Verifikation regelbasierter Konfigurationssysteme

Dissertation

der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Carsten Sinz
aus Albstadt-Ebingen

Tübingen
2003

Tag der mündlichen Qualifikation: 17.12.2003
Dekan: Prof. Dr. Martin Hautzinger
1. Berichterstatter: Prof. Dr. Wolfgang Küchlin
2. Berichterstatter: Prof. Dr. Klaus-Jörn Lange

Für Petra und meine Eltern

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Arbeitsbereich Symbolisches Rechnen der Fakultät für Informations- und Kognitionswissenschaften der Universität Tübingen. Die Implementation des Baubarkeits-Informations-Systems BIS fand größtenteils während meiner Beschäftigung am Steinbeis-Transferzentrum Objekt- und Internet-Technologien (OIT) an der Universität Tübingen statt.

Mein besonderer Dank gilt Herrn Prof. Dr. sc. techn. Wolfgang Küchlin, der das Projekt BIS initiierte und mir immer mit großem Einsatz und tatkräftiger Unterstützung zur Seite stand.

Für die finanzielle Unterstützung danke ich der DaimlerChrysler AG und der T-Systems ITS GmbH (vormals debis Systemhaus Industry GmbH), ohne die das Projekt nicht durchzuführen gewesen wäre. Für die gute Zusammenarbeit während der gesamten Zeit möchte ich mich stellvertretend für viele andere bei Frau S. Epple, Herrn M. Hummel, Herrn A. Nüssle und Herrn R. Wüsthofen von der DaimlerChrysler AG, sowie Herrn A. Krewitz und Herrn G. Müller von der T-Systems ITS GmbH bedanken.

Meinen Kollegen am Wilhelm-Schickard-Institut gilt mein besonderer Dank, insbesondere Dr. Wolfgang Blochinger, ohne den die parallele Version des Beweisers nicht zustande gekommen wäre und Andreas Kaiser, der einen wesentlichen Anteil an der Entwicklung des BIS mitträgt.

Herrn Prof. Dr. Klaus-Jörn Lange danke ich für die Anfertigung des Zweitgutachtens und Herrn Prof. Dr. Hans Kleine Büning für die Erstellung eines externen Gutachtens.

Schließlich möchte ich noch all jenen danken, die – jeder auf seine Art und manchmal mehr als dem jeweiligen bewusst war – zum Gelingen dieser Arbeit beigetragen haben.

Tübingen, Dezember 2003 Carsten Sinz

Inhaltsverzeichnis

Einleitung

1

1.1 Hintergrund

Produkte für den Massenmarkt werden in zunehmendem Maße individuell nach Kundenwunsch gefertigt. Für diese Erscheinung, die zwei sich scheinbar widersprechende Entwicklungen verbindet, hat sich im angelsächsischen Sprachraum der Ausdruck “mass customization” durchgesetzt, ein Begriff, der als Verschmelzung von “mass production” und “customization” erstmals 1987 von Stanley Davis verwendet wurde [Dav87]. Die kundenindividuelle Massenproduktion ist inzwischen bei den verschiedensten Produkten anzutreffen, angefangen von Computern, Möbeln (Koncraft Manufakturen), Fahrrädern (NBIC Panasonic) und Automobilen bis hin zu Musik-CDs (Custom Disc), Brillen (Paris Miki), Vitaminpräparaten (Acumins Corporation) und Kosmetikartikeln (Aveda Personal Blends). Aber auch Dienstleistungen, insbesondere im Internet, werden zunehmend an einzelne Kunden angepasst, wie dies beispielsweise bei der Einstiegsseite des Internet-Buchhändlers Amazon der Fall ist. Diese wird an jeden Kunden individuell adaptiert und kann dadurch speziell ausgewählte persönliche Empfehlungen geben.

Bei variantenreichen Produkten entstehen durch die kundenindividuelle Massenfertigung eine Vielzahl neuer Fragestellungen, die größtenteils mit der unvermeidbar werdenden Automatisierung der Abläufe und dem damit verbundenen Computereinsatz zusammenhängen: Wie kann eine so große Menge von Varianten überhaupt sinnvoll und eindeutig dargestellt und verwaltet werden, sowohl für den Kunden, als auch für bestriebsinterne Belange, z.B. zur Ermittlung der benötigten Teile für die Produktion? Wie kann eine bestimmte Variante produziert werden? Kann eine Variante überhaupt produziert werden, oder sind dazu umfangreiche technische Neuentwicklungen vonnöten?

Gerade in der Automobilindustrie ist die Diversität enorm und daher der Automatisierungsbedarf entsprechend groß. Fahrzeuge der Mercedes-Benz

Abbildung 1.1: Mercedes-Benz C-Klasse.

C-Klasse (siehe Abbildung 1.1) können beispielsweise in einer schier unüberschaubaren Variantenvielfalt bestellt und gefertigt werden. Der Kunde hat die Auswahl zwischen drei Karosserietypen (Limousine, T-Modell, Sportcoupé), bis zu acht Motorisierungsvarianten und einer großen Zahl an Lackierungen, Polsterfarben und -ausführungen, sowie diversen Zusatzausstattungen. Eine kleine Auswahl der über hundert lieferbaren und vielfältig kombinierbaren Sonderausstattungen ist in Tabelle 1.1 zu finden.

Nicht alle Ausstattungspakete sind jedoch kompatibel. So kann das AMG-Styling (772) nicht mit der Anhängervorrichtung mit abnehmbarem Kugelhals (550) kombiniert werden. Die Klimatisierungsautomatik (581) darf nur bei gleichzeitigem Einbau einer Batterie mit größerer Kapazität (673) verwendet werden, außer bei den größeren Benzinmotor-Varianten mit 2,6 und 3,2 Litern Hubraum. Verträglichkeitsbedingungen dieser Art sind typisch für variantenreiche Produkte. Sie können aus den unterschiedlichsten Gründen vorhanden sein und aus den verschiedensten Quellen stammen. Eine der häufigsten Ursachen für eine Inkompatibilität ist der räumliche Platzbedarf einzelner Komponenten. Daneben liefern aber auch gesetzliche, länderspezifische oder vertriebliche Rahmenbedingungen weitere Einschränkungen in der Kombinierbarkeit.

Code	Benennung
550	Anhängervorrichtung mit abnehmbarem Kugelhals
500	Außenspiegel links und rechts elektrisch heranklappbar
673	Batterie mit größerer Kapazität
614	Bi-Xenonscheinwerfer mit Scheinwerferreinigungsanlage und dynamischer Leuchtweitenregulierung
231	Garagentoröffner im Innenspiegel integriert
551	Einbruch-Diebstahl-Warnanlage (EDW) mit Abschleppschutz
581	Komfort-Klimatisierungsautomatik THERMOTRONIC
280	Lenkrad in Lederausführung (zweifarbig) mit Chromspange
921	Motor mit Pfanzenölmethylester-Betrieb (Bio-Diesel)
353	Audio 30 APS (Navigationssystem mit integriertem Radio und CD-Laufwerk)
293	Sidebags im Fond
671	Leichtmetallräder 4-fach, 7-Speichen-Design
228	Standheizung mit Fernbedienung
772	Styling AMG

Tabelle 1.1: Auszug Sonderausstattungen Mercedes-Benz C-Klasse.

1.2 Motivation und Zielsetzung

Zur automatisierten Verwaltung bedarf es einer systematischen Darstellung dieser Einschränkungen. Typischerweise wird hierfür eine formale *Dokumentationssprache* verwendet, in der sich die Produktstruktur und die Kompatibilitätsbedingungen bzw. Beschränkungen darstellen lassen. Oft – und gerade bei komplexen Produkten – wird in der Dokumentationssprache auch der Zusammenhang zwischen den kundenorientierten Verkaufscodes und den produktionsorientierten Teilenummern dargestellt. Die in der Dokumentationssprache niedergelegte *Produktdokumentation* besteht meist aus einem Satz von *Regeln*. Das Regelsystem für die Limousinen der Mercedes-Benz C-Klasse besteht beispielsweise aus 1.149 strukturbeschreibenden Einschränkungsregeln und 18.508 Regeln, die den Zusammenhang zwischen Verkaufscodes und Teilen herstellen. Es ist nicht verwunderlich, dass die Erstellung und Wartung eines Regelsystems dieser Größe besondere Anforderungen an das zuständige Personal stellt. Der Bedarf an Computerunterstützung bei den verschiedensten Wartungsarbeiten am Regelsystem ist immens. Durch die ständige Weiterentwicklung der Produkte und sich ändernde Produktionsumstände und -stätten handelt es sich bei der Dokumentation darüberhinaus meist nicht um eine fixe, unveränderliche

Produktbeschreibung, sondern um ein sich kontinuierlich veränderndes Regelwerk.

Durch die Verwendung einer logischen Dokumentationssprache mit (mathematisch) exakt definierter Semantik können formale Methoden, wie sie beispielsweise in der Hard- und Softwareverifikation eingesetzt werden, auf die Verifikation und Validierung der Produktdokumentation übertragen werden.

In dieser Arbeit wollen wir die Voraussetzungen, Grundlagen und Methoden zusammenstellen, die zu einer Anwendung formaler Verifikationsverfahren auf regelbasierte (Produkt-)Konfigurationssysteme benötigt werden. Dazu gehört neben der Vorstellung bestehender Verfahren auch die Entwicklung neuer formaler Modelle, die von den Besonderheiten konkreter Produktdokumentationssysteme abstrahieren und die dort zum Teil vorhandenen semantischen Ungereimtheiten beseitigen. Solche treten bei regelbasierten Systemen immer wieder auf (z.B. in der Form von Sonderregelungen, Präzedenz- oder Auswahlregeln) und erschweren eine systematische Analyse, die sich nur auf die Regeln alleine stützt. Die meist vorhandene, (scheinbar) eindeutige Semantik der Regeln wird durch die Algorithmen zur Regelverarbeitung so sehr häufig verwässert.

Vor diesem Hintergrund sehen wir es als geboten, auch die Auswertungsverfahren und Verarbeitungsalgorithmen in die Verifikation mit einzubeziehen, wodurch die Verifikation mehr zu einer Programmverifikationsaufgabe wird und sich von der ansonsten weit verbreiteten, ausschließlich statischen Analyse der Regeln entfernt ("Programmverifikation statt Datenverifikation"). Bei dieser Form der Verifikation der Regelsysteme als Programme reduziert sich die Rolle der Regeln auf eine Parametrisierung des Verarbeitungsalgorithmus. Anforderungen an ein in sich stimmiges (konsistentes) Regelsystem werden dann als erwünschte Eigenschaften an das formalisierte Verarbeitungsprogramm formuliert. Da die Verabreitungsprogramme typischerweise recht einfach sind (die "Logik" steckt in den Regeln), ist eine Programmverifikation deutlich einfacher durchzuführen als dies bei der Verifikation a priori uneingeschränkter Programme in einer gängigen Programmiersprache wie Java oder C++ der Fall wäre. Wir stellen in dieser Arbeit ein auf dynamischer Aussagenlogik (PDL) beruhendes Verifkationsverfahren vor, das sich auf diese Grundideen stützt.

Neben der Formalisierung als Regelverarbeitungsprogramm werden Konsistenzkriterien identifiziert, die von einer korrekten Dokumentation eingehalten werden müssen, sowie "weichere" Validierungskriterien, die auf mögliche fehlerhafte Stellen in der Dokumentation hindeuten. Darüberhinaus werden algorithmische Aspekte der Verifikation untersucht, die mit der Anwendung automatischer Beweisverfahren zusammenhängen. Dazu gehören die Themen Komplexität, Kompilierung und Parallelisierbarkeit.

1.3 Aufbau dieser Arbeit

Kapitel 2 gibt eine Übersicht über Anforderungen, betriebliche Einordnung und grundsätzliche Probleme der Produktdokumentation und schließt mit der ausführlichen Darstellung zweier Fallstudien. In Kapitel 3 beschreiben wir verschiedene in der Literatur vorgestellte logische Formalismen zur Darstellung von Konfigurationsfragen. Die von uns zur Verifikation der Produktdaten herangezogene dynamische Aussagenlogik (PDL) wird hier besonders ausführlich dargestellt. Außerdem entwickeln wir eine spezielle Kripke-Semantik, die sich zur Behandlung unserer Konsistenztests als vorteilhaft erwiesen hat. Für diese geben wir auch eine Transformation in reine Aussagenlogik an. Kapitel 4 entwickelt verschiedene Konsistenztests, die anhand der Formalisierung der Mercedes-Benz-Fahrzeugdokumentation in PDL dargestellt werden. Diese Tests sind unterteilt in “statische”, die Dokumentation nur zu einem festen Zeitpunkt berücksichtigende, und “dynamische”, die auch die zeitliche Entwicklung der Dokumentation mit berücksichtigen. Eine umfangreiche Darstellung der experimentellen Ergebnisse der Konsistenzprüfung der Fahrzeugdaten ist ebenfalls Teil dieses Kapitels. In Kapitel 5 untersuchen wir die Komplexität der sich aus den Konsistenztests ergebenden aussagenlogischen Erfüllbarkeitsprobleme anhand verschiedener Parameter. Wir geben auch mehrere Verfahren zur Kompilierung der Produktdaten an. Die Kompilierung hat die beschleunigte Durchführung umfangreicher Testreihen zum Ziel. Kapitel 6 schließlich gibt einen kurzen Einblick in den von uns entwickelten parallelen Beweiser für Aussagenlogik. Kapitel 7 beschließt mit einer Zusammenfassung der Ergebnisse dieser Arbeit.

In den beiden Anhängen A und B stellen wir unser für DaimlerChrysler als Zusatz zu deren elektronischen Produktdokumentationssystemen DIALOG und EDS/BCS entwickeltes Baubarkeits-Informations-System BIS vor, sowie ausgewählte Ergebnisse der Analyse der Mercedes-Benz-Produktdaten.

2

Produktdokumentationsverfahren

Die *Produktdokumentation* besteht aus allen Dokumenten, die für die Entwicklung, die Produktion, und den Vertrieb eines Produkts benötigt werden. Dazu gehört neben Betriebsanleitungen und Präsentationsprospekten hauptsächlich auch die technische Dokumentation, die für die Fertigung des Produkts notwendig ist und parallel zur Entwicklung des Produkts erstellt werden muss. Genau wie das Produkt selbst (und dessen Produktionsbedingungen) ist auch die Dokumentation nicht statisch, sondern verändert sich im Laufe der Zeit.

Die *technische Dokumentation* umfasst neben Planungsunterlagen, technischen Zeichnungen, CAD-Modellen und Software auch Unterlagen zu Produktionsabläufen und zur logischen Produktstruktur.

In dieser Arbeit beschränken wir uns auf den Teil der technischen Produktdokumentation, der sich mit der *logischen Struktur* des Produkts, den möglichen *Konfigurationen*, beschäftigt. Eine Beschreibung aller möglichen Konfigurationen bezeichnen wir als *Produktübersicht*. Diese Produktübersicht verwendet dieselben Begriffe, die auch der Kunde zur Spezifikation seines Auftrags verwendet, und die daher auch in den Verkaufskatalogen auftauchen. Technische Details zum Aufbau des Produkts sind hier typischerweise nicht zu finden. Getrennt davon ist die *Produktstruktur* zu sehen, also die Information, aus welchen Komponenten sich das Produkt auf welche Weise zusammensetzt. Diese ist technisch orientiert, verwendet Begriffe aus der Entwicklung (zum Teil auch der Produktion) und geht im Detaillierungsgrad bis hinunter zur “letzten Schraube”, also den Einzelteilen, aus denen sich das Produkt zusammensetzt. Häufig wird aus der Produktstruktur der Aspekt der räumlichen Beschreibung der Komponenten nochmals ausgegliedert, und dadurch als eigenständige *Produktgeometrie* (meist in der Form von CAD-Zeichnungen) sichtbar. Eine schematische Einteilung der technischen Produktdokumentation nach diesen Kategorien zeigt Abbildung 2.1.

Wenn im Folgenden von Produktdokumentation die Rede ist, so ist dies

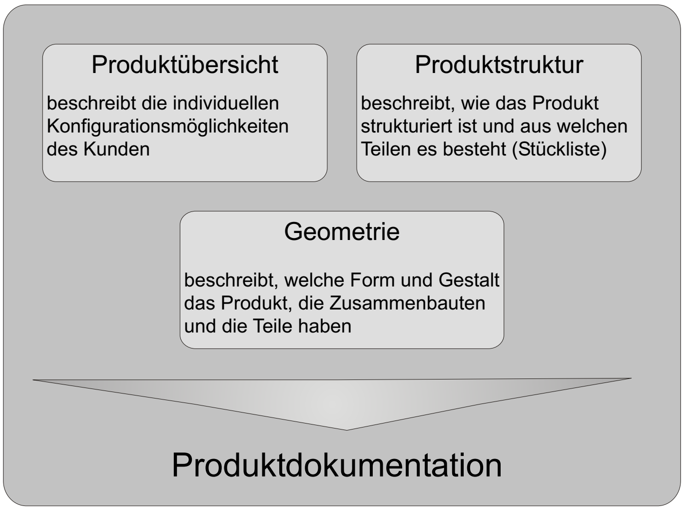


Abbildung 2.1: Schematische Einteilung der technischen Produktdokumentation (DaimlerChrysler).

in der auf die technische Dokumentation eingeschränkten Bedeutung zu verstehen. Die Produktgeometrie spielt dabei auch nur insofern eine Rolle, als sich deren Eigenschaften mittels abgeleiteter Restriktionen auf die Produktübersicht und Produktstruktur übertragen.

