Ein Werkzeug zum Generieren von Spezifikationen aus einer Familie formaler Anforderungen

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

Zusammenfassung

Wir beschäftigen uns mit der Wartung von formalen Anforderungsdokumenten im Anwendungsgebiet der Telefonvermittlungssysteme. Wir schlagen eine Spezifikationsmethode vor, die einige der so genannten Feature-Interaktions-Probleme von Anfang an vermeidet, und die einige weitere dieser Probleme in Typfehler verwandelt. Wir behandeln alle Varianten und Versionen eines solchen Systems gemeinsam als eine Familie von formalen Anforderungen. Dafür präsentieren wir einen formalen Feature-Kombinations-Mechanismus. Insbesondere präsentieren wir ein Werkzeug, das Feature-Interaktions-Probleme anzeigt, das ein gewünschtes Familienmitglied aus dem Familiendokument extrahiert, und das Dokumentation über die Struktur der Familie generiert.

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 [BGK00, CaMa00, KiBo98]. 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) (Abb. 1). 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

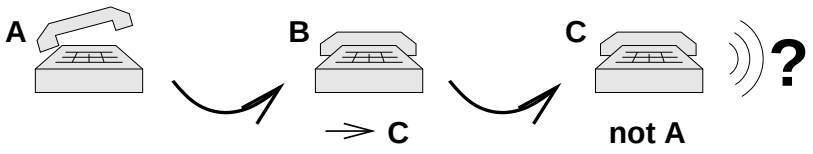

Abbildung 1: Feature-Interaktion zwischen Terminating Call Screening und Call Forwarding.

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.

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.

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 [Bre00d, Bre00a] haben wir die Idee der *Familien von formalen Anforderungen* eingeführt. Unser Ansatz schlägt vor, alle Anforderungsvarianten zusammenzuhalten und die Auswahl von Varianten einzukapseln. Er ermöglicht die Verwendung des Geheimnisprinzips (Information-Hiding) für Anforderungsdokumente, ähnlich wie es für Programme bekannt ist. Weiterhin ermutigt er, für zukünftige Mitglieder der Familie vorauszuplanen. Eine weitere Kernidee ist, den ursprünglichen Spezifizierer so viele Informationen wie möglich explizit dokumentieren zu lassen, die ein anderer Spezifizierer eines inkrementellen Leistungsmerkmals benötigen wird. Der Spezifizierer eines inkrementellen Leistungsmerkmals kann diese explizite Information entweder benutzen, um Feature-Interaktions-Probleme zu vermeiden, oder um sie zumindest zu finden, so daß sie behoben werden können.

Eine Familie von Anforderungen benötigt ein Ausdrucksmittel für inkrementelles Spezifizieren. Ein übliches Ausdrucksmittel dafür ist Verfeinerung. In Verfeinerungsansätzen spezifiziert man zuerst ein Basissystem, das dann Schritt für Schritt erweitert wird, indem zusätzliche Teile spezifiziert werden. Die Formalismen sind so aufgebaut, daß die interessanten Eigenschaften früherer Schritte von nachfolgenden Schritten erhalten werden. Zum Beispiel erlauben es einige Ansätze, zusätzliches mögliches Verhalten zu einer Verhaltensbeschreibung hinzuzufügen, wobei alles bereits vorhandene Verhalten weiterhin möglich bleibt. Andere Ansätze erlauben es, inkrementelle, explizite Einschränkungen für eine (implizite) Verhaltensbeschreibung zu machen, wobei alle früheren Einschränkungen aufgrund der Natur des logischen „Und" erhalten bleiben. Diese formalen Mechanismen sind für viele Anwendungen nützlich, aber sie reichen in unserem Anwendungsgebiet der Telefonvermittlungssysteme nicht aus. Nach unserer Erfahrung mit Anforderungen für Telefonvermittlungssysteme *ändern die meisten neuen Leistungsmerkmale das Verhalten des Basissystems oder anderer Leistungsmerkmale.* „Ändern" bedeutet hier, daß etwas wirklich anders wird als es war. Zum Beispiel unterbindet das Call-Forwarding-Leistungsmerkmal die Verbindung zur gewählten Nummer und schränkt so das Verhalten des Basissystems ein, und es fängt an, Verbindungen zur Weiterleitungsnummer aufzubauen, womit es das Verhalten des Basissystems erweitert. Ein Telefon-Leistungsmerkmal ist typischerweise *nicht-monoton.*

Eine Reihe von Feature-Operatoren sind vorgeschlagen worden, um nicht-monotone Änderungen zuzulassen. Sie erlauben es, die gewünschten Änderungen auszudrücken. So-

weit ihre Definition und Anwendung die gegenwärtige Praxis der Modellierung von Telefon-Leistungsmerkmalen widerspiegelt, treten Feature-Interaktions-Probleme in der Formalisierung auf die gleiche Weise auf wie im nicht formalisierten Fall. Es gibt umfangreiche Forschungen über die formale Analyse der Änderungen [BGK00, CaMa00, KiBo98, Bre97]. Nach unserer Erfahrung funktioniert sie in der Tat bis zu einem gewissen Maße, aber das exponentielle Wachstum der Komplexität überwältigt schließlich nicht nur den menschlichen Spezifizierer, sondern auch die automatischen Werkzeuge, sobald es um Systeme realer Größe mit hunderten von Leistungsmerkmalen geht. Insbesondere führen Feature-Operatoren, die beliebige syntaktische Modifikationen an einzelnen Quellcode-Zeilen erlauben, zu unüberwindlichen Feature-Interaktions-Problemen, weil das gesamte System auf die Konsequenzen dieser Modifikationen hin untersucht werden muß.

