsdokumenten. In unserem Anwendungsgebiet der Telefonvermittlungssysteme sind unerwünschte Feature-Interaktionen derzeit ein ernsthaftes Problem, die bei der inkrementellen Erweiterung der Systeme entstehen. Die vielen entstehenden Varianten behandeln wir als eine *Familie von Anforderungen* gemeinsam. Für die Formale Beschreibungstechnik CSP-OZ haben wir dazu eine Spracherweiterung definiert, mit deren Hilfe eine Generatorfunktion aus einem einzigen zentralen Anforderungsdokument jeweils die gewünschte Konfiguration von Anforderungen extrahieren kann. Zur Vermeidung von Feature-Interaktions-Problemen verwenden wir außerdem eine *hierarchische*, geschichtete Architektur von Leistungsmerkmalen, im Gegensatz zum derzeit verwendeten Intelligenten Netzwerk.

Die Abhängigkeiten von Eigenschaftsdefinitionen werden dabei explizit in der Hierarchie der Vererbungsrelation zwischen CSP-OZ-Klassen dokumentiert. Ebenso wird explizit zwischen änderbaren und essentiellen Eigenschaften unterschieden. Durch diese explizit dokumentierte Information wird die Erkennung von (bestimmten) Feature-Interaktions-Problemen durch eine Typüberprüfung möglich.

Es gibt in der Literatur viele Ansätze für formale Verfeinerungen, die gewünschte Eigenschaften eines Systems erhalten und damit Feature-Interaktionen verhindern. Zum Beispiel erlauben manche nur, Traces zu einem System hinzuzufügen, ohne alte zu entfernen, während andere umgekehrt nur erlauben, Nichtdeterminismus weiter einzuschränken. Leider sind diese Einschränkungen für unser Anwendungsgebiet oft zu hart, da ein neues Leistungsmerkmal dort meist eine „nicht-monotone" Erweiterung ist. Daher lassen wir sowohl Verhaltenseinschränkungen als auch neues Verhalten zu, stellen aber sicher, daß die explizit als „essentiell" markierten (Invarianz-)Eigenschaften erhalten bleiben.

Zur Zeit arbeiten wir daran, unsere Fallstudie zu erweitern, um noch mehr praktische Erfahrungen mit dem Ansatz zu sammeln. Außerdem arbeiten wir an der Verbesserung der Werkzeugunterstützung.

Literatur

[1] Kristofer Kimbler und L. G. Bouma (Herausgeber). „Feature Interactions in Telecommunications and Software Systems V". IOS Press, Amsterdam (September 1998).

[2] Petre Dini, Raouf Boutaba und Luigi Logrippo (Herausgeber). „Feature Interactions in Telecommunication Networks IV". IOS Press, Amsterdam (Juni 1997).

[3] Kong Eng Cheng und Tadashi Ohta (Herausgeber). „Feature Interactions in Telecommunications III". IOS Press, Amsterdam (1995).

[4] L. G. Bouma und Hugo Velthuijsen (Herausgeber). „Feature Interactions in Telecommunications Systems“. IOS Press, Amsterdam (1994).

[5] IEEE Communications Society. „1st International Workshop on Feature Interactions in Telecommunications Software Systems“, St. Petersburg, Florida, USA (Dezember 1992).

[6] Pamela Zave. Secrets of call forwarding: A specification case study. In Gregor von Bochmann, Rachida Dssouli und Omar Rafiq (Herausgeber), „Formal Description Techniques VIII“. Chapman & Hall (1996).

[7] Clemens Fischer. CSP-OZ: a combination of Object-Z and CSP. In Howard Bowman und John Derrick (Herausgeber), „Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)“, Band 2, Seiten 423–438. Chapman & Hall (Juli 1997).

[8] Clemens Fischer. Combining Object-Z and CSP. In Adam Wolisz, Ina Schieferdecker und Axel Rennoch (Herausgeber), „Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch“, GMD-Studien Nr. 315, Seiten 119–128. GMD, St. Augustin, Deutschland, Berlin (19.–20. Juni 1997).

[9] Clemens Fischer. „Combination and Implementation of Processes and Data: from CSP-OZ to Java“. Dissertation, Universität Oldenburg, Oldenburg (April 2000). *In Vorbereitung.*

[10] C. A. R. Hoare. „Communicating Sequential Processes“. Prentice-Hall (1985).

[11] A. W. Roscoe. „The Theory and Practice of Concurrency“. Prentice-Hall (1997).

[12] Graeme Smith. „The Object-Z Specification Language“. Kluwer Academic Publishers (2000).

[13] Jim Woodcock und Jim Davies. „Using Z – Specification, Refinement, and Proof“. Prentice-Hall (1996).

[14] John Michael Spivey. „The Z notation: a reference manual“. Series in Computer Science. Prentice-Hall, New York, 2. Auflage (1995).

[15] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe (Herausgeber), „A Classical Mind, Essays in Honour of C.A.R. Hoare“, Seiten 353–378. Prentice-Hall (1994).

[16] Clemens Fischer und Heike Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway und K. Taguchi (Herausgeber), „Proc. of the 1st Int'l. Conf. on Integrated Formal Methods – IFM'99“. Springer (1999).

[17] Boris Wirtz. Übersetzung des Daten- und Verhalten-integrierenden Spezifikationsformalismus CSP-OZ nach CSP. Diplomarbeit, Universität Oldenburg, FB Informatik (2000). *In Vorbereitung.*

[18] Jens von Garrel. Parsing, Typechecking und Transformation von CSP-OZ nach Jass. Diplomarbeit, Universität Oldenburg, FB Informatik (Juli 1999).

[19] ITU-T. „Q.12xx-Series Intelligent Network Recommendations“ (1997).

[20] James J. Garrahan, Peter A. Russo, Kenichi Kitami und Roberto Kung. Intelligent Network overview. *IEEE Commun. Mag.* **31**(3), 30–36 (März 1993).

[21] Jan Bredereke. Modular, changeable requirements for telephone switching in CSP-OZ – revision 2.0. Int. Ber. IBS-00-1, Universität Oldenburg, Oldenburg (2000). *In Vorbereitung.*

[22] Jan Bredereke. Modular, changeable requirements for telephone switching in CSP-OZ. Int. Ber. IBS-99-1, Universität Oldenburg, Oldenburg (Oktober 1999).

[23] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**(1), 1–9 (März 1976).

[24] Jan Bredereke. Families of formal requirements in telephone switching. In „Feature Interactions in Telecommunications and Software Systems VI“, Amsterdam (Mai 2000). IOS Press. *In Vorbereitung.*