2.1 Betriebliche Einordnung

Die Produktdokumentation begleitet ein Produkt von dessen Entstehung bis zu dessen Verkauf, zum Teil sogar noch darüber hinaus (sofern der Kundendienst auch auf die Dokumentation zurückgreift). Eine Konsequenz daraus ist, dass die Nutzung und Erstellung der Produktdokumentation nicht auf einen Unternehmensbereich beschränkt ist, vielmehr kommen fast alle Bereiche mit der Dokumentation in Kontakt. Dazu gehören Planung, Entwicklung, Fertigung, Montage, Marketing, Vertrieb und Kundendienst. Die einzelnen Abteilungen haben unterschiedliche Anforderungen an die Dokumentation, eigene Sichten, Begriffsbildungen und Prioritäten, die sich von denen anderer Bereiche erheblich unterscheiden können.

2.1.1 Marketing und Vertrieb

Es ist Aufgabe der Marketing- und Vertriebsabteilungen darüber zu entscheiden, welche Produktvarianten angeboten werden sollen. Durch das Marketing kann der Anstoß zu einer technischen Neuentwicklungen erfolgen, aber auch die Entscheidung, eine technisch mögliche Variante aus betrieblichen Gründen nicht zum Verkauf anzubieten. So ist es beispielsweise üblich, technische Neuentwicklungen, die als Wettbewerbsvorteil angesehen werden, zuerst nur in den gehobenen Produktklassen anzubieten.

Die Begriffe der Verkaufsprospekte, die dem Kunden die Möglichkeit zur Konfiguration geben und die auch die Grundlage der Produktübersicht bilden, werden hier ebenfalls festgelegt. Der Vertrieb ist daher maßgeblich an der Entwicklung der Produktübersicht beteiligt, wobei zur Klärung der technischen Machbarkeit eine enge Zusammenarbeit mit der Entwicklungsabteilung vonnöten ist.

Produktkonfiguration wird im Vertriebsbereich daher als Verkaufshilfe für den Kunden bzw. den Händler angesehen. Der Kunde wählt eine bestimmte Produktvariante aus, von der dann der Händler unter Zuhilfenahme des vertrieblich vorgegebenen Konfigurationswissens zu entscheiden hat, ob diese Variante erhältlich ist.

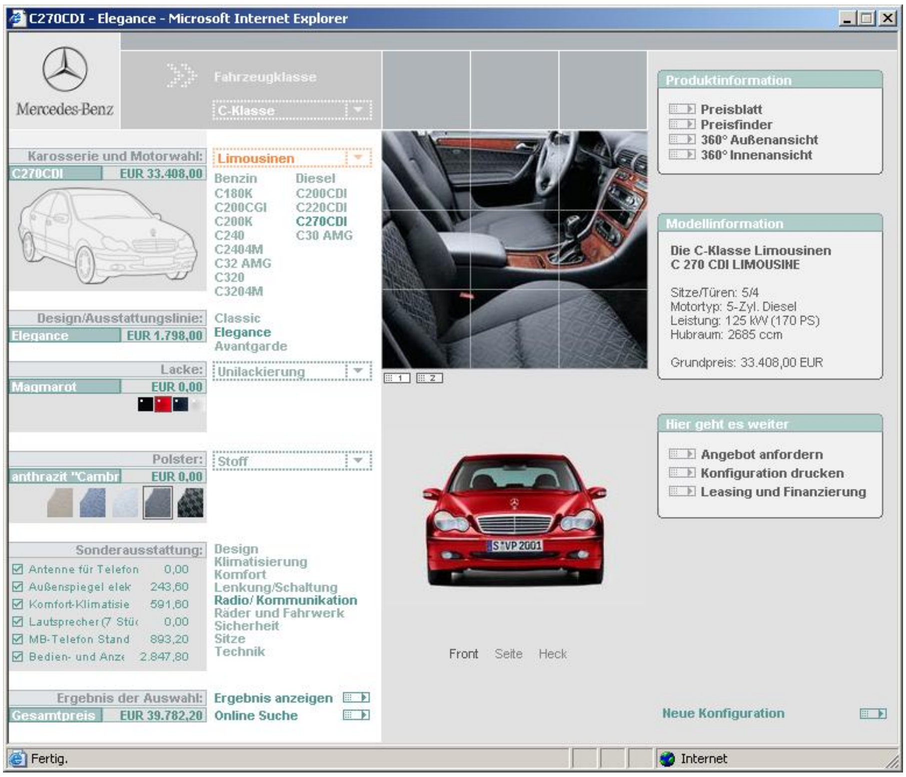
Ein elektronischer, beispielsweise über das Internet zugänglicher Produktkonfigurator sollte außer dem Wissen über verfügbare Varianten auch die Fähigkeit haben, dem Kundenwunsch möglichst nahe kommende Alternativvorschläge zu unterbreiten, falls die gewünschte Konfiguration nicht erhältlich ist.

Verkaufskonfiguratoren sind im Internet immer häufiger anzutreffen und zeichnen sich durch Interaktion mit dem Benutzer und Online-Konsistenzprüfung aus. Die Möglichkeit, Alternativen zu generieren ist inzwischen weit verbreitet. Noch nicht generell etabliert ist die Fähigkeit, Gründe für eine nicht erlaubte Konfiguration anzugeben. Bei Internet-Konfiguratoren spezifiziert der Kunde das von ihm gewünschte Produkt über nach außen hin erkennbare Attribute, wie z.B. Farbe, spezielle Funktionalität, oder optional erhältliches Zubehör.

Stellvertretend für andere ähnliche Verkaufskonfiguratoren ist in Abbildung 2.2 der von DaimlerChrysler für die Mercedes-Benz-Fahrzeuge verwendete Internet-Konfigurator zu sehen. [1]

---

[1] Der Konfigurator ist unter `www.car-configurator.mercedes-benz.com` zu finden.



C270CDI - Elegance - Microsoft Internet Explorer

Fahrzeugklasse: C-Klasse

Karosserie und Motorwahl: Limousinen
C270CDI EUR 33.408,00

Design/Ausstattungslinie:
Elegance EUR 1.798,00

Lacke: Unilackierung
Magmarot EUR 0,00

Polster: Stoff
anthrazit "Cambr EUR 0,00

Sonderausstattung:

☑ Antenne für Telefon	0,00
☑ Außenspiegel elek	243,60
☑ Komfort-Klimatisie	591,60
☑ Lautsprecher (7 Stüc	0,00
☑ MB-Telefon Stand	893,20
☑ Bedien- und Anze	2.847,80

Ergebnis der Auswahl:
Gesamtpreis EUR 39.782,20

Modellinformation

Die C-Klasse Limousinen
C 270 CDI LIMOUSINE

Sitze/Türen: 5/4
Motortyp: 5-Zyl. Diesel
Leistung: 125 kW (170 PS)
Hubraum: 2685 ccm

Grundpreis: 33.408,00 EUR

Fertig.

Abbildung 2.2: Internet-Konfigurator für Mercedes-Benz-Fahrzeuge.

Haag bezeichnet in seiner Konfigurations-Typisierung [Haa98] diese Art von Produktkonfiguration als “high-level configuration” im Gegensatz zur “low-level configuration”, wie sie in den produktionsorientierten Abteilungen anzutreffen ist. Für Haag ist High-Level-Konfiguration ein kreativer Prozess, der durch Interaktivität und die Verwendung von abstrakten Produktattributen gekennzeichnet ist. Im Gegensatz dazu ist Low-Level-Konfiguration durch maschinelle Massenverarbeitung gekennzeichnet und durch eine Auflösung des Teilebedarfs bis zu einem Detaillierungsgrad, der für den Kunden zwar nicht mehr interessant, für die Produktion aber unerlässlich ist.

2.1.2 Entwicklung

Die Daten zur Produktstruktur werden zum größten Teil in den Entwicklungsabteilungen erstellt. Die Produktstruktur enthält das Wissen über sämtliche technischen Variationsmöglichkeiten des Produkts in der Form von Abhängigkeiten, Strukturinformationen (Anordnung und Kombinierbarkeit von Komponenten) und technischen Zeichnungen. Da ein Produkt oder Teil anhand der während der Entwicklungsphase erstellten Konstruktionsdaten und -zeichnungen nicht nur aufgebaut (synthetisiert), sondern auch in seine Bestandteile zerlegt (analysiert) werden kann, beinhaltet die zum Aufbau des Produkts erstellte Dokumentation auch die insbesondere bei der Zerlegung sichtbar werdende Struktur. Die durch diese Analyse gewonnenen Daten können dann als Grundlage der Dokumentation der Teil-Ganzes-Abhängigkeiten dienen. Veranschaulichen kann man sich diese aus den Konstruktionsdaten abgeleitete Strukturinformation anhand der Analogie zu einer Explosionszeichnung, wie sie in Abbildung 2.3 dargestellt ist.

In einer solchen Zeichnung wird der interne Aufbau eines Teils oder Produkts erkennbar. Oft lassen sich die einzelnen Bestandteile Schritt für Schritt noch weiter zerlegen, über Teilzusammenbauten bis schlussendlich hin zu den Einzelteilen. Man erhält dadurch eine hierarchische Darstellung der Abhängigkeitsbeziehungen zwischen einzelnen Teilen und dem aus diesen Teilen assemblierten Ganzen.

Bei variantenreichen Produkten ist es nicht sinnvoll, die Gesamtstruktur eines jeden Modells getrennt abzulegen. Denn viele Varianten unterscheiden sich nur minimal, so dass ein mehrfaches Ablegen der gesamten Strukturdaten ein hohes Maß an Redundanz erzeugen würde. Außerdem versuchen die Entwicklungsingenieure neue Komponenten häufig so zu entwickeln,

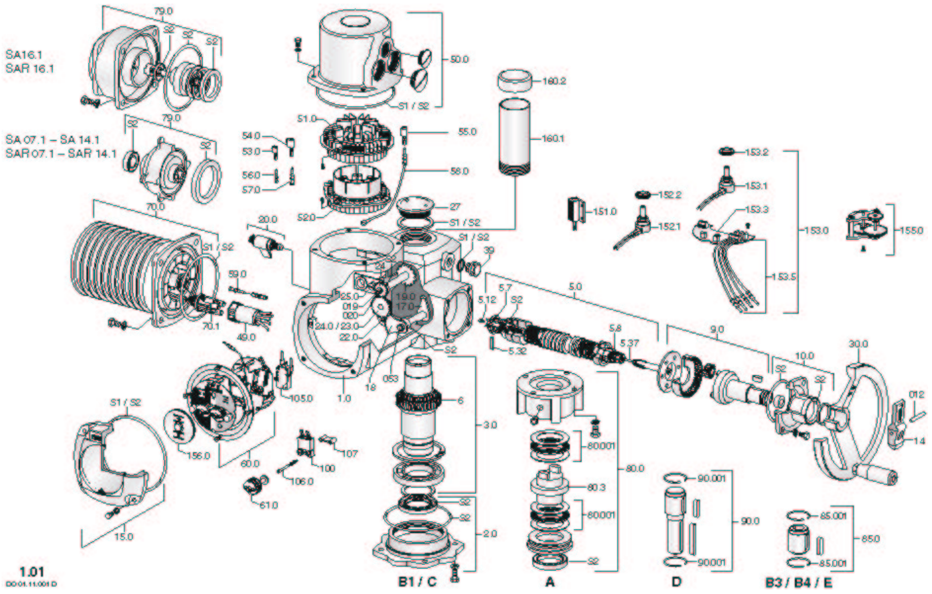


Abbildung 2.3: Explosionsskizze eines Drehantriebs.

dass sie für eine Vielzahl von Modellvarianten passend sind.

Für welche Varianten eine Komponente passend ist, und zu welchen anderen Komponenten Abhängigkeiten wie Ausschlüsse oder gegenseitiges Erfordern bestehen, kann von vielen Faktoren abhängen:

- **Platzbedarf**: Eine Komponente benötigt Platz, der schon von einer anderen Komponente belegt wird. In manchen Fällen kann der Platzbedarf auch erst während des Betriebs der Komponente entstehen.
- **Abhängigkeit von weiteren Ressourcen**, die von anderen Komponenten zur Verfügung gestellt werden, z.B. Kapazität einer Energiequelle, Speicherkapazität einer Festplatte oder Übertragungskapazität einer Verbindungsleitung. So muss beispielsweise bei elektrischen Produkten der Gesamtbedarf aller Komponenten an Energie durch die zur Verfügung stehende Energiequelle gedeckt werden können.
- **Funktionalität:** Von einem Produkt einer bestimmten Kategorie wird eine Mindestfunktionalität erwartet. Fehlen dafür notwendige Komponenten, so ist das Produkt nicht funktionsfähig oder unvollständig.

In die Produktstruktur fließt somit technisches Wissen über mögliche Zusammenbauten, Ausschlüsse, Unverträglichkeiten, und funktionale Abhängigkeiten ebenso ein, wie Aussagen über Material- und Teilebedarf sowie über den internen Aufbau des Produkts aus Teilkomponenten. All diese Zusammenhänge müssen in einer möglichst kompakten und leicht wartbaren Form abgelegt werden. Welche Möglichkeiten es hierzu gibt, soll weiter unten erläutert werden.

Der Nutzen der Produktdokumentation für die Entwickler selbst besteht zum einen darin, bereits bestehende Abhängigkeiten zwischen Teilen bzw. Komponenten nachschlagen zu können. Andererseits gibt sie ihnen aber auch die Möglichkeit, einen Überblick über die oft schwer überschaubare Menge der von Marketing und Vertrieb gewünschten Varianten zu erhalten. Letzteres ist durch den Zugriff auf die Produktübersicht gewährleistet.

In den Entwicklungsabteilungen liegt auch das Wissen vor über die Umsetzung einer Variante in die dafür benötigten Teile. Auch diese Informationen werden daher hier von den Entwicklungsingenieuren verwaltet und zusammen mit der Produktstruktur abgelegt.

2.1.3 Produktion

Die Produktinformation dient der Produktion dazu, die für eine vorgegebene Produktinstanz benötigten Teile zu bestimmen. Hierzu ist, im Gegensatz zum Vertrieb, weniger die funktionale Sicht von Interesse als vielmehr die teileorientierte. Strukturelle Informationen der Dokumentation können darüberhinaus von der Produktion dazu verwendet werden, Produktions- und Montageprozesse festzulegen und zu optimieren.

Da in der Produktion letztlich die technische Machbarkeit entscheidend ist, gibt es viele Anknüpfungspunkte und Gemeinsamkeiten zur Entwicklungsdokumentation. Es gibt aber auch Differenzen: Während die Dokumentation in der Entwicklung eine idealisierte Momentaufnahme dessen darstellt, was Ingenieure zum gegenwärtigen Zeitpunkt bewerkstelligen können, ist diese zeitlich eingeschränkte Sicht in der Produktion nicht möglich. Hier müssen zusätzliche Gesichtspunkte, die auch in der Zukunft liegen können, mitberücksichtigt werden: Ist ein bestimmtes Teil zu einem festgelegten Zeitpunkt verfügbar? In welcher Fertigungsstätte, -halle und an welcher Fertigungsstraße soll ein bestimmtes Produkt hergestellt werden? Welche Version des Produkts soll, z.B. bei einem anstehenden Modelljahr- oder Variantenwechsel, hergestellt werden? Aspekte der Produktionsplanung, insbesondere der Verfügbarkeit von Teilen und Maschinen, müssen sich daher auch in der Dokumentation für die Produktion niederschlagen.

2.1.4 Kundendienst

Für den Kundendienst ist ein Zugriff auf die Produktdaten notwendig, um im Bedarfsfall die richtigen Ersatzteile aussuchen zu können. Diese Auswahl kann von Eigenschaften der speziellen Variante abhängen, und der Austausch gegen ein (eventuell weiterentwickeltes, leicht verändertes) Ersatzteil kann zu Unverträglichkeiten mit bereits eingebauten Komponenten führen. Das Wissen um solche Unverträglichkeiten ist aber in den Produktstrukturdaten der Produktdokumentation durch die Entwickler abgelegt. Allerdings handelt es sich dabei meist um einen weit zurückliegenden Stand der technischen Entwicklung und damit auch der Produktdaten. Der Kundendienst ist daher auf eine Archivierung aller zurückliegenden Dokumentationsstände angewiesen.

2.1.5 Informationsfluss

Das Zusammenspiel der verschiedenen Abteilungen im Zusammenhang mit der Produktdokumentation und der dazu erforderliche Informationsfluss ist nochmals in Abbildung 2.4 zusammenfassend dargestellt.

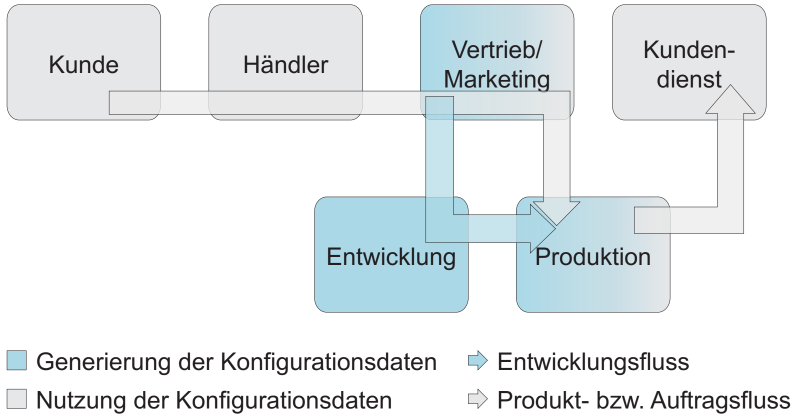


Abbildung 2.4: Erstellung, Fluss und Nutzung der Produktdaten

Generiert werden die Konfigurationsdaten der Produktübersicht von den Marketing- und Vertriebsabteilungen, indem diese vom Kunden spezifizierbare Eigenschaften des Produkts herausarbeiten. In Zusammenarbeit mit der Entwicklungsabteilung wird dann die Produktübersicht erstellt, durch welche Einschränkungen über bestellbare Produktvarianten dargelegt werden. Die so festgelegten Restriktionen spiegeln sowohl technische Machbarkeitsgrenzen als auch vertrieblich gewünschte Einschränkungen wieder. Die Entwicklung fügt Daten zur Produktstruktur und zur Umsetzung der Aufträge in Teilelisten hinzu. In der Produktion werden die Produktdaten dann hauptsächlich um zeitliche Eigenschaften wie Verfügbarkeit von Teilen und Maschinen ergänzt.