Wir versuchen deshalb, die Komplexität zu reduzieren, indem wir die Anforderungsdokumente im Information-Hiding-Sinne modularisieren. Wir unterstützen das durch ein Feature-Konstrukt, das zwar nicht-monotone Änderungen zuläßt, nicht aber beliebige Änderungen. Nach unserer Erfahrung reduziert Vorausplanen für eine sorgfältige Modularisierung und für zukünftige potentielle Modifikationen die Anzahl und Komplexität der Abhängigkeiten zwischen Features.

In Kapitel 2 geben wir zunächst einen kurzen Überblick über die von uns verwendete Formale Beschreibungstechnik, und wir präsentieren ein geeignetes, neues Feature-Konstrukt für sie. Kapitel 3 beschreibt unsere Werkzeugunterstützung für die Verwaltung von Features. Kapitel 4 faßt die Ergebnisse zusammen.

2 Eine Spezifikationsmethode zur Unterstützung von Familien von formalen Anforderungen

In diesem Abschnitt geben wir einen kurzen Überblick über die von uns benutzte formale Beschreibungstechnik und konzentrieren uns dabei auf die Ausdrucksmittel für die Strukturierung eines Spezifikationsdokumentes. Eine weitere Voraussetzung ist ein geeigneter Spezifikationsstil, den wir beschreiben. Wir fügen dann in einem ersten Schritt das Konzept der Familie zu der Sprache hinzu, und in einem zweiten Schritt das Konzept der nichtmonotonen, aber wohlstrukturierten Änderung.

2.1 Basissprache CSP-OZ

Wir haben die formale Beschreibungstechnik CSP-OZ [Fis00, Fis97] als Basis für unsere neuen Feature-Konstrukte gewählt. Diese Sprache ist für den Bereich der Telefonvermittlung geeignet, da sie Kommunikationsereignisse explizit unterstützt, und sie ist zum Spezifizieren von Anforderungen geeignet, da sie erlaubt, die äußerlich sichtbaren Kommunikationsereignisse zu spezifizieren, ohne irgendeine innere Struktur vorzuschreiben. CSP-OZ ist eine Kombination der Prozeßalgebra CSP (Communicating Sequential Processes) [Ros97] mit Object-Z [Smi00]. CSP-OZ erlaubt es, Kommunikationsaspekte im CSP-Teil zu spezifizieren und Datenaspekte getrennt davon im Object-Z-Teil. Der Object-Z-Datenteil wiederum basiert auf der Sprache Z [ISO99, Spi95], die eine Formalisierung der Mengenlehre ist. Object-Z bietet die Konzepte der abstrakten Datentypen und der Zustandsinvarianten, die es in CSP nicht gibt. Zahlreiche algebraische Gesetze sind für die Verhaltensverfeinerung in CSP und für die Datenverfeinerung in Z abgeleitet worden, sie bieten eine solide theoretische Grundlage für die von uns benutzten Vererbungsoperatoren.

Der Vererbungsoperator von CSP-OZ erlaubt, in seiner Grundform, Verfeinerung zu spezifizieren. Optional kann er durch einen Umbenennungsoperator ergänzt werden. Wenn letzterer benutzt wird, erlaubt er, das Verhalten einer geerbten Klasse vollständig zu ändern. Die Vererbung in CSP-OZ bietet daher entweder reine Verfeinerung oder mächtige, aber unkontrollierte, Änderungsmöglichkeiten.

Die Sprache Z wurde über viele Jahre durch Spiveys Referenz-Manual [Spi95] definiert. Gegenwärtig beschließt die International Standardization Organization (ISO) einen Standard für Z [ISO99]. Außer einer detaillierteren, wohlstrukturierten Definition der Syntax und Semantik von Z bietet der Standard auch einige wenige Erweiterungen zu Spiveys Z. Eine dieser Erweiterungen betrifft die Strukturierung eines Spezifikationsdokumentes, und sie unterstützt unsere Arbeit erheblich. Wir legen daher für unsere Arbeit bereits den in Vorbereitung befindlichen Z-Standard zugrunde. Zum Glück baut die Definition von CSP-OZ ebenfalls bereits auf dem neuen Z-Standard auf.

Eine Spezifikation in Z (und daher auch eine in CSP-OZ) besteht aus Absätzen („paragraphs") formalen Textes. Ein Absatz kann z.B. eine Typdefinition oder eine Schemadefinition sein. Der neue Standard erlaubt, eine *Spezifikation* („specification") und *Abschnitte* („section") über die Absätze zu setzen. Eine Spezifikation besteht aus Abschnitten, die wiederum aus Absätzen bestehen. Die Bedeutung einer Spezifikation in Z ist die Menge benannter Theorien, die durch ihre Abschnitte gegeben ist. Jeder Abschnitt fügt eine benannte Theorie zu der Menge hinzu, die durch die vorigen Abschnitte gegeben ist. Eine benannte Theorie ordnet einem Abschnittsnamen eine Menge von Modellen zu.

Damit folgt, daß jeder Abschnitt eine in sich abgeschlossene formale Bedeutung hat. Jede initiale Folge von Abschnitten kann man als die Anforderungen für eine Variante des spezifizierten Systems ansehen. Da jeder weitere Abschnitt dem System nur weitere Einschränkungen (und neue Deklarationen) hinzufügen kann, erlaubt dies einen constraint-orientierten, inkrementellen Spezifikationsstil.

Allerdings gibt es keine Ausdrucksmittel für nicht-monotone Erweiterungen. Die Abschnitte in Z haben ein *parents*-Konstrukt, das erlaubt, einen Abschnitt als Erweiterung von explizit genannten Abschnitten zu spezifizieren. Die einzige Möglichkeit, eine modifizierte Version eines Systems zu bekommen, ist, die zu ändernden Abschnitte zu kopieren, umzubenennen und zu editieren. Dabei müssen auch alle Abschnitte kopiert und umbenannt werden, die von diesen Abschnitten mit Hilfe des parents-Konstruktes abgeleitet worden sind. Falls ein Abschnitt geändert werden muß, der tief unten in der parents-Hierarchie sitzt, kann dies zur Folge haben, daß ein großer Teil des Dokumentes dupliziert wird.