Genutzt werden die Daten zuallererst vom Kunden, der für seine Bestellung zusammen mit dem Händler eine der möglichen Varianten auszuwählen hat. Diese Daten werden dann, nach Prüfung der Plausibilität und Korrektheit der Bestelldaten, über den Vertrieb an die Produktion weitergeleitet, die dann eine abschließende Korrektheitsprüfung durchführen und die zur Produktion benötigten Teile bestimmen kann. Die Produktionsdaten werden dann zur Archivierung an die Kundendienstabteilung weitergeleitet, die diese im Bedarfsfall wieder aktiviert und für Ersatzteilbestellungen zu Rate zieht.

2.2 Konfigurationsmodelle

Eine Konfiguration ist nach Stefiks recht knapper und abstrakter Definition lediglich eine Anordnung von Teilen (“arrangement of parts”) [Ste95]. Als Konfigurationsaufgabe (“configuration task”) bezeichnet er dann die Anforderung, Teile (*Komponenten*) so auszuwählen, anzuordnen und zu kombinieren, dass sie eine zuvor gegebene Spezifikation erfüllen.

Mittal geht von annähernd demselben Konfigurationsbegriff wie Stefik aus, verfeinert aber dessen Definition [MF89, MF90]. Mittal macht dazu die Beobachtung, dass zwei wichtige Prämissen bei fast allen Konfigurationsaufgaben erfüllt sind: Erstens werden die Produkte bezüglich einer bekannten Architektur konfiguriert, die sich über Funktionen des Produkts definiert, und zweitens gibt es für jede Funktionalität eine oder einige wenige Schlüsselkomponenten, die wesentlich zur Bereitstellung dieser Funktionalität beitragen.

Sabin und Weigel [SW98] heben ähnliche Eigenschaften hervor, die sie als wesentlich erachten, betrachten zusätzlich aber noch die Beziehungen und Wechselwirkungen der Komponenten untereinander. Sie identifizieren als typische Merkmale des Konfigurationsproblems: (1) Das zu konfigurierende Produkt setzt sich aus Instanzen einer fest vorgegebenen Menge von Komponententypen zusammen und (2) Komponenten interagieren in vorher festgelegter Weise.

Felfernig *et al.* beschreiben eine formal-logische Herangehensweise an das Konfigurationsproblem [FFJS00]. Dazu geht er von einer *Bereichsbeschreibung* (domain description, DD) aus, in der mittels logischer Aussagesätze verfügbare Komponententypen, deren Attribute und Zusammenspiel, aber auch deren Abhängigkeiten sowie zulässige Produktvarianten beschrieben sind. Die Bereichsbeschreibung entspricht in etwa der funktionalen Architektur Mittals, die Abhängigkeiten den festgelegten Komponenten-Interaktionen bei Sabin und Weigel.

Eine Konfigurationsaufgabe besteht bei Felfernig nun darin, zu einer Bereichsbeschreibung und einer Benutzerspezifikation (die Anforderungen an das gewünschte Produkt festlegt) eine Komponentenauswahl zu treffen, sodass sowohl die Bereichsbeschreibung als auch die Benutzerspezifikation eingehalten bzw. erfüllt sind.

In der Literatur wurde eine Vielzahl von detaillierten Konfigurationsmodellen beschrieben, die zum Teil ganz unterschiedliche Aspekte der Konfiguration in den Vordergrund stellen.[2] Wir wollen eine kleine Auswahl zusammen mit deren Vor- und Nachteilen hier vorstellen:

Produktionsregeln treten meist in der Form von *wenn-dann*-Regeln auf [HR85], die Abhängigkeiten zwischen Komponenten ausdrücken oder Aktionen angeben, die unter bestimmten Bedingungen auszuführen sind. Unter den frühesten Vertretern dieser Klasse sind die häufig zitierten Expertensysteme R1 und XCON [McD82, BOBS89] hervorzuheben, die von DEC (Digital Equipment Corporation) zur Konfiguration von Computersystemen entwickelt wurden. Die Vor- und Nachteile von Produktionsregeln sind:

- ✓ Häufig ergibt sich eine natürliche Formulierung von Expertenwissen als Situations-Aktions-Regeln.
- ✓ Einfach zu verstehender und sehr flexibler Formalismus.

- ✗ Versteckte Abhängigkeit vom Auswertungsalgorithmus, der Regelauswahl und -anwendung kontrolliert.
- ✗ Oft keine klare Trennung von Kontroll- und Gegenstandswissen.
- ✗ Keine standardisierte Semantik für Regelsysteme.

---

[2] Siehe z.B. Stumptners umfassende Übersicht der aktuellen Arbeiten auf diesem Gebiet [Stu97].

- ✗ Wenige oder keine Modularisierungs- und Strukturierungsmöglichkeiten.[3]

Ressourcenbasierte Systeme: Hier steht der Begriff der Ressource im Mittelpunkt. Dabei ist eine Ressource ein Werkstoff, eine Dienstleistung oder ein abstraktes Gut, für das es im Produkt “Produzenten” und “Konsumenten” gibt (z.B. Energie). In ressourcenbasierten Modellen [JH98] muss für jede Komponente angegeben werden, welche Ressourcen sie produziert, konsumiert und verwendet. Eine Konfigurationsaufgabe besteht dann im Auffinden einer Menge von Komponenten, die zusammen allen funktionalen Anforderungen genügen, und für die sich alle Ressourcen im Gleichgewicht befinden. Dazu müssen für alle Ressourcen ausreichend Produzenten vorhanden sein, um den Anforderungen der Konsumenten zu entsprechen.

- ✓ Ursachen für Ausschlüsse und Abhängigkeiten detailliert im Modell festgehalten, daher höherer Informationsgehalt.
- ✓ Mit anderen Ansätzen (z.B. Produktionsregeln) kombinierbar.

- ✗ Hoher Dokumentationsaufwand, da Ressourcen und Funktionalitäten herausgearbeitet und modelliert werden müssen.
- ✗ Hierarchischer Aufbau und räumliche Struktur der Produkte oft nur unzureichend berücksichtigt bzw. nur umständlich über Ressourcen zu dokumentieren.
- ✗ Inferenzalgorithmen noch unzureichend untersucht.

CSP-Modelle: Bei *Constraint Satisfaction Problemen* (CSPs) geht es darum, einer gegebenen (endlichen) Menge von Variablen Werte aus fest vorgegebenen (endlichen) Wertemengen zuzuweisen, so dass bestimmte, durch Relationen vorgegebene Randbedingungen, eingehalten werden [Tsa93]. Voraussetzung für CSP-basierte Konfigurationsmodelle ist, dass sich jede Konfiguration aus einer Menge von grundlegenden, atomaren Auswahlmöglichkeit zusammensetzen lässt. Jede Wahlmöglichkeit erlaubt dann, sich zwischen verschiedenen, funktional äquivalenten Varianten zu entscheiden. Damit entspricht jede Auswahlmöglichkeit einer CSP-Variablen, und jede wählbare Variante einem Wert im Wertebereich dieser Variable. Abhängigkeiten zwischen den verschiedenen Varianten wie Restriktionen und funktionale

---

[3] Unter den Vorschlägen zur Strukturierung sind z.B. strukturelle Protokolle zur Regelauswahl [SBJ87], Regelgruppen [BCP97] oder Einbettung in Objekte [MBS+93, Pac94].

Vollständigkeitseigenschaften lassen sich in diesem Modell dann über Relationen definieren. Jede Relation schränkt die zulässigen Variablenkombinationen der enthaltenen Variablen ein. Aus dem ursprünglichen CSP-Modell wurden verschiedene Varianten abgeleitet (dynamic CSP [MF90], hierarchical domain CSP [Kök94], composite CSP [SF96]), die sich mit Verfeinerungen unterschiedlicher Aspekte befassen.

- ✓ Etablierter, rein deklarativer Formalismus.
- ✓ Struktur der Produkte lässt sich in einigen Erweiterungen (z.B. composite CSP) gut dokumentieren.
- ✓ Vielzahl gut untersuchter (Such-)Algorithmen vorhanden.
- ✓ Modularisierung in hierarchischen Varianten gut unterstützt.

- ✗ Bei vielen Algorithmen explizite (nicht-symbolische) Darstellung der Relationen erforderlich.
- ✗ Strukturierung nur durch Auswahl und Festlegung der Relationen oft wenig intuitiv.

Graphen-basierte Modelle: Bei diesen sind die Komponenten als Knoten eines Graphen (meist eines Baums) dargestellt, der die Produktstruktur beschreibt. In der einfachsten Form, den UND-ODER-Graphen, teilt ein Knoten sich entweder in seine konstituierenden Komponenten (UND-Knoten) oder in eine Auswahl zwischen alternativen Komponenten (ODER-Knoten) auf. Notwendig dazu ist die Einführung neuer, abstrakter Komponenten (den ODER-Knoten entsprechend), die funktional verwandte Komponenten zusammenfassen bzw. klassifizieren [CGS$^+$89]. Die Struktur des Produkts wird über die graphische Teil-Ganzes-Komponentenhierarchie der UND-Knoten repräsentiert. Wegen der Ähnlichkeit zu objekt-orientierten Klassenhierarchien wurde auch UML [RJB98] als (grafische) Sprache zur Beschreibung der Komponenten- und Abstraktionshierarchien vorgeschlagen [FFJ00].

- ✓ Grafischer Formalismus oft sehr eingängig und intuitiv.
- ✓ Komponenten- und Abstraktionshierarchie direkt aus dem Graph ablesbar.
- ✓ Gute Strukturierungsmöglichkeiten.
- ✓ Kombinierbar mit anderen Modellen, auch individuell für jeden Knoten.

- ✗ Grafischer Formalismus für sehr große Produktstrukturen weniger geeignet.
- ✗ Nur in Kombination mit anderen Formalismen (zur Beschreibung von Restriktionen) sinnvoll einsetzbar.
- ✗ Zur Anwendung von Algorithmen häufig Übersetzung in alternative textuelle Repräsentation notwendig.

Objekt-orientierte Modelle sind konzeptuell zwischen den CSP-Modellen und den Graphen-basierten Modellen anzusiedeln. Im Unterschied zum CSP-Formalismus sind hier die Wertebereiche der Variablen keine unstrukturierten Mengen, sondern können aus anderen Wertebereichen zusammengesetzt sein (z.B. Referenzen, kartesische Produkte). Komponenten werden als Objekte bezeichnet und gehören zu Klassen, die in einer Klassenhierarchie angeordnet sind, ähnlich wie bei den Graphen-basierten Formalismen. Darüberhinaus ist über Referenzierung eine Komposition von Objekten möglich, wodurch auch physikalische Teil-Ganzes-Beziehungen modelliert werden können. Zum Teil werden verschiedene Arten von Komposition unterschieden, z.B. durch exklusiv oder gemeinsam genutzte Referenzen [KBG89]. Beeinflusst wurden solche Modelle stark von Entwicklungen im Bereich objekt-orientierter Datenbanken [FBC+87].

- ✓ In Programmiersprachen und der Datenmodellierung sehr weit verbreiteter, geläufiger Formalismus.
- ✓ Modellierung von zusätzlichen Komponentenattributen wird unterstützt (z.B. Preis).
- ✓ Gute Strukturierungsmöglichkeiten.

- ✗ Bislang nur geringfügige Anpassungen an Konfigurationsbesonderheiten.
- ✗ Spezifikation von Restriktionen weitgehend offengelassen.[4]

Allen Ansätzen zur Modellierung gemein ist die zum Großteil deklarative Beschreibung der Konfiguration. D.h. die Modelle beschreiben nur, was eine gültige Konfiguration ausmacht, nicht aber, wie diese zu generieren ist. Dadurch ist bei all diesen Modellen zur Lösung einer Konfigurationsaufgabe, also zur Bestimmung aller erforderlichen Komponenten zu einer

---

[4] Mailharro schlägt zur Vermeidung dieser Nachteile ein Modell vor, in dem objektorientierte Aspekte mit einem CSP-Formalismus verbunden werden [Mai98].

gegebenen Spezifikation, die Verwendung eines *kombinatorischen Suchalgorithmus* erforderlich. Bei den Produktionsregeln ist dies noch am wenigsten der Fall, da hier oft die Auswertung der Regeln bereits zur Lösung der Konfigurationsaufgabe führt, sofern eine solche existiert.

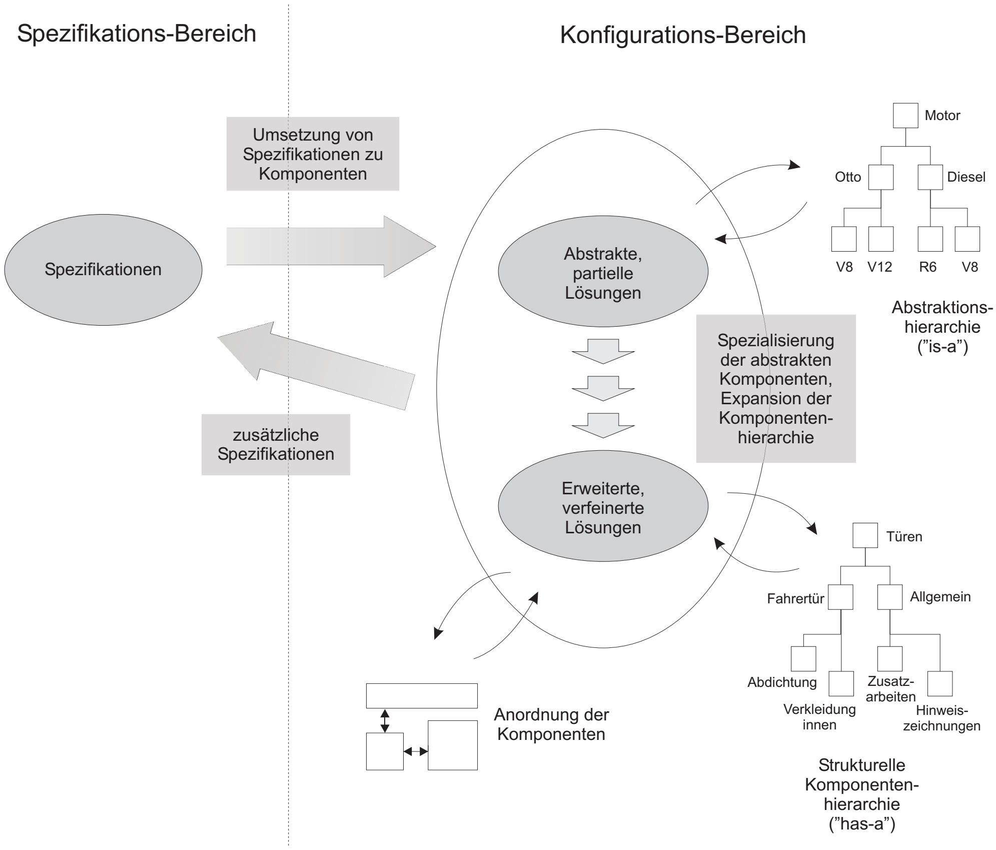


Abbildung 2.5: Abstrakte Beschreibung der Konfigurationsaufgabe (nach Stefik [Ste95]).

Stefik hat ein aufwändig ausgearbeitetes Modell der Konfiguration vorgestellt [Ste95], das aus der Analyse einer Vielzahl in der Literatur beschriebener und in der Praxis verwendeter Konfiguratoren entstanden ist. So ist es nicht verwunderlich, dass viele Aspekte der oben vorgestellten Modellierungsverfahren in diesem vereinheitlichten Modell wiederzufinden sind. Die Grundzüge von Stefiks Modells sind in Abbildung 2.5 dargestellt.

Für Stefik besteht ein Konfigurationsprozess aus mehreren Phasen. In einem ersten Schritt wird die Benutzerspezifikation in eine angenäherte Konfiguration übersetzt, die aus einer Menge abstrakter Komponenten, den *Schlüsselkomponenten*, besteht. Die Schlüsselkomponenten ergeben zusammen bereits ein rudimentäres Produkt mit der kompletten Funktionalität. In einem weiteren Schritt wird diese abstrakte Konfiguration dann spezialisiert und expandiert, wodurch konkrete Komponenten ausgewählt werden und deren Anordnung festgelegt wird. Dabei können auch neue Spezifikationsanforderungen entstehen.

Stefik unterscheidet zwischen Expansion und Spezialisierung der (vorläufigen) Konfiguration. Expansion ist ein Prozess, der sich auf eine Komponentenhierarchie beruft, die Teil-Ganzes-Beziehungen abbildet, und durch den komplexe Komponenten in ihre Bestandteile aufgelöst werden. Spezialisierung ersetzt abstrakte Komponenten mit einer bestimmten Basis-Funktionalität durch konkretere, funktional mindestens gleichwertige Komponenten. Die beiden Prozesse sind oft nicht unabhängig und im oben aufgeführten Modell der UND-ODER-Graphen sogar in derselben Datenstruktur zusammengefasst.

Das Hinzufügen von abhängigen, erforderlichen Komponenten ist oft ein eigenständiger Prozess, der auch Änderungen und Vervollständigung der Spezifikation mit beinhalten kann. Darüberhinaus wird in manchen Konfigurationssystemen oder -formalismen nicht zwischen Spezifikations- und Konfigurationssprache unterschieden: Spezifikationen werden dann als Teileanforderungen (meist in der Form von Schlüsselkomponenten) direkt in der Konfigurationssprache formuliert, und der Konfigurationsprozess beginnt mit einer solchermaßen gegebenen initialen Liste von Komponenten.

2.2.1 Anforderungen

Formalismen der Konfiguration haben sich mit den folgenden Problemfeldern zu beschäftigen [SF96]:

- Wissensrepräsentation,
- Effiziente Anwendung des Wissens,
- Mechanismen, um mit hohen Änderungsraten der Wissensdatenbank umzugehen.

Für eine präzise Wissensrepräsentation ist eine exakte (formale) Sprache unabdingbar, die es erlaubt, alle für die Konfiguration erforderlichen Konstrukte knapp und unmissverständlich wiederzugeben. Als Formalisierungssprache der oben erwähnten Modelle haben sich daher logische Sprachen weitgehend durchgesetzt. Diese variieren von der relativ einfach zu verstehenden Aussagenlogik über verschiedene Modal- und Beschreibungslogiken bis zur vollständigen Prädikatenlogik (meist nur erster Stufe). Auch wenn diese logischen Sprachen nicht direkt als Dokumentationssprachen verwendet werden, bietet es sich doch oft an, zumindest die Semantik der Konfigurationsmodelle in diesen zu definieren. Dies ist insbesondere zur Verifikation erforderlich, bei der Exaktheit und Eindeutigkeit der Sprache unverzichtbar sind.

Aus dem Konfigurationswissen sollen letztendlich hauptsächlich zwei Fragen beantwortet werden können:

1. Gibt es zu einer Kundenspezifikation eine passende, d.h. gültige, Konfiguration und wie sieht diese aus?

2. Wie kann die gewählte Konfiguration produziert werden, welche Prozesse und Teile werden dafür benötigt?

Um diese Fragen automatisiert und effizient behandeln zu können, sind effektive Algorithmen erforderlich, die auf dem zur Verfügung stehenden Konfigurationswissen arbeiten. Meist sind dies Inferenz- oder Suchalgorithmen, die aus dem logischen Produktwissen Konsequenzen ableiten können.

Erschwerend kommt hinzu, dass das Konfigurationswissen typischerweise einer hohen Änderungsrate unterworfen ist. Für den Datenerfasser stellt sich dann das Problem, diese Änderungen in die bestehende Dokumentation einarbeiten zu müssen. Ein umfassender Überblick über den Zustand der aktuellen Wissensbasis ist vonnöten, um eine widerspruchsfreie Datenbasis aufrechterhalten zu können. Ohne eine Strukturierung des Konfigurationswissens und eine systematische Herangehensweise ist dies bei komplexen Produkten praktisch unmöglich. Aber trotz aller Strukturierungs- und Qualitätssicherungs-Maßnahmen bleibt immer ein gewisses Risiko mit inkonsistenten Daten zu arbeiten. Automatisierte Prüfverfahren, die die Wissensdatenbank auf verschiedene Konsistenzkriterien hin untersuchen können, helfen, verbleibende Restfehler aufzuspüren.

Als Konsequenzen ergeben sich daraus die folgenden Anforderungen an einen Konfigurationsformalismus:

- Leicht erlernbare, möglichst intuitiv verständliche Modellierungssprache.

- Minimierung des Dokumentations- und damit Änderungsaufwands durch Vermeidung redundanter oder nicht adäquat nutzbarer Information. (Dazu ist eine Abwägung zwischen exakterer Modellierung und geringerem Dokumentationsaufwand erforderlich.)
- Strukturierte und modularisierte Datenhaltung, in der das für einen bestimmten Sachverhalt relevante Wissen kompakt dargestellt ist oder einfach generiert werden kann ("Lokalität des Wissens").
- Leicht anpassbar an Veränderungen: Kleine Produktänderungen erfordern auch nur kleine Änderungen an der Dokumentation.
- Algorithmisch effizient auswertbar. Dies beinhaltet sowohl effiziente logische Inferenzalgorithmen zur Auswertung als auch die Möglichkeit effizienter Konsistenztests zur Fehlersuche ("Sicherstellung von Funktionalität und Korrektheit").

2.2.2 Entwurfsentscheidungen zur Produktmodellierung

Soll eine Produktlinie neu dokumentiert werden, so sind eine Reihe von Entwurfsentscheidungen zur Modellierung zu treffen:

- **Trennung von Spezifikations- und Konfigurationssprache?** Falls der anvisierte Kundenkreis genügend technisches Wissen (oder die entsprechende Beratung) besitzt oder die Produkte aus einer überschaubaren Anzahl von Komponenten bestehen, kann unter Umständen auf eine Trennung in Spezifikations- und Konfigurationssprache verzichtet werden. Dadurch ergibt sich eine deutliche Vereinfachung der Dokumentation, da mit der Unterscheidung der Sprachen auch die Umsetzung zwischen den beiden Begriffswelten entfällt. Bei komplex strukturierten Produkten ist eine solche Vereinfachung nicht ratsam, da es den Kunden normalerweise nicht zuzumuten ist, ihre Wünsche in technisch detaillierten Kategorien zu benennen. Stattdessen sind funktionale Begriffe zur Spezifikation vorzuziehen (vgl. Mittal und Freyman [MF89]).
- **Rein deklaratives Konfigurationsmodell oder Auswertungsregeln und -algorithmus?** Bei rein deklarativen, häufig logischen Konfigurationsmodellen (z.B. CSP) ist typischerweise eine etablierte Semantik vorhanden, die sich auch auf den Modellierungsformalismus überträgt. Die Algorithmen zur Berechnung gültiger Konfigurationen sind

dann nicht konfigurationsspezifisch, sondern allgemeine kombinatorische Suchalgorithmen (z.B. A*, Davis-Putnam, ACL4). Im Gegensatz dazu ist bei der zweiten Modellierungsalternative die Semantik typischerweise in speziellen Algorithmen codiert, teils unter Zuhilfenahme von Produktionsregeln, teils auch ganz ohne diese. Systeme, die erstere Alternative wählen, sind oft spezialisierte Eigenentwicklungen und damit zunächst einmal auf den gewählten Anwendungsbereich bzw. dessen Produktgattung und deren konkrete Produktionsbedingungen festgelegt. Die Verwendung von Produktionsregeln mildert diese Problematik, löst sie aber nicht auf, da sich bisher keine allgemein akzeptierte Semantik für Produktionsregeln durchsetzen konnte: Teils wird die Semantik der Produktionsregeln entweder über den Regelauswertungsalgorithmus alleine festgelegt, teils ergibt sie sich erst unter Zuhilfenahme weiterer Auswertungsregeln, die z.B. die Präzedenz der Konfigurationsregeln kontrollieren (siehe XCON/R1).

- **Einsatz von Komponenten- und Abstraktionshierarchien?** Hierbei steht meist weniger die Frage im Vordergrund, ob solche Hierarchien überhaupt eingesetzt werden sollen, als vielmehr auf welche Art und Weise. Wie weiter oben angedeutet, sind zur Beschreibung der Produkthierarchie die unterschiedlichsten (Misch-)Formen denkbar: UND-ODER-Bäume, UML-Klassendiagramme oder einfache Gruppierungen, die eher dem menschlichen Verständnis als der maschinellen Verarbeitung dienen. Vor- und Nachteile der verschiedenen Varianten wurden bereits weiter oben diskutiert. Damit zusammenhängend zu sehen ist auch das Problem der

- **Form der Zuordnung von einschränkenden Nebenbedingungen (*constraints*), z.B. zu Komponenten?** Die Spezifikation von Constraints kann sowohl auf der Ebene der Spezifikationen als auch auf der Komponentenebene angebracht sein. Bei letzterer kann man einschränkende Constraints nur zwischen elementaren, nicht weiter unterteilbaren Elementen zulassen oder Restriktionen zwischen beliebigen Elementen der Komponentenhierarchie erlauben. Die erstgenannte Vorgehensweise hat den Vorteil, dass die Komponentenhierarchie nicht (explizit) im Konfigurationsmodell vorhanden sein muss ("flache Stückliste" ausreichend). Obendrein unterstützt dies die in Abschnitt 2.2.1 aufgestellte Minimalitätsforderung.

  Häufig auftretende Restriktionen sind gegenseitiger Ausschluss bzw. Unverträglichkeit, notwendiges Erfordern einer anderen Komponente sowie Selektion einer festgelegten Anzahl von Elementen aus einer Gruppe von Komponenten.

2.3 Fallstudien

Wir wollen nun anhand zweier Fallstudien aus der Automobilindustrie und der Medizintechnik das breite Spektrum der Produktdokumentationsverfahren näher beleuchten und die Umsetzung der zuvor aufgeworfenen Anforderungen und Entwurfsentscheidungen beispielhaft darstellen.

2.3.1 Mercedes-Benz-Fahrzeuge von DaimlerChrysler

DIALOG, das Produktdokumentationssystem für die Fahrzeuge der Mercedes-Benz-Linien von DaimlerChrysler, ist regelbasiert und wird in erster Linie zur automatischen Massenkonfiguration im Entwicklungs- und Produktionsbereich eingesetzt. DIALOG besteht aus einer funktions- und einer teileorientierten Schicht, der *Produktübersicht* und der *Produktstruktur* (oder *Stückliste*), die die Teiledaten enthält. Erstere wird zur Auftragsspezifikation verwendet, letztere dient unter anderem der Umsetzung eines Auftrags in die für diesen benötigten Teile.

Zentrale Begriffe im DIALOG-System sind *Baureihen*, *Codes* und *Regeln*. Baureihen dienen der Klassifizierung von Fahrzeugen und fassen Modelle mit ähnlicher Charakteristik zusammen. In DIALOG wird jede Baureihe (z.B. C-Klasse, E-Klasse, etc.) weitgehend eigenständig dokumentiert. Codes bezeichnen entweder Ausstattungsmerkmale bzw. Verkaufsoptionen (*Ausstattungscodes*), oder sie werden zu internen Auftragsverarbeitungs- und Produktionssteuerungszwecken eingesetzt (*Kontroll- bzw. Steuercodes*). Ein Kundenauftrag besteht aus einer Baureihenwahl zusammen mit einer Liste von Ausstattungscodes.

Regeln treten in drei verschiedenen Ausprägungen mit unterschiedlicher Funktionalität auf: Als *Baubarkeitsregeln* (diese auch in der Variante der *pauschalen Coderegeln*), *Zusteuerungsregeln* oder teilebezogene *Coderegeln*, letztere entweder in kurzer oder langer Darstellung. Die drei Arten von Regeln dienen der Auftragsverarbeitung von der Kundenspezifikation bis zur Ermittlung des (vollständig expandierten) Teilebedarfs. Die Bearbeitung findet dabei, entsprechend den drei Regeltypen, in drei aufeinanderfolgenden Phasen statt: (1) Zusteuerung, (2) (Baubarkeits-)Prüfung und (3) Teilebedarfsermittlung, wie in Abbildung 2.6 dargestellt.

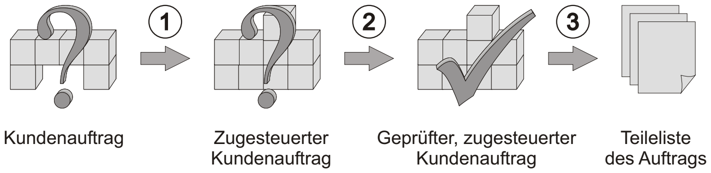


Abbildung 2.6: Auftrags-Verarbeitung in Dialog.

Bei den Regeln handelt es sich um Produktionsregeln, die alle zuerst eine aussagenlogische Formel auswerten und abhängig vom Ergebnis eine bestimmte Aktion ausführen. Die aussagenlogische Formel enthält als Variablen die Dokumentations-Codes. Der Auftrag wird als Variablenbelegung dieser Codes verstanden: wählt der Kunde eine bestimmte Ausstattung, so ist der entsprechende Code mit wahr bewertet, ansonsten mit falsch.

Wir werden nun die drei Phasen der Konfiguration, die zugehörigen Regeln und die dazu verwendete Dokumentationsstruktur vorstellen.

Baubarkeitsprüfung

Die Baubarkeitsprüfung basiert auf Baubarkeitsregeln. Jede Baubarkeitsregel ist einem Code zugeordnet und gibt Bedingungen an, unter denen dieser Code in einem Auftrag verwendet werden darf. Die (Prüf-)Bedingung ist eine aussagenlogische Formel und wird auch *Baubarkeitsbedingung* der Regel genannt.

In Dialog gibt es zwei Arten von Baubarkeitsregeln. Neben den gewöhnlichen Baubarkeitsregeln, die baureihenspezifisch sind, gibt es auch solche, die über alle Baureihen hinweg Gültigkeit besitzen, so genannte *pauschale Baubarkeits-* bzw. *Coderegeln*. Letztere fassen übergreifende Gemeinsamkeiten zusammen. Diese liegen in Dialog zu einem beträchtlichen Teil auch in der aussagenlogischen Codierung endlicher Variablen-Wertebereiche begründet. Schließen sich beispielsweise n verschiedene Ausstattungspakete generell gegenseitig aus, so kann dies über n pauschale Codebedingungen festgeschrieben werden. Nahe liegender wäre eine Codierung mittels Variablen mit endlichen Wertebereichen, wie dies beispielsweise im CSP-Bereich üblich ist. In unserem Beispiel entsprächen dann die n Ausstattungspakete n verschiedenen Werten einer Variablen. Eine spezielle Codierung des gegenseitigen Ausschlusses entfiele somit.

Zur Auswertung einer Baubarkeitsregel wird deren Baubarkeitsbedingung mit der durch den Auftrag sich ergebenden Variablenbelegung evaluiert. Ergibt diese Evaluierung, dass die Formel erfüllt ist, so ist der Baubarkeitstest für diese Regel (und den zugehörigen Code) bestanden, ansonsten ist der Code nicht baubar. In einem *gültigen* oder *baubaren* Auftrag muss jeder im (eventuell durch die Zusteuerung erweiterten) Auftrag vorkommende Code baubar sein. Einem Code können mehrere Baubarkeitsbedingungen zugeordnet sein, sowohl baureihenspezifische, als auch pauschale. Die pauschalen Baubarkeitsregeln eines Codes müssen, sofern vorhanden, alle erfüllt sein. Bei den baureihenspezifischen Regeln hängt die logische Verknüpfung der Regeln von einer zusätzlich vorhandenen Strukturierung der Regelmenge in Gruppen, Positionen und Varianten ab. Diese Strukturierung ist in Abbildung 2.7 dargestellt.

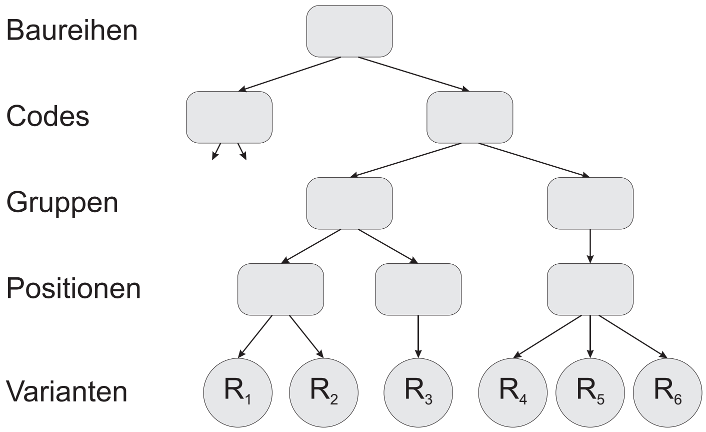


Abbildung 2.7: Struktur der baureihenspezifischen Baubarkeitsregeln.

Damit ein Code baubar ist, muss demnach für jede seiner Positionen die Baubarkeitsregel mindestens einer Variante erfüllt sein. Sowohl Gruppen als auch Varianten können vom DIALOG-Anwender frei festgelegt und benannt werden. Die Strukturierung der Regeln dient also in erster Linie der Einteilung der Regeln nach verschiedenen vom Benutzer vorgegebenen Aspekten und damit der Modularisierung. Eine der bei DaimlerChrysler verwendeten Gruppen fasst beispielsweise länderspezifische Beschränkungen zusammen.

Ein Auszug der baureihenspezifischen Baubarkeitsregeln der Baureihe C202 (in der Ausführungsart[5] FW) ist in der nachfolgenden Tabelle wiedergegeben:

Code	Grp.	Pos.	Var.	Lenk.	Baubarkeitsbedingung
494	CLA	1020	0001	L	$M104 \wedge \neg(212 \vee 248 \vee 798 \vee 819 \vee 910 \wedge \neg(460 \vee 957))$
494	CLA	1020	0002	R	$M112 \wedge M28 \wedge \neg(772 \vee 774)$
494	CLA	1020	0003	–	$M111 \wedge M23 \wedge \neg M001 \wedge \neg(280 \wedge \neg 460 \vee 654 \vee 772)$
400	CAU	0400	0001	–	$(040A \vee 740A) \wedge \neg 808$
423	CAG	0423	0007	R	M112

Die in dieser Tabelle vorkommende Spalte “Lenk.” spiegelt eine weitere Besonderheit der DIALOG-Dokumentationsmethode wider und ergibt sich aus einer Eigenschaft, die für die Automobilbranche charakteristisch ist: Eine Wahlmöglichkeit mit ungewöhnlich weitreichendem Einfluss auf weitere Konfigurationsoptionen und Teilebedarf ist die Wahl zwischen Links- und Rechtslenker-Ausführung. Da diese so grundsätzlich ist, haben die beiden diesen Sachverhalt beschreibenden Codes L und R eine Sonderrolle inne: Alle Baubarkeits-Varianten können ein zusätzliches Lenkungsattribut besitzen. Nur wenn dieses, sofern vorhanden, mit der vom Kunden spezifizierten Lenkungsausführung übereinstimmt, wird die Variante aktiv und die Regel damit berücksichtigt. Ergibt sich auf Grund der Filterwirkung der Lenkungsattribute, dass alle Varianten einer Position deaktiviert werden, so ist der zugehörige Code in dieser Lenkungsvariante nicht baubar (entsprechend der üblichen Definition der leeren Disjunktion).[6]

Ein weiterer Sonderfall tritt ein, wenn für einen Code überhaupt keine Positionen, Varianten und Baubarkeitsregeln angegeben sind. Auch dann ist der Code nicht baubar.