2.2 Spezifikationsstil

Für Anforderungen bevorzugen wir einen constraint-orientierten Spezifikationsstil. Das Hinzufügen einer (kleinen) Einschränkung nach der anderen hilft uns, uns auf einen Aspekt des Systems zur Zeit zu konzentrieren, ohne dabei aus Versehen zu restriktiv in Bezug auf andere Aspekte zu sein.

Wenn wir nicht-monoton erweitern, ist es wichtig, daß die Änderungen auf eine kontrollierte Art und Weise geschehen. Daher legen wir fest, daß ein Feature ein anderes Feature auf der Ebene von Z-Abschnitten modifizieren darf. Es wäre zu grobkörnig, ein ganzes Feature zu entfernen und gleichzeitig ein neues, gleichartiges Feature mit einer kleinen Modifikation dazuzufügen. Wenn das Feature komplex ist, resultiert dies in einer beträchtlichen Verdoppelung von Code. Es wäre zu feinkörnig, einem Feature zu erlauben, den Spezifikationstext eines anderen Features auf eine beliebige Art und Weise zu modifizieren, z.B. durch Ändern

eines Zahlenwertes oder durch Ersetzen eines arithmetischen Operators. Die Konsequenzen solcher Änderungen sind schwer nachzuvollziehen, insbesondere wenn viele davon zur gleichen Zeit durchgeführt werden. Dies führt leicht zu Feature-Interaktions-Problemen.

Wir unterscheiden zwischen dem *essentiellen Verhalten* und dem *änderbaren Verhalten* eines Features. Einige Teile eines Features sind essentiell für sein Wesen. Andere Teile werden nur benötigt, um die Anforderungsspezifikation vollständig zu machen. Zum Beispiel könnten einige Verhaltenseinschränkungen nur gemacht worden sein, damit das Verhalten für den Benutzer vorhersagbar ist. Falls das änderbare Verhalten modifiziert wird, kann dies niemals zu einer unerwünschten Feature-Interaktion führen. Wenn ein Spezifizierer ein Feature modifizieren muß, das vor längerer Zeit von einer anderen Person geschrieben wurde, dann wird er Schwierigkeiten haben herauszufinden, welcher Teil des Verhaltens essentiell ist. Daher verlangen wir vom ursprünglichen Spezifizierer, daß er dokumentiert, welches Verhalten essentiell und welches Verhalten änderbar ist, indem er ein entsprechendes Sprachkonstrukt verwendet.

Weiterhin unterscheiden wir zwischen den Anforderungen an das System und den Anforderungen an seine Umgebung. Dies ist für jeden Softwareentwicklungsvertrag wichtig. Es ist die Pflicht des Entwicklers, die ersteren Anforderungen zu implementieren, und es die Pflicht des Kunden, die Erfülltheit der letzeren sicherzustellen, damit das System auch funktionieren kann. Formal sammeln wir beide Teile mittels Vererbung in je eine CSP-OZ-Klasse. Wir vereinigen diese dann durch Parallelkomposition zu einer Beschreibung aller Aspekte der Welt, die für den Vertrag relevant sind.

2.3 Erweiterung von CSP-OZ: Familien mit ausschließlich monotonen Inkrementen

Um Familien einzuführen, ändern wir die Grobstruktur der Sprache. Wir entfernen Spezifikationen aus der Syntax und fügen stattdessen *Familien*, *Feature-Kapitel* und *Familienmitglied-Kapitel* hinzu.

Eine Familie besteht aus Feature-Kapiteln und aus Familienmitglied-Kapiteln. Familienmitglied-Kapitel sind benannte Kapitel, die nichts als eine Liste mit Feature-Namen enthalten. Informell definiert jedes Familienmitglied-Kapitel eine Spezifikation in einfachem CSP-OZ. Wir können ein Familienmitglied aus einem Familiendokument extrahieren, indem wir alle seine Features nehmen und alle Abschnitte dieser Features hintereinander hängen, so daß wir eine Spezifikation in einfachem CSP-OZ erhalten.

Eine detaillierte Definition der Syntax und Semantik der Spracherweiterung findet sich im Manual [Bre00c].

Der Vererbungsoperator des einfachen CSP-OZ kann durch einen Umbenennungsoperator ergänzt werden, der erlaubt, das Verhalten einer geerbten Klasse vollständig zu ändern. Wir verbieten daher die Verwendung dieser Umbenennungsoption in unserer Erweiterung von CSP-OZ.

2.3.1 Semantik

Die Menge *Features* enthält die Namen aller Feature-Kapitel. Die Menge *Sections* enthält alle Abbildungen von Feature-Kapitel-Namen auf Mengen von Abschnitts-Namen: $Sections == Features \pfun \mathbb{P}\, Name$. Die Bedeutung eines Feature-Kapitels kann man bestimmen, indem man die Funktion *Sections* auf den Feature-Kapitel-Namen anwendet. Die Bedeutung einer Familie ist eine Abbildung von Familienmitglied-Namen auf Bedeutungen

von einfachen CSP-OZ-Spezifikationen.

$$[\![\cdot]\!]^{\mathcal{F}} \quad : \quad \mathit{Family} \pfun (\mathit{Name} \pfun \mathbb{P}\, \mathit{Theory})$$

Das heißt, daß jedes (reine) Familienmitglied-Kapitel im wesentlichen eine vollständige CSP-OZ-Spezifikation ist, außer daß alle ihrer Abschnitte in Feature-Kapitel verschoben worden sind, die mit anderen Familienmitgliedern geteilt werden können.