Gänzlich von der Baubarkeitsprüfung ausgenommen sind die beiden Lenkungscodes L und R. Codes, die angeben, in welches Land das Fahrzeug geliefert werden soll (*Ländercodes*), sind bei fehlender Baubarkeitsbedingung, im Gegensatz zu der Regelung für die sonstigen Codes, standardmässig baubar.

Häufig sind es solche Sonderfälle, die bei der Regelerstellung und -wartung übersehen werden und dann zu Fehlern in der Produktdokumentation führen.

---

[5] Zur Begriffsdefinition siehe Seite 36.

[6] Die Spezifikation des Algorithmus ist hier etwas unklar. Eine andere Interpretationsmöglichkeit wäre, dass ein Code nur dann nicht baubar ist, wenn bereits ohne die Filterwirkung der Lenkungsattribute keine Varianten vorhanden sind.

```
ALGORITHMUS: Baubarkeit
EINGABE:  $A = \{x_1, \ldots, x_n\} \subseteq \mathrm{Codes}$,   // (zugesteuerter) Auftrag
          $l \in \{\mathrm{L}, \mathrm{R}\}$                      // Lenkung
AUSGABE:  $b \in \{\textbf{true}, \textbf{false}\}$              // Auftrag baubar?

$b$ := true
for each $c \in A$ do
    $b'$ := true;
    for each $g \in \mathrm{Gruppen}(c),\ p \in \mathrm{Positionen}(c, g)$ do
        $b''$ := false;
        for each $v \in \mathrm{Varianten}(c, g, p, l)$ do
            if $\mathrm{eval}(\mathrm{BRegel}(c, g, p, v), A)$ = true then
                $b''$ := true
        if $b''$ = false then $b'$ := false;
    if $b'$ = false $\vee$ $\mathrm{eval}(\mathrm{PCRegel}(c), A)$ = false then
        $b$ := false
```

Abbildung 2.8: Algorithmus Baubarkeit in Dialog.

In Abbildung 2.8 ist der Pseudo-Code des in Dialog implementierten Baubarkeits-Algorithmus wiedergegeben. Die Funktionen $\mathrm{Gruppen}(c)$, $\mathrm{Positionen}(c, g)$ und $\mathrm{Varianten}(c, g, p, l)$ selektieren dabei die entsprechenden Strukturen aus Abbildung 2.7, wobei letztere zusätzlich eine Lenkungseinschränkung l mit berücksichtigt. $\mathrm{BRegel}(c, g, p, v)$ bezeichnet die baureihenspezifische Baubarkeitsbedingung von Code c in Gruppe g an Position p in Variante v. $\mathrm{PCRegel}(c)$ steht für die Konjunktion aller pauschalen Coderbedingungen von Code c, und $\mathrm{eval}(F, A)$ wertet die Formel F für Auftrag A, d.h. unter der durch die charakteristische Funktion von A gegebenen Variablenbelegung, aus.

Zusteuerung

Die Zusteuerung erweitert die Kundenspezifikation um implizite Codes. Teilweise handelt es sich dabei um zusammenfassende Codes (z.B. für Ländergruppen), teils um die Expansion von Ausstattungspaketen in die enthaltenen Sonderausstattungen. Auch die automatische Auswahl von Standardwerten für vom Kunden nicht spezifizierte Merkmale und die Auftragserweiterung um betriebsinterne Steuercodes sind Aufgabe der Zusteuerung.

Der Zusteuerungsprozess wird durch baureihenspezifische *Zusteuerungsregeln*, die jeweils einem Code zugeordnet sind und eine *Zusteuerungsbedingung* enthalten, gesteuert. Eine Zusteuerungsregel kann höchstens den ihr zugeordneten Code dem Auftrag hinzufügen, dies allerdings auch nur, wenn bestimmte Bedingungen erfüllt sind. Die Zusteuerungsregeln sind genau wie die baureihenspezifischen Baubarkeitsregeln in Gruppen, Positionen und Varianten angeordnet und werden auch in derselben Struktur ergänzend zu den Baubarkeitsregeln abgelegt. Dadurch gehört zu jeder Zusteuerungsregel auch eine baureihenspezifische Baubarkeitsregel, die Umkehrung gilt allerdings meist nicht. Zur Aktivierung einer Zusteuerungsregel muss neben der Zusteuerungsbedingung auch die zugehörige baureihenspezifische Baubarkeitsbedingung und alle pauschalen Codebedingungen des zuzusteuernden Codes erfüllt sein. Dadurch soll sichergestellt werden, dass durch die Zusteuerung kein baubarer Auftrag ungültig gemacht wird. Sind alle diese Bedingungen erfüllt, so wird der Code zur Auftragsspezifikation hinzugenommen.

Einige Beispiele für Zusteuerungsregeln, wiederum für die Baureihe C202 in der Ausführungsart FW, sind der folgenden Tabelle zu entnehmen:

Code	Grp.	Pos.	Var.	L.	Prio.	Zusteuerbedingung
423	CAG	0423	0001	–	CG	$494 \vee 498 \vee 625 \wedge (M111 \wedge M23 \wedge \neg M001 \vee M605) \vee 023 \wedge 817L \vee Y94$
423	CAG	0423	0004	L	CG	$472 \vee 494 \vee 498 \vee 828 \vee 032 \wedge \neg 817L \vee Y94) \vee P14$
423	CAG	0423	0007	R	BG	M112

Idealerweise sollte der Zusteuerungsprozess so lange ablaufen, bis sich keine Änderungen am Auftrag mehr ergeben. Der tatsächlich implementierte Zusteuerungsalgorithmus führt allerdings nur eine feste Zahl von Zusteuerungsläufen, d.h. Runden, in denen jede Zusteuerungsregel ein Mal betrachtet wird, durch. Darüberhinaus ist eine zweistufige Priorisierungssteuerung der Regeln implementiert, die zuerst in drei initialen Zusteuerungsläufen nur Regeln mit hoher Priorität ("CG", Code-gebunden) betrachtet, und danach einen (BG-)Lauf durchführt, in dem Regeln mit niedriger Priorität ("BG", Baureihen-gebunden) ausgewertet und deren Codes zugesteuert werden.

Der Pseudo-Code für diesen Zusteuerungsalgorithmus ist in Abbildung 2.9 wiedergegeben. Dabei sind $\mathrm{Gruppen}$, $\mathrm{Positionen}$ und $\mathrm{Varianten}$ als Selektionsfunktionen zu verstehen, die entsprechend Abbildung 2.7 die jeweiligen Elemente der Zusteuerungs- und Baubarkeitsstruktur auswählen. Bei der Auswahl der Varianten ist durch die letzten beiden Argumente der Selektionsfunktion $\mathrm{Varianten}(c, g, p, l, o)$ sichergestellt, dass auch die Lenkungseinschränkung l und die Priorisierung o (CG der BG) berücksichtigt werden. $\mathrm{ZRegel}(c, g, p, v)$ und $\mathrm{BRegel}(c, g, p, v)$ bezeichnen wiederum die Zusteuerungs- und Baubarkeitsregeln von Code c in Gruppe g an Position p in Variante v, $\mathrm{PCRegel}(c)$ steht für die Konjunktion aller pauschalen Coderegeln von Code c, und $\mathrm{eval}(F, A)$ wertet Formel F für Auftrag A aus.

```
ALOGRITHMUS: Zusteuerung
EINGABE:  $A = \{x_1, \dots, x_n\} \subseteq \mathrm{Codes}$,     // Kundenauftrag
          $l \in \{\mathrm{L}, \mathrm{R}\}$                       // Lenkung
AUSGABE:  $A' = \{x_1, \dots, x_n, \dots, x_{n+m}\}$    // zugesteuerter Auftrag

for $i = 1$ to $\mathrm{max\_cg}$ do                     // CG-Läufe
   for each $c \in \mathrm{Codes}, g \in \mathrm{Gruppen}(c)$,
         $p \in \mathrm{Positionen}(c, g), v \in \mathrm{Varianten}(c, g, p, l, \mathrm{CG})$ do
      if $\mathrm{eval}(\mathrm{ZRegel}(c, g, p, v), A) = \mathbf{true} \wedge$
        $\mathrm{eval}(\mathrm{BRegel}(c, g, p, v), A) = \mathbf{true} \wedge$
        $\mathrm{eval}(\mathrm{PCRegel}(c), A) = \mathbf{true}$ then
          $A := A \cup \{c\}$
for $i = 1$ to $\mathrm{max\_bg}$ do                     // BG-Läufe
   for each $c \in \mathrm{Codes}, g \in \mathrm{Gruppen}(c)$,
         $p \in \mathrm{Positionen}(c, g), v \in \mathrm{Varianten}(c, g, p, l, \mathrm{BG})$ do
      if $\mathrm{eval}(\mathrm{ZRegel}(c, g, p, v), A) = \mathbf{true} \wedge$
        $\mathrm{eval}(\mathrm{BRegel}(c, g, p, v), A) = \mathbf{true} \wedge$
        $\mathrm{eval}(\mathrm{PCRegel}(c), A) = \mathbf{true}$ then
          $A := A \cup \{c\}$
$A' := A$
```

Abbildung 2.9: Algorithmus Zusteuerung in DIALOG.

Produktstruktur und Teilebedarfsermittlung

Die Produktstruktur wird in DIALOG für jede Baureihe getrennt in einer *Baureihen-Gesamtstückliste* abgelegt. Diese enthält neben den Teilen selbst auch Informationen zur Produktstruktur. Sie ist in Module untergliedert, wobei jedes Modul alle möglichen Teile eines funktional und geometrisch weitgehend eigenständigen Teilzusammenbaus umfasst. Module sind weiter in Positionen unterteilt. In einer Position sind all die Teile zusammengefasst, die alternativ an einer räumlichen (geometrischen) Position im Fahrzeug eingebaut werden können und die eine vergleichbare Funktionalität bereitstellen. Die Alternativen werden *Positionsvarianten* genannt. Jeder Positionsvariante ist eindeutig eine Teilenummer und eine Regel, die so genannte *Coderegel*, zugeordnet. Die Coderegel gibt an, unter welchen Bedingungen eine Variante erlaubt ist und somit verbaut werden darf. Die Coderegel gibt es in einer kurzen und einer langen Ausführung. Die kurze Coderegel einer Variante kann nur zusammen mit den anderen Positionsvarianten derselben Position interpretiert werden, da sie die anderen Varianten implizit als Ausschluss mit enthält. In der langen Coderegel, die aus der kurzen mittels eines speziellen Algorithmus generiert wird, sind die Ausschlüsse explizit gemacht. Daher kann diese direkt mit der aussagenlogischen Standardsemantik interpretiert werden. Die langen Coderegeln werden bei Bedarf und unter Berücksichtigung des Termins generiert.[7] Wir werden weiter unten noch genauer auf die Bildung der langen Coderegel eingehen. Die Struktur der Baureihen-Gesamtstückliste ist in Abbildung 2.10 dargestellt.

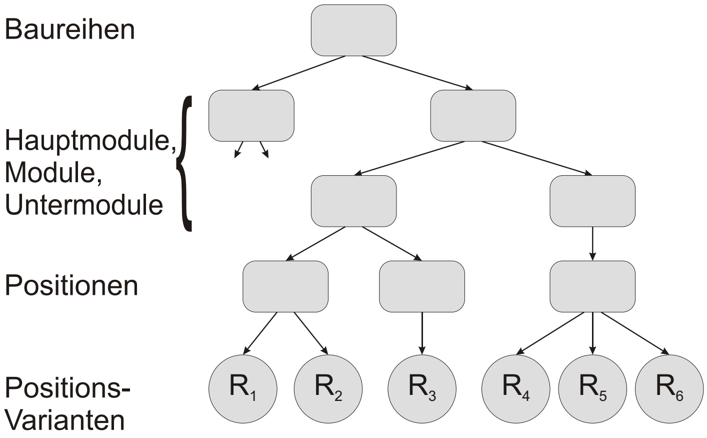


Abbildung 2.10: Produktstruktur, Baureihen-Gesamtstückliste.

Stellt man das DIALOG-Dokumentationsmodell in Beziehung zu den oben angeführten Konfigurationsmodellen, so findet man auf der Spezifikationsebene die Produktionsregeln wieder. In der Darstellung der Produktstruktur (entsprechend Stefiks Abstraktions- und Komponentenhierarchien) kann man konzeptionelle Ähnlichkeiten zu den UND-ODER-Graphen erkennen.

---

[7] Siehe dazu auch weiter unten die Ausführungen zu terminlichen Aspekten in der Dokumentation für die Produktion (S. 35).

In der graphischen Produktstruktur von Abbildung 2.10 sind sämtliche Modulknoten UND-Knoten und bilden daher die Komponentenhierarchie. Nur auf der obersten Ebene der Baureihen und der untersten Ebene der Positionen, die nun als ODER-Knoten zu verstehen sind, ist eine Abstraktionsschicht vorhanden, in der für jede Position ein konkretes Teil anhand der Positionsvarianten ausgewählt werden muss.

Die Entwurfsentscheidungen aus Abschnitt 2.2.2 betrachtend, stellt man weiter fest, dass DIALOG zwischen Spezifikations- und Konfigurationssprache unterscheidet. Erstere ist durch die Codes und Regeln der Produktübersicht geprägt, letztere durch die Modulstruktur und die Teiledaten (einschließlich der Teilenummern). Eine Umsetzung zwischen den Sprachen findet durch die Coderegeln der Produktstruktur statt.

Restriktionen können direkt nur auf der Spezifikationsebene mittels Baubarkeits- und Zusteuerungsregeln angegeben werden. Um auf der Ebene der Teile oder Positionsvarianten Abhängigkeiten auszudrücken, muss man sich spezieller Konstrukte bedienen. Wir wollen dies für zwei häufig vorkommende Restriktionstypen exemplarisch vorführen:[8]

1. **Gegenseitiger Ausschluss zweier Positionsvarianten:** Nehmen wir dazu an, die Positionsvarianten seien bereits dokumentiert, so dass die Formeln R_1 und R_2 der Coderegeln beider Positionsvarianten bereits vorhanden sind. Durch Abändern von R_1 in $R_1 \wedge F$ und von R_2 in $R_2 \wedge \neg F$ für eine beliebige Formel F kann dann der gegenseitige Ausschluss der beiden Positionsvarianten sichergestellt werden. Dies lässt sowohl die beiden Extreme zu, dass eine Positionsvariante die Verwendung der anderen bestimmt (z.B. durch Wahl von $F = R_1$), als auch beliebige Zwischenstufen, wozu auch das Einführen einer neuen Variablen zu rechnen ist.

2. **Eine Positionsvariante erfordert eine andere:** Angenommen, eine Positionsvariante v_1 mit Coderegel R_1 erfordert eine andere Positionsvariante v_2 mit Coderegel R_2. Ändert man Coderegel R_1 in $R_1 \wedge R_2$, so ist sichergestellt, dass mit v_1 immer auch v_2 vohanden ist.

Restriktionen auf höheren Ebenen, also z.B. der Modulebene, sind im DIALOG-Modell nicht sinnvoll, da alle inneren Knoten nicht zur Abstraktions-, sondern zur Komponentenhierarchie gehören und somit alle inneren Knoten in einem korrekt konfigurierten Fahrzeug auftreten müssen.

---

[8] Die folgende Argumentation basiert auf der Coderegel in langer Darstellung.

Teileauswahlregeln und Disambiguierung

Wir wollen nun noch ausführlicher auf die Unterscheidung zwischen kurzer und langer Coderegel eingehen. Hintergrund dieser Unterscheidung ist das von regelbasierten Expertensystemen bekannte Problem sich widersprechender Regeln (siehe z.B. [WS92]). Ein klassisches Beispiel zur Verdeutlichung der Problematik liefert das folgende intuitiv verständliche, aus drei *wenn-dann*-Regeln bestehende Expertensystem:

$$\begin{aligned} \text{Vogel}(x) &\longrightarrow \text{fliegt}(x) \\ \text{Vogel}(x) \wedge \text{Pinguin}(x) &\longrightarrow \neg\text{fliegt}(x) \\ \text{Vogel}(x) \wedge \text{Pinguin}(x) \wedge \text{inFlugzeug}(x) &\longrightarrow \text{fliegt}(x) \end{aligned}$$

Jede dieser Regeln ist spezieller als die vorhergehende, so dass beispielsweise mit der zweiten Regel immer auch die erste angewandt werden kann, woraus dann aber widersprüchliche Konsequenzen folgen. Um dies zu vermeiden werden Regeln häufig priorisiert, wobei spezielleren Regeln eine höhere Priorität zugewiesen wird. Dies, zusammengenommen mit der Meta-Regel, dass speziellere Regeln allgemeinere deaktivieren (oder alternativ, dass nur eine Regel aus einer Regelgruppe ausgeführt werden darf), erlaubt dann die Auflösung von widersprüchlichen Regeln.