2.3.2 Abkürzende Notationen

Optional kann ein Familienmitglied-Kapitel einen Feature-Kapitel-Rumpf enthalten. Dies wird syntaktisch in ein eigenes Feature-Kapitel und ein „reines" Familienmitglied-Kapitel transformiert. Solch ein Feature-Kapitel-Rumpf kann nützlich sein, um die Komposition aller Bestandteile aus den verschiedenen Features geeignet zu spezifizieren.

CSP-OZ-Klassen sind größere Einheiten als schlichte (Z-)Absätze. Daher fanden wir es gelegentlich hilfreich, Klassen zur Strukturierung zu verwenden anstelle von Abschnitten. Dies gilt insbesondere dann, wenn jeder Abschnitt nur eine Klasse enthält. Daher erlauben wir auch eine Klasse an den Stellen, an denen ein Abschnitt stehen darf. Wenn wir die Semantik bestimmen, transformieren wir eine solche Klasse in eine Abschnittüberschrift gefolgt von dieser Klasse, und gefolgt von was auch immer an formalen Absätzen folgen mag. Der Name dieses impliziten Abschnitts ist der Name der Klasse. (Man beachte, daß Familienmitglieder, Features und Abschnitte jeweils eigene Namensräume besitzen.) Die Namen der parents-Abschnitte des Abschnitts sind die Namen der Klassen, die von dieser Klasse geerbt werden.

2.4 Erweiterung von CSP-OZ: Familien mit nicht-monotonen Inkrementen

Abschnitte sind unsere Größeneinheit für monotone Inkremente, und sie sind ebenfalls unsere Größeneinheit für nicht-monotone Änderungen. Wir unterscheiden zwischen dem essentiellen und dem änderbaren Verhalten eines Features, und wir spezifizieren diese Information explizit, indem wir verschiedene Arten von Abschnitt verwenden. Die Typregeln für die neuen Konstrukte, die wir weiter unten definieren, werden uns erlauben, diese Information für formale Prüfungen zu nutzen.

Eine einzelne weitere Art von Abschnitt ist allerdings noch nicht ausreichend. Die essentiellen und die änderbaren Verhaltens-Einschränkungen müssen zusammengefügt werden. Die Kompositionsoperatoren dafür müssen ebenfalls in Abschnitte gruppiert werden, und diese Operatoren spezifizieren weder essentielles noch änderbares Verhalten. Sie benötigen eine dritte Art von Abschnitt.

Wir führen daher zwei weitere Arten von Abschnitt ein. Wir schreiben die essentiellen Eigenschaften eines Features in die normale Art von Abschnitt. Die zwei neuen Arten von Abschnitt haben die gleiche Syntax wie die normale Art, außer daß wir die neuen Schlüsselwörter default_section und collect_section für sie verwenden. *Default-Abschnitte* („default_section") unterscheiden sich dadurch, daß sie von einem *remove-Konstrukt* referenziert werden können, und sie dienen dazu, änderbare Eigenschaften auszudrücken. Ein remove-Konstrukt ist ein weiteres neues Konstrukt, und es darf innerhalb von Feature-Kapiteln benutzt werden. Wenn das Familienmitglied ein Feature umfaßt, das ein remove-Konstrukt enthält, dann werden die in ihm genannten Default-Abschnitte beim Bestimmen der Menge der Abschnitte des Familienmitglieds ausgelassen. Ein *Sammel-Abschnitt* („collect_section") unterscheidet sich dadurch, daß er seine Liste von parents-Abschnitten

automatisch in Bezug auf verlorengegangene Default-Abschnitte durch remove-Konstrukte anpaßt. Ein Sammel-Abschnitt sollte nur verwendet werden, um in anderen Abschnitten definierte Eigenschaften zu sammeln und zusammenzufügen.

Wir definieren sechs *Typregeln* für diese verschiedenen Arten von Abschnitten. Sie stellen die beabsichtigte Einsatzweise der Abschnitte sicher, und sie zeigen Fehler auf, die Feature-Interaktions-Probleme darstellen können. Eine grundlegende Regel ist, daß nur änderbare Eigenschaften entfernt werden können: 1) Ein remove-Konstrukt darf nur Default-Abschnitte nennen. 2) Das parents-Konstrukt eines normalen, d.h. essentiellen Abschnitts darf keinen Default-Abschnitt nennen, außer wenn dieser zu einem anderen Feature oder Familienmitglied gehört.

Die obige Ausnahme ist notwendig, damit ein Feature eine bestimmte, nicht-essentielle Version eines anderen Features als Grundlage nehmen kann. Beachte: Ein Type-Checker-Werkzeug kann eine Warnung ausgeben, wenn es mehr als ein remove-Konstrukt für einen einzelnen Default-Abschnitt in den Features eines Familienmitgliedes gibt.

Weitere Regeln beschränken parents-Konstrukte und remove-Konstrukte auf Abschnitts-Namen, die sinnvoll sind in Hinsicht auf die einzelnen Familienmitglieder: 3) Für jedes Familienmitglied m konstruieren wir die Menge $SectionNames(m)$ mit den Namen aller normalen, Default- und Sammel-Abschnitte aller Features von m. Für jedes Familienmitglied m dürfen alle remove-Konstrukte seiner Features nur Abschnitte aus der Menge $SectionNames(m)$ nennen. 4) Wir konstruieren außerdem die reduzierte Menge $RSectionNames(m)$ aus $SectionNames(m)$, indem wir diejenigen Default-Abschnitte entfernen, die in einem remove-Konstrukt dieses Familienmitglieds genannt werden. Für jeden normalen oder Default-Abschnitt in $RSectionNames(m)$ muß die Menge seiner parents-Abschnitte eine Teilmenge von $RSectionNames(m)$ sein.

Schließlich darf die veränderliche Menge definierter Dinge, die in Sammel-Abschnitten stehen, nicht in eigenschaftsdefinierenden Abschnitten referenziert werden: 5) Weder normale noch Default-Abschnitte dürfen einen Sammel-Abschnitt in ihrem parents-Konstrukt nennen. 6) Die parents-Listen von Sammel-Abschnitten dürfen nur Abschnitte enthalten, die entweder zum selben Feature gehören oder die selbst Sammel-Abschnitte sind.

Wir führen auch eine neue Art von Vererbungsoperator ein, genannt „default_properties“, der ignoriert wird, falls die geerbte Klasse entfernt worden ist. Wir erweitern die Syntax entsprechend, und wir legen zwei weitere Typregeln für den neuen Vererbungsoperator fest: 7) Der default_properties-Operator darf nur in einem Sammel-Abschnitt verwendet werden. 8) Der Klassenname in dem Operator muß ein gültiger Klassenname im gesamten Familiendokument sein; für jedes einzelne Familienmitglied dürfen die default_properties-Operatoren aber Klassennamen von außerhalb seiner Abschnitte referenzieren.

Eine detaillierte Beschreibung der Syntax und Semantik dieses Teils der Spracherweiterung findet sich wiederum im Manual [Bre00c].

2.4.1 Semantik

Die Bedeutung eines remove-Konstruktes ist die Menge seiner Namen von Default-Abschnitten. Die Funktion *Remove* bildet Feature-Kapitel-Namen auf die Mengen von Namen von Default-Abschnitten ab, die in einem remove-Konstrukt des betreffenden Feature-Kapitels genannt sind: $Remove : Features \pfun \mathbb{P}\, Name$.

Die Features-Liste eines Familienmitglied-Kapitels bestimmt die Features, die verwendet werden, um seine CSP-OZ-Spezifikation zu konstruieren. Die Features-Liste ist eine Teilmenge der Menge *Features*. Ihre Bedeutung ist die Vereinigung aller Mengen der Abschnitts-

Namen der aufgeführten Features, abzüglich der Menge derjenigen Default-Abschnitte, die von den aufgeführten Features in einem remove-Konstrukt genannt werden.

$$\begin{aligned}[\![fe_1 \ldots fe_n]\!]^{\mathcal{FL}} &= (Sections(fe_1) \cup \ldots \cup Sections(fe_n)) \setminus \\ &\quad (Remove(fe_1) \cup \ldots \cup Remove(fe_n))\end{aligned}$$

Die Abschnitte der CSP-OZ-Spezifikation, die für ein Familienmitglied-Kapitel konstruiert wird, sind genau alle Abschnitte $[\![fe_1 \ldots fe_n]\!]^{\mathcal{FL}}$. Die einzige Modifikation ist, daß wir alle Abschnitts-Namen in $(Remove\ fe_1 \cup \ldots \cup Remove\ fe_n)$ aus den parents-Konstrukten der Abschnitte entfernen. Die Bedeutung der Spezifikation wird dann in der üblichen Art und Weise aus diesen Abschnitten konstruiert.

2.4.2 Abkürzende Notationen

Falls Klassen anstelle von Abschnitten zur Strukturierung benutzt werden, dann wird eine Klasse genau dann in einen Sammel-Abschnitt statt in einen normalen Abschnitt transformiert, wenn sie einen default_properties-Operator enthält. Eine Klasse wird genau dann in einen Default-Abschnitt transformiert, wenn ein default_properties-Operator irgendwo im selben Feature existiert, der ihren Namen referenziert.

Wir komponieren die Anforderungen an ein Familienmitglied, indem wir die Sammel-Abschnitte aller Features eines Familienmitglieds in einem einzigen Sammel-Abschnitt des Familienmitglieds zusammensetzen. Die Struktur dieses Sammel-Abschnitts folgt stets einem festen Muster: Eine Klasse erbt alle Anforderungen an das System, eine zweite Klasse erbt alle Anforderungen an dessen Umgebung und eine dritte, abschließende Klasse komponiert die beiden vorigen mit einem Parallelkompositionsoperator. Da dieser abschließende Sammel-Abschnitt eine solch feste Struktur hat, darf er weggelassen werden, die Semantikdefinition der Sprache leitet ihn dann aus der Features-Liste ab. In diesem Falle besteht das gesamte Familienmitglied-Kapitel nur aus der Features-Liste.

3 Das Werkzeug

Unser Ansatz wird durch das Werkzeug *genFamMem 2.0* unterstützt, das

- aus einem Familiendokument Spezifikationen in einfachem CSP-OZ extrahiert,
- Feature-Interaktionen erkennt, indem es
  - zusätzliche Typprüfungen für Familien durchführt, wie die Gültigkeit der Randbedingungen für normale, Default- und Sammel-Abschnitte, für das remove-Konstrukt und für die default_properties-/inherit-Operatoren,
  - heuristische Warnungen erzeugt, zum Beispiel wenn zwei verschiedene remove-Konstrukte dieselbe Klasse entfernen, und das
- Feature-Interaktionen vermeiden hilft, indem es Dokumentation über die Struktur der Familie generiert.