In DIALOG hat das Vorhandensein von zwei Arten von Coderegeln zum Ziel, solche widersprüchliche Regeln zu vermeiden. Insbesondere soll von den Positionsvarianten einer Position immer genau (oder höchstens) eine gültig sein, d.h. jeder baubare Auftrag soll genau (oder höchstens) eine Variante auswählen. Darüberhinaus soll durch die Verwendung der kurzen Coderegeln der Dokumentations- und Änderungsaufwand verringert und eine größere Transparenz und Übersichtlichkeit gewährleistet werden.

Der Algorithmus zur Umsetzung von kurzen in lange Coderegeln ist in DIALOG in COBOL implementiert. Er basiert im wesentlichen auf dem Ausschluss nicht subsumierter Literalkonjunktionen anderer Coderegeln zur Disambiguierung.

Dazu betrachten wir eine Stücklistenposition mit n Positionsvarianten v_i und zugehörigen kurzen Coderegeln r_i ($1 \leq i \leq n$). Da kurze Coderegeln immer durch Anwendung der (auch für diese gültigen) Distributivgesetze in disjunktive Normalform gebracht werden können, ist

$$r_i = q_{i,1} \vee \cdots \vee q_{i,m_i}$$

für ein m_i und Literalkonjunktionen $q_{i,j}$. Wir definieren nun die Mengen

$Q_{i,j}$ der von $q_{i,j}$ nicht implizierten Literalkonjunktionen anderer Positionsvarianten:

$$Q_{i,j} = \{q_{k,l} \mid 1 \leq k \leq n, k \neq i, 1 \leq l \leq m_k \text{ und } q_{i,j} \text{ impliziert nicht } q_{k,l}\}$$

Dann ist die aus r_i generierte lange Coderegel R_i definiert als

$$R_i = \bigvee_{1 \leq j \leq m_i} \left(q_{i,j} \wedge \bigwedge_{q \in Q_{i,j}} \neg q \right)$$

Als Beispiel betrachten wir drei Positionsvarianten sowie deren kurze und lange Coderegeln:[9]

	r_i	R_i
v_1	$\top$	$\bar{x}\bar{y}$
v_2	y	$\bar{x}y$
v_3	$x + xy$	$xy + x\bar{y} \equiv x$

In der kurzen Darstellung ist erkennbar, dass v_1 die Standardvariante ist, die ausgewählt werden soll, wenn keine der anderen spezielleren Varianten passt. Anhand der langen Coderegel ist des weiteren zu erkennen, dass v_2 nur dann gewählt wird, wenn die speziellere Regel v_3 nicht passt, so dass im Fall $x = y = \textbf{true}$ die letzte Variante v_3 gewählt würde.

Zu beachten ist noch, dass für die kurzen Coderegeln einige Gesetze der Booleschen Algebra, wie z.B. das Absorptionsgesetz $x = x + xy$, nicht gültig sind. Durch unbedachte Anwendung dieser Gesetze können so Dokumentationsfehler entstehen.

Als einfachere Alternative zu der in DIALOG gewählten Disambiguierung wäre auch eine simple Priorisierung der Regeln über eine Ordnung der Varianten denkbar.

DIALOG **für die Produktion**

In der Produktion spielen terminliche Aspekte eine große Rolle. So muss beispielsweise die Verfügbarkeit von Teilen, Maschinen und Personal für die Herstellung einer bestimmten Produktvariante sichergestellt sein. Darüberhinaus sind Änderungen am Produkt auch nach dem ersten Entwurf recht häufig, das gleiche gilt auch für dessen Produktionsbedingungen. Ursachen

---

[9] Wir verwenden hier eine abgekürzte Schreibweise für Formeln: $\bar{x}$ bezeichnet die Negation von x, xy die Konjunktion und $x + y$ die Disjunktion von x und y.

für solche Änderungen können z.B. die Überarbeitung oder Neuentwicklung von Komponenten, wechselnder Teilebezug, Produktionsverlagerung, oder anderweitige Umstellungen in der Produktion sein. Eine passende Dokumentationsmethode sollte all diese Änderungen umfassen.

In DIALOG sind daher spezielle Erweiterungen für die Produktion vorhanden, mit deren Hilfe sich, auch für jedes Werk unterschiedliche, terminliche Aspekte mit einbeziehen lassen. Dazu ist jede Regel mit einem (halboffenen) Gültigkeits-Zeitintervall $I = [t_\alpha, t_\omega)$ mit $t_\alpha \leq t_\omega$ und speziellen Start- und Stoppcodes CC_α und CC_ω ausgestattet. Eine Regel wird nur dann verwendet, wenn sie zum betrachteten Zeitpunkt *aktiviert* ist. Die Aktivierung ist für einen Auftrag A und einen Zeitpunkt t wie folgt geregelt:

1. Ist $t < t_\alpha$, so ist die Regel nur dann aktiviert, wenn der Startcode CC_α im Auftrag A vorkommt, der Stoppcode CC_ω aber nicht.
2. Ist $t_\alpha \leq t < t_\omega$, so ist die Regel aktiviert, wenn der Stoppcode CC_ω nicht im Auftrag A vorkommt.
3. Nach t_ω (d.h. $t \geq t_\omega$) ist die Regel nie aktiviert.

Die Steuercodes können also dazu verwendet werden, das Gültigkeitsintervall außer Kraft zu setzen. Der Startcode kann dabei den Anfangstermin zeitlich vorverlegen, der Stoppcode den Endtermin. Ein Aufschub der Termine ist nicht möglich.

Mit Hilfe der Zeitintervalle und Steuercodes können dann auch komplexe Sachverhalte wie z.B. überlappender Teiletausch abgebildet werden [SK01].

Statistische Größen

Einige charakteristische Größen der DIALOG-Fahrzeugdokumentation sind in Tabelle 2.1 zusammengestellt. In der ersten Spalte ist die Baureihenbezeichnung und die Ausführungsart (AA) wiedergegeben. Baureihen werden intern nochmals in Ausführungsarten (z.B. Limousine, Kombi, Coupé) untergliedert. Die Ausführungsart hat in DIALOG auf die Produktdaten eine Filterwirkung analog der Lenkungsausführung. Die Dokumentationsunterschiede zwischen verschiedenen Ausführungsarten sind eher gering. Allerdings liegen der späteren Verifikation die Daten der einzelnen Baureihen-Ausführungsarten zugrunde, so dass zugunsten einer besseren Vergleichbarkeit die Daten dieser Tabelle auch schon nach Ausführungsarten getrennt ausgewiesen werden.

Baureihe/AA	Codes	Baubark.-Regeln	Zust.-Regeln	Module	Code-Regeln
C129/FR	513	685	174	613	13708
C140/FC	397	516	137	605	13170
C140/FV	463	660	182	673	22809
C140/FW	459	652	182	687	19463
C163/FW	313	408	139	523	8746
C168/FW	494	706	215	564	11257
C169/FV	59	64	15	300	1694
C169/FW	59	64	15	333	1875
C170/FR	441	790	230	551	9174
C171/FR	205	236	87	384	3555
C202/FS	606	951	202	709	16773
C202/FW	692	952	197	716	18508
C203/FCL	369	444	114	573	6682
C203/FS	431	524	131	634	8969
C203/FW	529	627	129	634	9078
C208/FA	426	627	182	617	12030
C208/FC	491	732	196	647	11909
C209/FA	307	391	113	474	4749
C209/FC	346	430	114	505	5053
C210/FS	586	849	211	740	23324
C210/FVF	350	477	118	600	10573
C210/FW	653	993	227	777	29136
C211/FS	239	263	44	581	8883
C211/FW	318	364	44	613	9688
C215/FC	388	487	133	570	7771
C220/FV	549	724	212	652	13269
C220/FW	541	681	176	647	12470
C230/FR	334	431	135	500	5610
C250/FV	127	140	39	410	3155
C250/FW	127	140	39	434	3308
C638/FKA	535	576	60	433	6393
C638/FKB	515	544	41	432	6077
C638/FVK	463	490	29	405	6020
D1119/M20	38	60	12	76	2717
D1119/M23	47	67	12	75	2871

Tabelle 2.1: Charakteristika der Produktdaten einiger Mercedes-Benz Fahrzeugbaureihen.

In der zweiten Spalte ist die Anzahl der in der Produktübersicht verwendeten Codes angegeben, wobei hier nur die zur Gruppierung (siehe Abbildung 2.7) verwendeten Codes berücksichtigt wurden. Insbesondere wurden also die Lenkungs- und Ländercodes nicht mitgezählt.

Aus den nächsten beiden Spalten kann die Anzahl der Baubarkeits- und Zusteuerungsregeln abgelesen werden, darauf folgen die Anzahl der Stücklistenmodule sowie die Anzahl der Positionsvarianten der Gesamtstückliste (die mit der Anzahl der Coderegeln der Stückliste übereinstimmt).

2.3.2 Magnetresonanztomografen von SIEMENS Medizintechnik

Wir betrachten nun als zweites Beispiel die formale Produktdokumentation der SIEMENS Magnetresonanztomografen (MR). Dort wird die Dokumentation neben der Konfiguration der Geräte selbst auch zur Vollständigkeitsprüfung der für jedes Gerät individuell zusammengestellten Betriebs- und Wartungshandbücher verwendet.

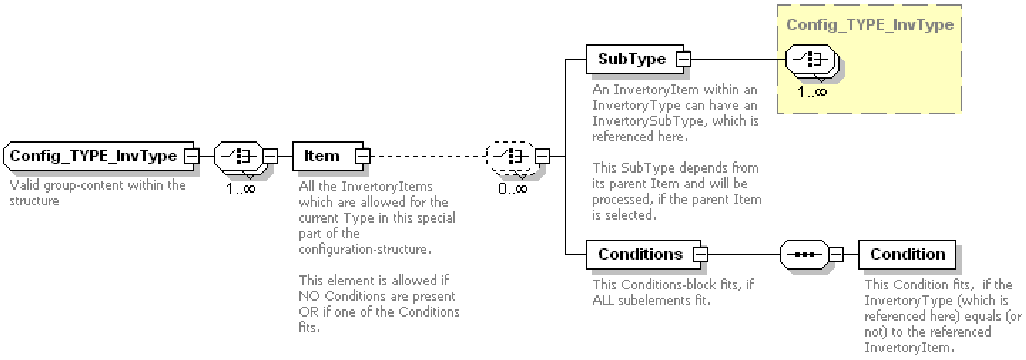


Abbildung 2.11: XML-Schema der SIEMENS MR-Geräte-Konfiguration.

Die gesamten Produktdaten werden bei SIEMENS in einer XML-Datenbank abgelegt. Konfigurationsrelevante Auszüge aus dem XML-Schema, welches das zugrundeliegende Datenmodell beschreibt, sind in Abbildung 2.11 wiedergegeben. In der SIEMENS XML-Dokumentation wird die Struktur der MR-Geräte explizit durch einen Baum beschrieben, der deren hierarchischen Aufbau wiederspiegelt. Dieser Konfigurationsbaum enthält sowohl die Abstraktions- als auch die Komponentenhierarchie. Man kann ihn daher auch als einen UND-ODER-Graphen (genauer: UND-ODER-Baum) verstehen, wobei hier die zusätzliche Einschränkung gilt, dass sich auf jedem Pfad die UND- und ODER-Knoten abwechseln müssen. Die SIEMENS-Nomenklatur spricht von Gruppen- (ODER) und Elementknoten (UND). Im XML-Schema sind diese unter den Begriffen *Type* bzw. *SubType* und *Item* zu finden. Ein Gruppenknoten dient der Zusammenfassung mehrerer funktional ähnlicher oder äquivalenter Elemente, zwischen denen der Kunde auswählen kann. Ein Elementknoten stellt eine solche Auswahlmöglichkeit in der Form einer (eventuell abstrakten) Komponente dar, die ihrerseits die weitere Konfiguration darin enthaltener Unterkomponenten (in der Form von Gruppenknoten) erforderlich machen kann. Im CSP-Sinn kann man die Gruppenknoten auch als Variablen der Konfiguration auffassen, die direkten Kindknoten sind dann die möglichen Variablenwerte, zusammengenommen also der Wertebereich dieser Variablen.

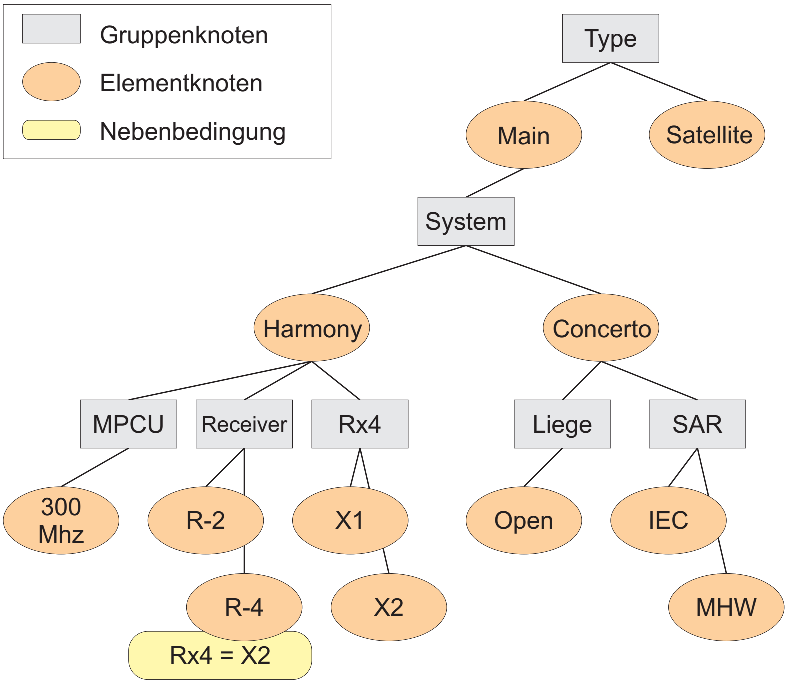


Abbildung 2.12: Ausschnitt der Produktstruktur der SIEMENS Magnetresonanztomografen.

Abbildung 2.12 zeigt beispielhaft Ausschnitte aus den Konfigurationsdaten der MR-Geräte “Concerto” und “Harmony”. Das System “Harmony” enthält demnach drei konfigurierbare Untergruppen (MPCU, Receiver und Rx4). Als Receiver kann die Variante R-2 oder R-4 ausgewählt werden, wobei für R-4 eine zusätzliche Einschränkung angegeben ist, die besagt, dass R-4 nur dann verwendet werden darf, wenn für Gruppe Rx4 das Element X2 konfiguriert ist.

Einschränkungen, im XML-Schema *conditions* genannt, werden aus atomaren Ausdrücken der Form $g = e$ und $g \neq e$ aufgebaut. Dabei ist g die Bezeichnung eines Gruppenknoten und e die Bezeichnung eines Elementknotens, wobei, um Trivialfälle zu vermeiden, e ein Element der Gruppe g sein sollte. Aus diesen Elementareinschränkungen können dann komplexere Aussagen durch die Verwendung mehrerer *conditions*- bzw. *condition*-Einträge in der XML-Beschreibung. Erstere werden dabei implizit als disjunktiv verknüpft verstanden, letztere als konjunktiv, so dass sich insgesamt eine disjunktive Normalformdarstellung der Nebenbedingungen ergibt. Ist überhaupt kein *conditions*-Eintrag für einen Elementknoten vorhanden, so ist dieser Knoten gültig, d.h. es liegen keine einschränkenden Bedingungen vor.

Die Gruppenknoten besitzen darüberhinaus Attribute, durch die weitere Konfigurationseigenschaften festgelegt werden können. Zum einen gibt es ein optionales *Default*-Attribut, über das sich Standardwerte für Auswahlmöglichkeiten festlegen lassen. Daneben gibt es die beiden zwingend anzugebenden Attribute *minOccurs* und *maxOccurs*, die eine Zahleneinschränkung der gleichzeitig wählbaren Elementknoten dieser Gruppe festschreiben.

Die Prüfung einzelner Gerätekonfigurationen sowie die oben erwähnte Vollständigkeitsprüfung der Handbücher wird dann durchgeführt, indem aus den XML-Produktdaten eine aussagenlogische Formel generiert wird. Im ersten Fall der Konsistenzprüfing einzelner Konfigurationen stimmt dann die Erfüllbarkeit der so generierten Formel mit der Gültigkeit der Konfiguration überein. Im zweiten Fall der Vollständigkeitsprüfung der Textbausteine der Handbücher (Hilfepakete) ist zuerst eine zusätzliche Codierung der Themeninhalte und Verwendungspositionen dieser Bausteine erforderlich. Darauf aufbauend kann dann eine aussagenlogische Formel generiert werden, die auch die Konfigurationsformel beinhaltet und deren Allgemeingültigkeit mit der Vollständigkeit der Hilfepakete äquivalent ist (siehe auch [SK02]).

3

Formale Modelle

Zur exakten Beschreibung der Konfigurationsaufgabe und der Semantik eines gewählten Konfigurationsmodells ist eine *Formalisierung*, ein *formales Modell* erforderlich. In der Literatur wurden zu diesem Zweck verschiedene logische Sprachen vorgeschlagen, die wir im folgenden zunächst vergleichend gegenüberstellen wollen. Danach werden wir ausführlich auf dynamische Aussagenlogik (PDL) eingehen, eine Sprache, die sich gut zur Formalisierung von Produktionsregeln, oder allgemeiner von regelbasierten Expertensystemen, eignet. Wir werden insbesondere die PDL-Formeln in speziellen Strukturen, so genannten *regulären Kripke-Strukturen* interpretieren, die sich für unsere Anwendung als vorteilhaft erwiesen haben. Die Umsetzung regelbasierter Konfigurationssysteme zunächst in PDL und anschließend in pure Aussagenlogik wird danach thematisiert werden. Ziel dieser Vorgehensweise in zwei Schritten ist es, moderne aussagenlogische Beweiser (*SAT-Checker*) zur Verifikation von Konfigurationssystemen einsetzen zu können.

3.1 Gebräuchliche Konfigurationsformalismen

Wir stellen nun einige der gebräuchlicheren in der Literatur beschriebenen Sprachen zur Codierung von Konfigurationsproblemen vor.

3.1.1 Beschreibungslogiken

Beschreibungslogiken [BMNPS03] (*description logics*, DL) wurden als Formalismen zur Wissensrepräsentation und -verarbeitung entwickelt. In der Nachfolge semantischer Netzwerke [Qui67, BL85] und Frame-basierter Systeme [Min81] traten sie anfangs auch unter der Bezeichnung “terminologische Systeme” oder “Konzeptsprachen” auf. Heute werden sie hauptsächlich in der künstlichen Intelligenz (das Anwendungsgebiet Konfiguration eingeschlossen), zur Entwicklung von Datenbankschemata oder jüngst auch zur Beschreibung von Inhalten und Zusammenhängen im Internet ("Semantic Web") eingesetzt. Für letzteres etabliert sich die standardisierte Beschreibungssprache OWL [OWL02] als Nachfolger von DAML+OIL [DAM01] zusehends.

Beschreibungslogiken erlauben das Erfassen von Wissen über ein Anwendungsgebiet (die "Welt"), indem zuerst wesentliche Konzepte in einer so genannten *Terminologie* definiert werden, die dann verwendet wird, um die Gegenstände eines konkreten Ausschnitts dieses Gebiets und deren Zusammenhänge zu spezifizieren.

Die grundlegenden Begriffe der Beschreibungslogiken sind *Konzepte* (unäre Prädikate), *Rollen* (binäre Prädikate) und *Individuen* (Konstanten). Konzepte werden als Mengen oder Gruppierungen von einzelnen Objekten verstanden, Rollen geben die Beziehungen zwischen diesen an. Durch Individuen können bestimmte Objekte ausgezeichnet und so eindeutig benannt werden. Die DL-Formel

$$\mathsf{Vogel} \sqsubseteq \mathsf{Fliegt},$$

in der die beiden Konzepte Vogel und Fliegt in Zusammenhang gebracht werden, drückt beispielsweise aus, dass das Konzept Fliegt das Konzept Vogel einschließt, womit ausgedrückt wird, dass alle Vögel fliegen können. Mit dem DL-Ausdruck $\mathsf{Vogel}(\mathsf{dieseAmsel})$ lässt sich der Sachverhalt beschreiben, dass $\mathsf{dieseAmsel}$ ein Vogel ist. Ersteres könnte man durchaus auch in Prädikatenlogik als

$$(\forall x)(\mathsf{Vogel}(x) \Rightarrow \mathsf{Fliegt}(x))$$

formulieren, die Notation in der Beschreibungslogik ist allerdings um einiges kompakter und benötigt keine (Hilfs-)Variablen. Allgemein lassen sich viele der in der Literatur betrachteten DL-Dialekte in das 3-Variablen-Fragment $\ddot{\mathcal{L}}^3$ der Prädikatenlogik einbetten, unter Verzicht auf Relationskomposition für Rollen sogar in das entscheidbare 2-Variablen-Fragment $\ddot{\mathcal{L}}^2$ [Bor96].

Syntax

Komplexe Ausdrücke in Beschreibungslogiken werden aus atomaren Konzepten und Rollen durch verschiedene Konstruktoren aufgebaut. Für Konzepte verwenden wir die Bezeichner $C, D, \ldots$, Rollen werden wir mit $R, S, \ldots$ benennen, Individuen mit $a, b, c, \ldots$. Je nach betrachteter DL unterscheiden sich die zur Verfügung stehenden Konstruktoren. Wir haben hier eine unseres Erachtens für die Konfiguration passende Auswahl getroffen, die Felfernigs Auswahl [FFJ+03] ähnelt. Dieser verwendet jedoch eine an DAML+OIL angelehnte, weniger formal-logische Syntax. Unsere Notation haben wir dem einführenden Kapitel von Baader und Nutt [BN03] aus dem Handbuch der Beschreibungslogiken entnommen.

Konzeptkonstruktoren:

$\top$	universelles Konzept (“alle Gegenstände”)
$C \sqcup D$	Disjunktion, Konzeptvereinigung
$\neg C$	(Konzept-)Negation
$\geq n\, R.C$	qualifizierte Mächtigkeitseinschränkung ($n \in \mathbb{N}$)
$\{a_1, \ldots, a_n\}$	Individuenmenge

Rollenkonstruktoren:

$R \circ S$	Komposition

Informell bezeichnet die qualifizierte Mächtigkeitseinschränkung $\geq n\, R.C$ die Menge der Gegenstände, die zu mindestens n dem Konzept C angehörigen Gegenständen in der Beziehung R stehen. So bezeichnet z.B.

$$\geq 2\, \mathsf{IstVaterVon.Weiblich}$$

das Konzept aller Väter von mindestens zwei Töchtern und

$$\geq 2\, \mathsf{IstVaterVon} \circ \mathsf{IstElternteilVon}.\{\mathsf{Anne}, \mathsf{Klaus}\}$$

das Konzept der (gemeinsamen) Großväter von Anne und Klaus.

Wir wollen die so entstandene Beschreibungslogik mit DL$_0$ bezeichnen. Mathematisch exakt ausgedrückt sind DL$_0$-Formeln entweder Konzept- oder Rollenausdrücke. Diese definieren sich wie folgt:

Definition 3.1.1 (DL$_0$-Ausdrücke) *Sei Γ_0 die Menge der atomaren Konzepte, Π_0 die Menge der atomaren Rollen und Υ die Menge der Individuen, wobei wir fordern, dass Γ_0, Π_0 und Υ paarweise disjunkt sind. Dann sind die Menge der **DL$_0$-Konzepte** Γ und die Menge der **DL$_0$-Rollen** Π definiert als die kleinsten Mengen, für die gilt:*

1. $\Gamma_0 \subseteq \Gamma$, $\top \in \Gamma$, $\Pi_0 \subseteq \Pi$

2. *Wenn* $C, D \in \Gamma$*, dann auch* $(C \sqcup D) \in \Gamma$ *und* $(\neg C) \in \Gamma$*. Für alle* $n \in \mathbb{N}$ *ist mit* $C \in \Gamma$ *und* $R \in \Pi$ *auch* $({\geq} n\, R.C) \in \Gamma$*.*
3. *Falls* $a_1, \ldots, a_k \in \Upsilon$*, so ist* $\{a_1, \ldots, a_k\} \in \Gamma$*.*
4. *Wenn* $R, S \in \Pi$*, dann auch* $(R \circ S) \in \Pi$*.*

Die sich so ergebende Beschreibungslogik ist eine Erweiterung der bekannten Logik $\mathcal{ALC}$ (*attribute language with complement*) [SSS91] um Rollenkomposition, Nominalkonstruktoren (Individuenmengen), sowie qualifizierte Mächtigkeitseinschränkungen.

Um Klammern einzusparen, verwenden wir die üblichen von der Prädikatenlogik her bekannten Präzedenzregeln und notationellen Vereinfachungen.

Wir führen außerdem noch weitere abgeleitete Operatoren (für $C, D \in \Gamma, R \in \Pi, a \in \Upsilon, n \in \mathbb{N}$) ein:

$$
\begin{array}{rcll}
\bot & := & \neg\top & \text{leeres Konzept (“kein Gegenstand”)} \\
C \sqcap D & := & \neg(C \sqcup D) & \text{Konjunktion} \\
C \sqsubseteq D & := & \neg C \sqcup D & \text{Inklusion} \\
C \equiv D & := & (C \sqsubseteq D) \sqcap (D \sqsubseteq C) & \text{Äquivalenz} \\
\exists R.C & := & {\geq} 1\, R.C & \text{existentielle Quantifizierung} \\
\forall R.C & := & \neg(\exists R.\neg C) & \text{universelle Quantifizierung} \\
{\leq} n\, R.C & := & \neg({\geq}(n+1)\, R.C) & \text{(qualifizierte)} \\
{=} n\, R.C & := & {\leq} n\, R.C \sqcap {\geq} n\, R.C & \text{Mächtigkeitseinschränkungen} \\
R : a & := & \exists R.\{a\} & \text{Füll-Konstruktor (\textit{filler})}
\end{array}
$$

Für eine Relation $\diamond \in \{\leq, =, \geq\}$ verwenden wir darüberhinaus die *unqualifizierte Mächtigkeitseinschränkung* $\diamond n\, R$ als Abkürzung für $\diamond n\, R.\top$.

Semantik

Wir geben nun eine modell-theoretische Semantik für Beschreibungslogiken an, in der Konzepte als Mengen und Rollen als binäre Relationen aufgefasst werden. Zur Bestimmung der Wahrheitswerte wird eine Interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}})$ herangezogen, bestehend aus einer nicht-leeren Menge $\Delta^{\mathcal{I}}$, dem Grundbereich, und einer Interpretationsfunktion $(\cdot)^{\mathcal{I}}$. Die Interpretationsfunktion weist jedem atomaren Konzept A eine Menge $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, jeder atomaren Rolle R eine binäre Relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ und jeder Konstante (jedem Individuum) a ein Element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ zu. Bei letzterem gilt darüberhinaus üblicherweise die *unique name assumption (UNA)*, die besagt, dass für alle Konstanten a, b aus $a^\mathcal{I} = b^\mathcal{I}$ auch $a = b$ folgt; wir werden diese Annahme auch stillschweigend voraussetzen. Die Semantik komplexer Konzepte und Rollen wird dann induktiv definiert durch

$$\begin{aligned}
\top^\mathcal{I} &= \Delta^\mathcal{I} \\
(C \sqcup D)^\mathcal{I} &= C^\mathcal{I} \cup D^\mathcal{I} \\
(\neg C)^\mathcal{I} &= \Delta^\mathcal{I} \setminus C^\mathcal{I} \\
(\geq n\, R.C)^\mathcal{I} &= \{x \in \Delta^\mathcal{I} \mid \left|\{x \mid (x,y) \in R^\mathcal{I} \wedge y \in C^\mathcal{I}\}\right| \geq n\} \\
(\{a_1, \dots, a_n\})^\mathcal{I} &= \{a_1^\mathcal{I}, \dots, a_n^\mathcal{I}\} \\
(R \circ S)^\mathcal{I} &= R^\mathcal{I} \circ S^\mathcal{I} = \{(x,y) \mid (\exists z)((x,z) \in R^\mathcal{I} \wedge (z,y) \in S^\mathcal{I})\}
\end{aligned}$$

Eine Interpretation $\mathcal{I}$ *erfüllt* ein Konzept C, wenn $C^\mathcal{I} \neq \emptyset$. Gibt es ein $\mathcal{I}$, das C erfüllt, so heißt C *erfüllbar*. Falls für zwei Konzepte C und D für alle Interpretationen $\mathcal{I}$ gilt, dass $C^\mathcal{I} \subseteq D^\mathcal{I}$, so wird C von D *subsumiert*. Dafür wird auch die Notation $\models C \sqsubseteq D$ verwendet. Ist $C^\mathcal{I} = D^\mathcal{I}$ für alle Interpretationen $\mathcal{I}$, so heißen C und D *äquivalent*, was auch mit $\models C \equiv D$ bezeichnet wird.

Meist wird ein Konzept C nicht für sich alleine bewertet, sondern in Bezug auf eine bestehende Wissensdatenbank. Diese kann *terminologisches Wissen* (terminological knowledge, “T-Box”) in der Form von Konzeptdefinitionen sowie *Aussagewissen* (assertional knowledge, “A-Box”) in der Form von Konzept- und Rollenzugehörigkeiten von Individuen enthalten. Die Wissensdatenbanken entsprechen in der üblichen formal-logischen Terminologie den Theorien.

Konzeptdefinitionen oder *terminologische Axiome* haben die Form $C \sqsubseteq D$ oder $C \equiv D$ und können entweder in einer *deskriptiven* oder einer *rekursiven* Semantik interpretiert werden [Neb91]. Grob gesprochen entsprechen Axiome in deskriptiver Semantik Abkürzungen für komplexe Ausdrücke (Makros), während Axiome in rekursiver Semantik als möglicherweise rekursive, definierende Gleichungen aufgefasst werden können. Der Unterschied tritt nur bei zyklischen Terminologien auf, wie z.B. bei der Definition

$$\mathsf{BinaryTree} \;\equiv\; \mathsf{Tree} \sqcap {\leq} 2\, \mathsf{has\text{-}branch} \sqcap \forall \mathsf{has\text{-}branch}.\mathsf{BinaryTree} \quad ,$$

die einen Binärbaum als einen Baum mit höchstens zwei Kindknoten, die jeweils wiederum Binärbäume sind, charakterisiert. Wir werden uns im folgenden auf die einfachere deskriptive Semantik beschränken, die sich für unsere Anwendung jedoch als ausreichend erwiesen hat. In dieser Semantik lassen sich zyklische Definitionen (wie z.B. $C \equiv D \sqcup C$) immer durch äquivalente azyklische ersetzen [Sch91] (im Beispiel: $D \sqsubseteq C$), so dass wir uns sogar auf azyklische Axiome beschränken können.

Aussagewissen ist Faktenwissen über die Konzept- und Rollenzugehörigkeit von Individuen. Für Individuen a, b, ein Konzept C und eine Rolle R verwenden wir die Notationen $C(a)$ und $R(a, b)$ um auszudrücken, dass a zum Konzept C und (a, b) zur Rolle R gehört. In der formalen Semantik ist $C(a)$ für eine Interpretation $\mathcal{I}$ erfüllt, falls $a^{\mathcal{I}} \in C^{\mathcal{I}}$. Analog gilt $R(a, b)$, falls $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.

Aussagewissen (das immer atomar ist) kann mittels der Individuenmengen und des Füllkonstruktors leicht zu Konzeptdefinitionen transformiert werden. Dabei wird eine Konzeptzugehörigkeitsaussage der Form $C(a)$ durch das terminologische Axiom $\{a\} \sqsubseteq C$ und eine Rollenzugehörigkeitsaussage $R(a, b)$ durch $\{a\} \sqsubseteq R : b$, oder äquivalent $\{a\} \sqsubseteq \exists R.\{b\}$, ersetzt. Wir beschränken uns daher im folgenden auf Wissensdatenbanken, die ausschließlich Konzeptdefinitionen enthalten. Es sei noch angemerkt, dass die so definierte Semantik für Aussagewissen eine Offene-Welt-Semantik (“open world semantics”) ist, so dass aus dem Fehlen von Fakten nur auf deren Unbestimmtheit, nicht aber auf deren Ungültigkeit geschlossen werden darf.

Wir wollen nun die Begriffe Erfüllbarkeit, Subsumtion und Äquivalenz auf Aussagen bezüglich Wissensdatenbanken erweitern. Dazu betrachten wir eine Wissensdatenbank (oder Terminologie), die durch eine (endliche) Menge $\mathcal{T}$ von Konzeptdefinitionen K der Form $C \sqsubseteq D$ oder $C \equiv D$ gegeben sei. Eine Interpretation $\mathcal{I}$ erfüllt dann eine Inklusion $C \sqsubseteq D$, falls $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, und eine Äquivalenz $C \equiv D$, falls $C^{\mathcal{I}} = D^{\mathcal{I}}$. Eine Theorie $\mathcal{T}$ wird von $\mathcal{I}$ erfüllt, wenn alle Konzeptdefinitionen $K \in \mathcal{T}$ von $\mathcal{I}$ erfüllt werden.

Wir bezeichnen nun ein Konzept C als *erfüllbar bezüglich* $\mathcal{T}$, falls es eine Interpretation $\mathcal{I}$ gibt, die sowohl C als auch $\mathcal{T}$ erfüllt. C *subsumiert* D *bezüglich* $\mathcal{T}$, geschrieben als $\mathcal{T} \models C \sqsubseteq D$, falls jede Interpretation, die $\mathcal{T}$ erfüllt, auch $C \sqsubseteq D$ erfüllt. C und D heißen *äquivalent bezüglich* $\mathcal{T}$, falls jede Interpretation mit $\mathcal{T}$ auch $C \equiv D$ erfüllt. Dafür schreiben wir auch $\mathcal{T} \models C \equiv D$.

Terminologiebezogene Erfüllbarkeits-, Äquivalenz- und Subsumtionsaussagen können auf solche ohne Bezug zu einer Terminologie reduziert werden. Dazu ist eine genügend ausdrucksstarke Beschreibungslogik erforderlich, beispielsweise wie oben definiert, die Negation und Vereinigung von Konzepten vorsieht. Dann ist eine *Internalisierung* von Terminologien möglich, die wir am Beispiel der Subsumtionsaussage $\mathcal{T} \models C \sqsubseteq D$ erläutern wollen.

Dazu nehmen wir an, dass $\mathcal{T} = \{I_1, \ldots, I_m, E_1, \ldots, E_n\}$, wobei I_j Inklusionsaussagen der Form $I_j = (C_j \sqsubseteq D_j)$ und E_k Äquivalenzaussagen der Form $E_k = (F_k \equiv G_k)$ seien. Dann ist die terminologieunabhängige Subsumtionsaussage

$$\models \neg(C_1 \sqsubseteq D_1) \sqcup \cdots \sqcup \neg(C_m \sqsubseteq D_m) \sqcup \\ \neg(F_1 \equiv G_1) \sqcup \cdots \sqcup \neg(F_n \equiv G_n) \sqcup C \sqsubseteq D$$

äquivalent zu obiger terminologie-relativer Aussage $\mathcal{T} \models C \sqsubseteq D$. Die Internalisierung von Terminologien entspricht der Anwendung des Deduktionstheorems (siehe z.B. [Gal86]) in klassischen Logiken.