Das Werkzeug besteht aus einem Lexer/Parser für unsere Erweiterung von CSP-OZ, der mit dem lex/yacc Parser-Generator [LMB92] geschrieben wurde, und aus C-Code. Das Werkzeug umfaßt etwa 8500 Zeilen kommentierten Quellcode. Abbildung 2 zeigt seine Modulstruktur. Das Werkzeug hat eine modulare Struktur, die die wohlstrukturierte Definiti-

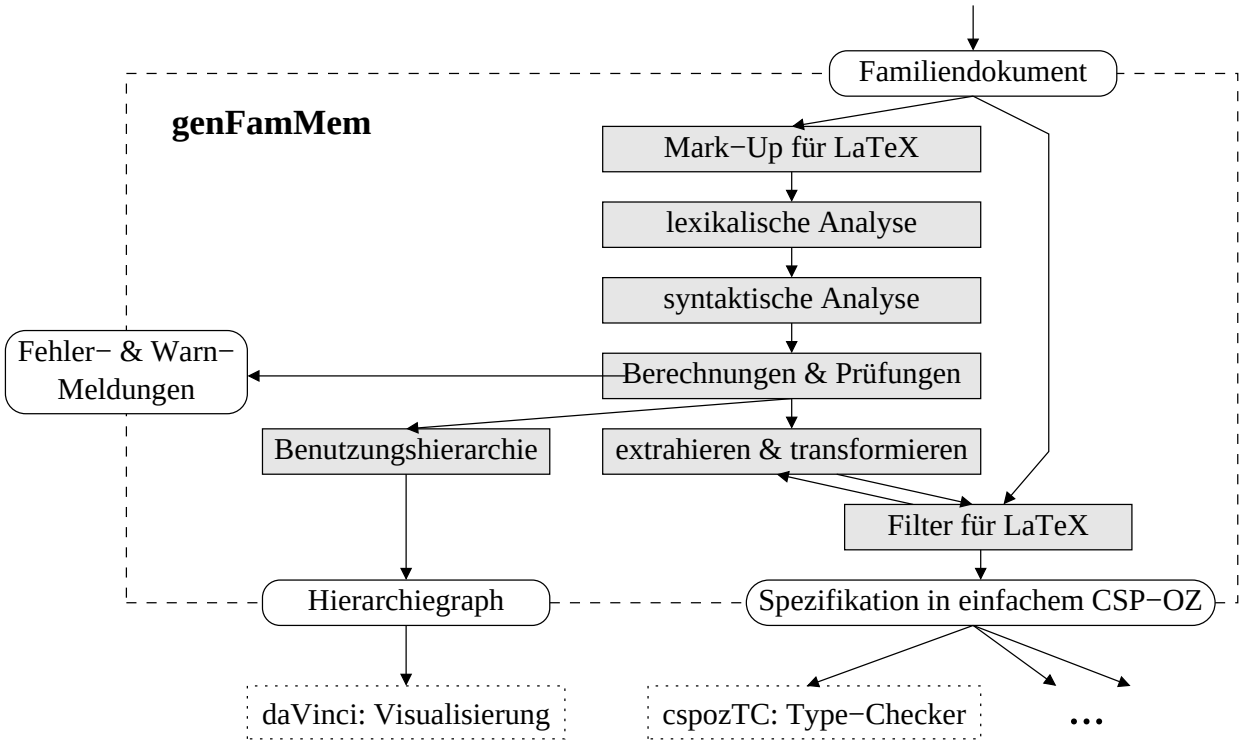

Abbildung 2: Struktur des genFamMem-Werkzeugs.

on der Semantik von Z im kommenden Standard widerspiegelt. Der Standard definiert die Syntax und Semantik in neun Phasen, deren erste Mark-Up, lexikalische und syntaktische Analyse sind und deren letzte die semantische Relation ist. Die getrennte Mark-Up-Phase löst das Problem, daß Z sehr viel mehr Sonderzeichen verwendet als eine normale Tastatur zur Verfügung stellt, indem die Sonderzeichen durch Folgen von Zeichen kodiert werden. Der Standard definiert zwei Mark-Ups, LaTeX-Mark-Up und Email-Mark-up.

In einem ersten Durchlauf nimmt das Mark-Up-Modul ein Familiendokument und übersetzt LaTeX-Befehle wie „`\fset`“, „`\pfun`“ usw. in CSP-OZ-Zeichen wie „$\mathbb{F}$“, „$\nrightarrow$“ usw. Es könnte durch Mark-Up-Module für andere Mark-Ups ersetzt werden. Die nächsten Schritte sind die lexikalische und die syntaktische Analyse. Das semantische Berechnungsmodul nimmt die Information der syntaktischen Analyse und den Namen eines Familienmitglieds, und es berechnet, welche Abschnitte zur generierten Spezifikation gehören sollen. Es führt außerdem die oben beschriebenen Typprüfungen durch, und es meldet alle Typfehler in der Struktur des Familiendokuments. Diese Typfehler zeigen Feature-Interaktions-Probleme an. Sofern keine Typfehler auftraten, extrahiert das Werkzeug das gewünschte Spezifikationsdokument in einem zweiten Durchlauf. Dabei paßt es die parents-Konstrukte an, es wandelt die Default-Properties-Konstrukte in geeignet zusammengestrichene inherit-Konstrukte um, und es erzeugt ggf. den abschließenden Familienmitglied-Sammel-Abschnitt. Danach führen wir eine Typprüfung auf dem resultierenden Anforderungsdokument in einfachem CSP-OZ durch, indem wir von Garrels CSP-OZ-Type-Checker cspozTC [vG99] verwenden.

Das Werkzeug genFamMem kann außer zum Generieren und Prüfen von Familienmitgliedern auch dazu benutzt werden, Dokumentation über die Struktur der Familie zu erzeugen. Es extrahiert die parents-Beziehungen aller Abschnitte und die Vererbungsbeziehungen aller Klassen, es extrahiert die Zuordnung von Abschnitten zu Features, und es konstruiert daraus

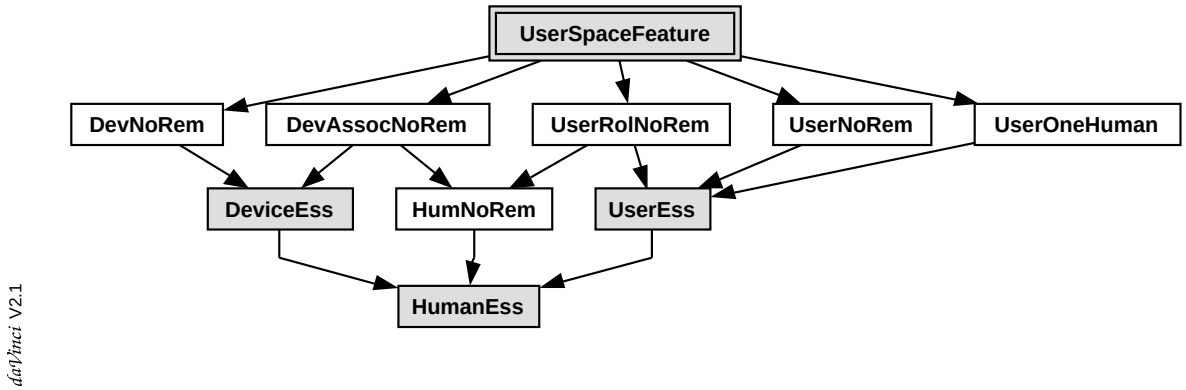

Abbildung 3: Die Benutzungshierarchie der Abschnitte des UserSpace-Features aus unserer Fallstudie [Bre01a, Bre99].

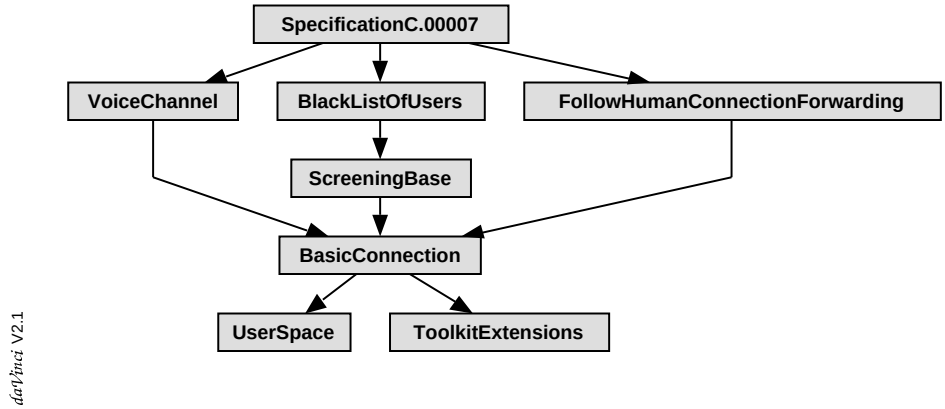

Abbildung 4: Die Benutzungshierarchie der Features des Familienmitgliedes SpecificationC aus der Fallstudie.

einen *Benutzungshierarchie-Graphen* und gibt ihn aus. Der Benutzungshierarchie-Graph beschreibt, welche Abschnitte welche anderen Abschnitte als Voraussetzung benötigen. Dieser Graph wird mit Hilfe des Werkzeugs daVinci [Wer98] visualisiert. Wir können bei genFamMem wählen, ob wir die Benutzungshierarchie aller Abschnitte der Familie sehen wollen (die umfangreich ist!), oder nur der Abschnitte eines Familienmitgliedes, oder nur der Abschnitte eines Features. Weiterhin können wir wahlweise die Abschnitte zu Features verdichten, so daß wir die Benutzungshierarchie aller Features der Familie sehen, oder auch aller Features eines Familienmitgliedes.

Abbildung 3 zeigt zum Beispiel den Benutzungshierarchie-Graphen der Abschnitte des UserSpace-Features aus unserer Fallstudie [Bre01a, Bre99]. Einfache graue Kästen stellen essentielle Abschnitte dar, weiße Kästen änderbare Abschnitte, und doppelt umrandete Kästen stellen Sammel-Abschnitte dar. Da die Benutzungsrelation transitiv ist, werden redundante Pfeile weggelassen, wie zum Beispiel der von DevAssocNoRem nach HumanEss. Abbildung 4 zeigt die Benutzungshierarchie des Familienmitgliedes SpecificationC aus unserer Fallstudie, verdichtet zu Features.

Unsere Erweiterung kann nicht nur für CSP-OZ definiert werden, sondern auch für einfaches Z und für $\mathrm{CSP_Z}$. Letzteres ist ein Dialekt von CSP, der Z für Datendefinitionen be-

nutzt, der aber keine objektbasierten Konstrukte wie Klassen und Vererbung bietet. Unser Werkzeug unterstützt bereits Z und CSP_Z durch Kommandozeilenoptionen, die die weitergehenden Sprachkonstrukte deaktivieren. Wir haben noch keine Experimente mit Familien von Z- oder CSP_Z-Spezifikationen durchgeführt, aber wir planen dies zu tun.

Das Werkzeug ist kostenfrei über seine WWW-Seiten [Bre00b] erhältlich.

4 Zusammenfassung

Wir beschäftigen uns mit der Wartung von formalen Anforderungsdokumenten. In unserem Anwendungsgebiet der Telefonvermittlungssysteme sind unerwünschte Feature-Interaktionen derzeit ein ernsthaftes Hindernis dabei, diese Systeme den Marktanforderungen entsprechend weiterzuentwickeln.

Wir schlagen eine Spezifikationsmethode vor, die entweder 1) Feature-Interaktions-Probleme in Anforderungsdokumenten von Anfang an vermeidet, oder die 2) hilft, diese zu entdecken, so daß sie aufgelöst werden können. Der ursprüngliche Spezifizierer eines Features muß zusätzliche Information darüber dokumentieren, was der Kern eines Features ist und was die Abhängigkeiten von anderen Features sind. Dadurch, daß wir alle Varianten und Versionen eines Systems gemeinsam als eine Familie von Anforderungen behandeln, können wir diese Information auswerten.

Für das Spezifizieren von Familien präsentieren wir in diesem Beitrag (und mit allen Details im Manual [Bre00c]) die Syntax und Semantik eines formalen Feature-Kombinations-Mechanismus'. Er ist eine Erweiterung der Spezifikationssprache CSP-OZ. In unserem Formalismus werden einige Feature-Interaktionen zu Typfehlern.

Insbesondere präsentieren wir ein Werkzeug, das Feature-Interaktions-Probleme anzeigt, das ein gewünschtes Familienmitglied aus dem Familiendokument extrahiert und transformiert, und das Dokumentation über die Struktur der Familie generiert. Das Werkzeug ist frei erhältlich.

In einem in Vorbereitung befindlichen Beitrag [Bre01b] werden wir außerdem über eine größere Fallstudie berichten. Wir zeigen dort, wie das Standardbeispiel einer Feature-Interaktion aus der Einleitung wegen der modularen Anforderungsstruktur, die wir verwenden, nicht auftritt, und wir werden zeigen, wie unser Werkzeug eine weitere Feature-Interaktion erkennt, die wir dann auflösen.

Weitere Forschung ist in mehrere Richtungen sinnvoll. Mehr praktische Erfahrung mit unserer Methode ist wichtig; wir planen, unsere Fallstudie weiter auszubauen. Unser Ansatz sollte auch mit anderen Formalismen als CSP-OZ anwendbar sein, solange der zugrunde liegende Formalismus einen constraint-orientierten Spezifikationsstil erlaubt und ein Ausdrucksmittel für inkrementelle Verfeinerung bietet. Unser Werkzeug unterstützt sogar bereits die Sprachen CSP_Z und Z. Schließlich ist die Beziehung zwischen Anforderungsfamilien einerseits und Programmfamilien andererseits in Hinsicht auf einen wiederverwendungsorientierten Implementierungsprozeß zu untersuchen.

Literatur

[BGK00] Bouma, L. G., Griffeth, N. und Kimbler, K. *Special issue: Feature interactions in telecommunications systems*. Comp. Networks **32**(4) (Apr. 2000).

[Bre97] Bredereke, J. *Communication Systems Design With Estelle – On Style, Efficiency, and Analysis*. Dissertation, Universität Kaiserslautern, Shaker Verlag, Aachen, Deutschland (Aug. 1997).

[Bre99] Bredereke, J. *Modular, changeable requirements for telephone switching in CSP-OZ*. Int. Ber. IBS-99-1, Universität Oldenburg, Oldenburg (Okt. 1999).

[Bre00a] Bredereke, J. *Families of formal requirements in telephone switching*. In Calder und Magill [CaMa00], S. 257–273.

[Bre00b] Bredereke, J. *genFamMem 2.0 Home Page*. Universität Bremen (2000). URL http://www.tzi.de/~brederek/genFamMem/.

[Bre00c] Bredereke, J. *genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements*. Universität Bremen (Okt. 2000). URL http://www.tzi.de/~brederek/genFamMem/.

[Bre00d] Bredereke, J. *Hierarchische Familien formaler Anforderungen*. In Grabowski, J. und Heymer, S. (Hrsg.), „Formale Beschreibungstechniken für verteilte Systeme – 10. GI/ITG-Fachgespräch", S. 31–40, Lübeck (Juni 2000). Shaker Verlag, Aachen.

[Bre01a] Bredereke, J. *Modular, changeable requirements for telephone switching in CSP-OZ – revision 2.0*. Int. Ber. IBS-00-1, Universität Oldenburg, Oldenburg (2001). *In Vorbereitung.*

[Bre01b] Bredereke, J. *A tool for generating specifications from a family of formal requirements*. In „Formal Description Techniques XIV". Kluwer Academic Publishers (Aug. 2001). *In Vorbereitung.*

[CaMa00] Calder, M. und Magill, E. (Hrsg.). *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, Amsterdam (Mai 2000).

[Fis97] Fischer, C. *CSP-OZ: a combination of Object-Z and CSP*. In Bowman, H. und Derrick, J. (Hrsg.), „Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)", Bd. 2, S. 423–438. Chapman & Hall (Juli 1997).

[Fis00] Fischer, C. *Combination and implementation of processes and data: from CSP-OZ to Java*. Dissertation, Berichte aus dem FB Informatik 2/2000, Universität Oldenburg, Oldenburg (Apr. 2000).

[ISO99] ISO Panel JTC1/SC22/WG19, Final Committee Draft, CD 13568.2. *Z Notation* (Aug. 1999).

[KiBo98] Kimbler, K. und Bouma, L. G. (Hrsg.). *Feature Interactions in Telecommunications and Software Systems V*. IOS Press, Amsterdam (Sep. 1998).

[LMB92] Levine, Mason und Brown. *Lex & Yacc*. O'Reilly & Associates Inc (1992).

[Ros97] Roscoe, A. W. *The Theory and Practice of Concurrency*. Prentice-Hall (1997).

[Smi00] Smith, G. *The Object-Z Specification Language*. Kluwer Academic Publishers (2000).

[Spi95] Spivey, J. M. *The Z notation: a reference manual*. Series in Computer Science. Prentice-Hall, New York, 2. Aufl. (1995).

[vG99] von Garrel, J. *Parsing, Typechecking und Transformation von CSP-OZ nach Jass*. Diplomarbeit, Universität Oldenburg, FB Informatik (Juli 1999).

[Wer98] Werner, M. *daVinci V2.1.x Online Documentation*. Universität Bremen (Juni 1998). URL: http://www.tzi.de/~davinci/doc_V2.1/.