Syntaktische und semantische Alternativen

Zur hier vorgenommenen Auswahl an Konzept- und Rollenkonstruktoren sind verschiedene Alternativen denkbar. So können beispielsweise anstelle der qualifizierten Mächtigkeitseinschränkungen $(\leq n\, R.C)$ unqualifizierte $(\leq n\, R)$ verwendet werden. Um keine Einschränkung der Ausdrucksfähigkeit der Sprache in Kauf nehmen zu müssen, ist es dann aber erforderlich, mindestens einen der Quantifizierungsoperatoren, also z.B. $\exists R.C$, zur Menge der grundlegenden, nicht abgeleiteten Konzeptkonstruktoren hinzuzunehmen und die Rollenkonstruktoren um einen Vereinigungsoperator $(R \sqcup S)$ zu erweitern. Dann lässt sich das Auftreten einer qualifizierten Mächtigkeitseinschränkung $\leq n\, R.C$ in einer Formel F ersetzen durch die unqualifizierte Einschränkung $\leq n\, R'$, sofern F erweitert wird zu

$$F' \ \sqcap \ \forall R'.C \ \sqcap \ \forall R''.\neg C \ ,$$

und F' aus F dadurch entsteht, dass jedes Vorkommen von R durch $R' \sqcup R''$ ersetzt wird. R' und R'' sind dabei zwei neue, in F nicht vorkommende Rollensymbole. Die Idee dieser Vorgehensweise ist, die mit dem Konzept C qualifizierte Rolle R in zwei Rollen R' und R'' aufzuspalten, von denen erstere nur zum Konzept C gehörige Elemente in ihren Wertebereich aufnimmt, letztere nur nicht zum Konzept C gehörige.

Eine weitere syntaktische Alternative ist die (häufig anzutreffende) Erweiterung um zusätzliche Rollenkonstruktoren. Dazu gehören neben den Booleschen Operatoren der Rollenvereinigung $(R \sqcup S)$, der Rollennegation $(\neg R)$ und des Rollenschnitts $(R \sqcap S)$ auch inverse (oder konverse) Rollen ($R^{\smile}$ in Tarskis Notation [TG87]), transitiver Abschluss von Rollen, sowie Rollengleichheits- und -inklusionsoperatoren $(R = S,\ R \subseteq S)$. Besonders letztere nehmen im Zusammenwirken mit der Rollenkomposition eine herausragende Stellung ein und sind in der Literatur dann unter der Bezeichnung Rollen-Wert-Abbildung (*role-value-map*, [BS85]) oder, im Spezialfall funktionaler Rollen, als Feature- oder Attributübereinstimmungen (*attribute/feature agreement/disagreement*, [Smo88]) anzutreffen. Dabei werden dann hauptsächlich Gleichungen der Form

$$R_1 \circ \cdots \circ R_i = S_1 \circ \cdots \circ S_j$$

verwendet, die als simple Mengengleichheiten der sich ergebenden Produktrelationen interpretiert werden. So kann man beispielsweise mit der bedingten Gleichung

$$\exists \mathsf{hatFarbe}.\top \ \sqsubseteq \ (\mathsf{hatTeil} \circ \mathsf{hatFarbe} = \mathsf{hatFarbe})$$

ausdrücken, dass die Bestandteile (festgelegt durch die Teil-Ganzes-Relation $\mathsf{hatTeil}$) aller Gegenstände, die eine Farbe haben ($\exists\mathsf{hatFarbe}.\top$), auch farbig sind und darüberhinaus dieselbe Farbe (oder dieselben Farben) besitzen wie diese selbst.

Beschränkt man sich auf funktionale Rollen, d.h. auf solche, für die mit $(a, b) \in R$ und $(a, c) \in R$ immer auch $b = c$ gilt (äquivalent beschrieben durch $\leq 1\, R$) , so ist man bei den funktionalen Rollenketten (*functional chains*) angelangt. Für funktionale Ketten $f = R_1 \circ \cdots \circ R_i$ und $g = S_1 \circ \cdots \circ S_j$ bezeichnet dann der Ausdruck $f \doteq g$ das Konzept

$$\{x \in \Delta^{\mathcal{I}} \mid (\exists y)((x, y) \in f^{\mathcal{I}} \wedge (x, y) \in g^{\mathcal{I}})\}$$

und der Ausdruck $f \dot{\neq} g$ das Konzept

$$\{x \in \Delta^{\mathcal{I}} \mid (\exists y_1, y_2)((x, y_1) \in f^{\mathcal{I}} \wedge (x, y_2) \in g^{\mathcal{I}} \wedge y_1 \neq y_2)\} \ .$$

Mit funktionalen Ketten lassen sich Attributübereinstimmungen und -verschiedenheiten (wobei Attribute jeweils definiert sein müssen) explizieren. Solche Attributübereinstimmungen spielen, insbesondere zusammen mit den weiter unten zu beschreibenden konkreten Grundbereichen, nicht nur in der Produktkonfiguration eine entscheidende Rolle, wo sie Felfernig beispielsweise zur Modellierung von Produkteigenschaften wie Preis, Farbe oder Kapazitäten einsetzt [FFJ+03], sondern auch generell in Wissensrepräsentationssprachen und -systemen wie beispielsweise KL-ONE [BS85]. Wir werden später im Abschnitt über Feature-Logik noch genauer auf die Konstrukte zur Feststellung von Attributübereinstimmungen und deren Anwendungen eingehen.

Die letzte, gleichzeitig syntaktische und semantische Variante, die wir vorstellen wollen, betrifft die Konkretisierung von Teilen des Grundbereichs, der Wertebereiche von Attributen. Dabei wird eine bekannte Struktur, beispielsweise die natürlichen oder die rationalen Zahlen, als Teilstruktur in den Grundbereich eingebettet und mit einer speziellen Semantik versehen. Um die konkreten Grundbereiche (*concrete domains*) nutzbar zu machen, muss die Beschreibungslogik auch einer syntaktischen Erweiterung unterzogen werden. Wir schließen uns bei deren Wahl Baader und Hanschke an [BH91, Han92], verwenden aber eine vereinfachte Semantik.

Definition 3.1.2 *Ein* ***konkreter Grundbereich*** $\mathcal{D}$ *besteht aus einer Menge* $\Delta^{\mathcal{D}}$*, dem Grundbereich von* $\mathcal{D}$*, und einer Menge PRED*$(\mathcal{D})$ *von Prädikaten beliebiger Stelligkeit über* $\mathcal{D}$*.*

Die Syntax unserer Logik DL$_0$ wird dann erweitert um zusätzliche Konzeptkonstruktoren der Form

$$\exists(f_1, \ldots, f_n).P \quad \text{und} \quad \forall(f_1, \ldots, f_n).P \ ,$$

wobei die f_i (nicht zwingend funktionale) Rollenketten und $P \in \text{PRED}(\mathcal{D})$ ein konkretes Prädikat der Stelligkeit n ist. Die so konstruierten Konzepte heißen *existenzielle* bzw. *universelle Prädikatrestriktionen*. Erstere sind (informell) so zu interpretieren, dass man durch die Rollenketten f_i zu Werten im konkreten Grundbereich gelangt, die der Relation P genügen. Die Definition der Menge der DL$_0$-Ausdrücke muss nun natürlich noch entsprechend angepasst werden, wobei wir auch Werte aus dem konkreten Grundbereich als Individuen zulassen wollen, so dass wir auch den Fall $\Upsilon \cap \Delta^{\mathcal{D}} \neq \emptyset$ zulassen.

Zur Bestimmung von Wahrheitswerten verändern wir die Semantik der DL$_0$-Ausdrücke wie folgt: Als Interpretationen $\mathcal{I}$ sind nur noch solche zulässig, deren Grundbereich $\Delta^{\mathcal{I}}$ der Einschränkung $\Delta^{\mathcal{D}} \subseteq \Delta^{\mathcal{I}}$ genügt. Die Interpretationsfunktion $(\cdot)^{\mathcal{I}}$ wird auf die neuen Konzeptkonstruktoren ausgedehnt durch

$$\begin{aligned}(\exists(f_1, \ldots, f_n).P)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid (\exists r_1, \ldots, r_n) \\ &\qquad (r_i \in \Delta^{\mathcal{D}} \text{ und } (x, r_i) \in f_i^{\mathcal{I}} \text{ für alle } i \text{ und } (r_1, \ldots, r_n) \in P)\}\end{aligned}$$

und

$$\begin{aligned}(\forall(f_1, \ldots, f_n).P)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid (\forall r_1, \ldots, r_n) \\ &\quad (\text{falls } r_i \in \Delta^{\mathcal{D}} \text{ und } (x, r_i) \in f_i^{\mathcal{I}} \text{ für alle } i\text{, so ist } (r_1, \ldots, r_n) \in P)\}.\end{aligned}$$

Außerdem werden die konkreten Individuen $r \in \Delta^{\mathcal{D}}$ durch sich selbst interpretiert, so dass $(r)^{\mathcal{I}} = r$ für $r \in \Delta^{\mathcal{D}}$ gilt.

An einem Beispiel wollen wir die konkreten Grundbereiche erläutern. Dazu betrachten wir als konkreten Grundbereich die natürlichen Zahlen $\mathbb{N}$ mit der Ordnungsrelation $\geq$. Als einstellige Spezialisierung der Relation betrachten wir darüberhinaus auch noch das Prädikat $\geq_{10}$, das wir durch $x \in \geq_{10} \;\Leftrightarrow\; (x, 10) \in \geq$ definieren. Für unseren konkreten Grundbereich $\mathcal{D}$ gilt also $\Delta^{\mathcal{D}} = \mathbb{N}$ und $\text{PRED}(\mathcal{D}) = (\geq, \geq_{10})$. Dann bezeichnet der DL-Ausdruck

$$\mathsf{Computer} \sqcap \forall(\mathsf{hatFestplatte} \circ \mathsf{hatKapazität}).\ \geq_{10}$$

diejenigen Computer, bei denen die Kapazität aller Festplatten größer als 10 (GB) ist. Nehmen wir für ein weiteres erläuterndes Beispiel die Rollenzugehörigkeitsaussagen

$$\begin{array}{ll} \mathsf{benötigtRAM}(\mathsf{Programm\text{-}1}, 150), & \mathsf{hatRAM}(\mathsf{Computer\text{-}1}, 128), \\ \mathsf{benötigtRAM}(\mathsf{Programm\text{-}2}, 40), & \mathsf{hatRAM}(\mathsf{Computer\text{-}2}, 256) \end{array}$$

als gegeben an, die ausdrücken sollen, dass Programm 1 (2) mindestens 150 (40) MB Hauptspeicher zur Ausführung benötigt und dass Computer 1 (2) einen Hauptspeicher der Kapazität 128 (256) MB bereitstellt. [1] Die DL-Aussage

$$\mathsf{Computer} \sqcap \forall(\mathsf{hatRAM}, \mathsf{hatProgramm} \circ \mathsf{benötigtRAM}).\ \geq \tag{3.1}$$

beschreibt dann diejenigen Computer, die für alle ihnen (mittels der Rolle hatProgramm) zugeordneten Programme genügend Hauptspeicherkapazität zur Verfügung stellen. Mit den weiteren Rollenzugehörigkeitsaussagen

$$\begin{array}{l} \mathsf{hatProgramm}(\mathsf{Computer\text{-}1}, \mathsf{Programm\text{-}1}), \\ \mathsf{hatProgramm}(\mathsf{Computer\text{-}2}, \mathsf{Programm\text{-}2}) \end{array}$$

ist Aussage 3.1 in unserem Beispiel für Computer 2 erfüllt, für Computer 1 jedoch nicht.

Abschließend wollen wir noch vermerken, dass auch Funktionen wie z.B. die Addition durch konkrete Grundbereiche in die Beschreibungslogik einbezogen werden können. Dazu wird einer n-stelligen Funktion ein $n+1$-stelliges Prädikat zugeordnet, das dann bereits die Gleichheitsrelation mit enthält. Beispielsweise ergibt sich so für die Addition ein Prädikat $P(x, y, z)$, das durch $(x, y, z) \in P \;\Leftrightarrow\; x + y = z$ definiert ist.

---

[1] Das so gegebene Aussagewissen lässt sich mittels der oben angegebenen Transformationsmethode auch in Konzeptaussagen umformen.

Leider sind Berechnungen, die sich auf eine variable Anzahl von Gegenständen beziehen, wie sie z.B. zur Summenbildung über die Kapazitäten aller Programme benötigt würde, mit der Erweiterung um konkrete Grundbereiche nicht zu realisieren. Dies ist gerade in der Anwendung von Beschreibungslogiken zur Produktkonfiguration eine nicht zu unterschätzende Einschränkung. Zur Attributmodellierung wie z.B. des Preises mittels natürlicher Zahlen sind konkrete Grundbereiche allerdings ausreichend.

Entscheidungsverfahren und Komplexität

Die beiden wichtigsten algorithmischen Fragestellungen der Beschreibungslogiken sind die Erfüllbarkeit von Konzepten (Gibt es eine Interpretation, in der das Konzept nicht leer ist?) und die Subsumtion von Konzepten (Ist ein Konzept in einem anderen enthalten?). In Logiken mit Konzeptnegation und -disjunktion, wie sie typischerweise betrachtet werden, lässt sich letztere Fragestellung auf erstere zurückführen, so dass wir uns im folgenden auf die Entscheidbarkeit und Komplexität des Erfüllbarkeitsproblems beschränken wollen. Darüberhinaus setzen wir eine deskriptive Semantik der Terminologieaxiome voraus, da diese zur Behandlung von Konfigurationsaufgaben ausreichend ist. Algorithmen zur Behandlung von Terminologien in rekursiver Semantik können bei Baader [Baa96] und Nebel [Neb90] gefunden werden.

Die oben definierte Beschreibungslogik DL_0 ist entscheidbar. Der grundlegende, Tableaux-basierte Algorithmus stammt von Schmidt-Schauß und Smolka, die damit auch die PSPACE-Vollständigkeit des Erfüllbarkeitsproblems gezeigt haben [SSS91]. Tabelle 3.1 zeigt die Komplexität der verschiedenen oben vorgestellten Erweiterungen von DL_0.

Logik	Komplexität	Bemerkungen
DL_0	PSPACE-vollständig	ähnelt der Logik $\mathcal{ALCN}$ aus [SSS91]
$\mathrm{DL}_0^* + \mathrm{BR}$	PSPACE bis unentscheidbar	je nach erlaubten Operatoren, siehe [BS99][2]
$\mathrm{DL}_0^* + \mathrm{RVM}$	unentscheidbar	[SS89]
$\mathrm{DL}_0^* + \mathrm{FA}$	PSPACE-vollständig	[HN90]
$\mathrm{DL}_0^* + \mathrm{CD}$	NEXPTIME-hart[3]	[Lut01]

Tabelle 3.1: Entscheidbarkeit und Komplexität einiger Beschreibungslogiken.

Mit DL$_0^*$ bezeichnen wir die Logik DL$_0$ ohne Nominalkonstruktoren, für die Erweiterungen von DL$_0$ haben wir die folgenden Abkürzungen verwendet:

BR:	Boolesche Rollenkonstruktoren
RVM:	Rollen-Wert-Abbildungen (Gleichungen, role-value-maps)
FA:	Attribut-Übereinstimmungen (feature agreement)
CD:	konkrete Grundbereiche (concrete domains)

Die Grenze zwischen entscheidbaren und unentscheidbaren Beschreibungslogiken hängt zum Teil von relativ unscheinbaren Änderungen ab. So sind Rollen-Gleichungen in ihrer allgemeinen Form unentscheidbar, die Einschränkung auf funktionale Rollen macht sie aber entscheidbar. Ähnlich subtil ist die Abhängigkeit von der Auswahl der Booleschen Rollenkonstruktoren. Baader und Sattler geben hier eine vollständige Aufschlüsselung des Grenzverlaufs [BS99].

Die in der Literatur anzutreffenden Entscheidungsverfahren lassen sich grob in zwei Klassen einteilen: Tableaux-basierte und solche, die auf einer Übersetzung in entscheidbare Fragmente der Prädikatenlogik erster Stufe beruhen. Borgidas Transformation [Bor96] übersetzt beispielsweise in das entscheidbare 2-Variablen-Fragment $\mathcal{L}^2$; hierbei ist allerdings auf die Rollenkomposition zu verzichten, da diese drei Variablen benötigt. Ebenso lassen sich Mächtigkeitseinschränkungen im allgemeinen nicht mit zwei Variablen ausdrücken. Eine gangbare Alternative ist hier die Übersetzung in 2-Variablen-Logik mit zählenden Quantoren $(\exists^n)$, $\mathcal{C}^2$. Grädel *et al.* [GOR97] haben gezeigt, dass diese Logik entscheidbar ist, von Pacholski *et al.* [PST00] stammt ein Entscheidungsalgorithmus für das Erfüllbarkeitsproblem, der in der Komplexitätsklasse 2-NEXPTIME liegt. Wie man obiger Tabelle entnehmen kann, liefern direkte Entscheidungsverfahren für DL aber oft bessere obere Schranken für Laufzeit und Platzbedarf als die Transformation in Fragmente der Prädikatenlogik.

Konfiguration in DL

Wir wollen nun beispielhaft die Umsetzung von Ausschnitten einer Fahrzeugkonfiguration in die Beschreibungslogik DL$_0$ betrachten. Als Ausgangspunkt unserer Umsetzung wählen wir eine Beschreibung der Fahrzeugstruktur in UML [RJB98], wie in Abbildung 3.1 angegeben.

---

[2] Erweiterung um $\{\sqcup, \sqcap\}$ ist z.B. unentscheidbar.

[3] Für polynomiell entscheidbaren Grundbereich $\mathcal{D}$.

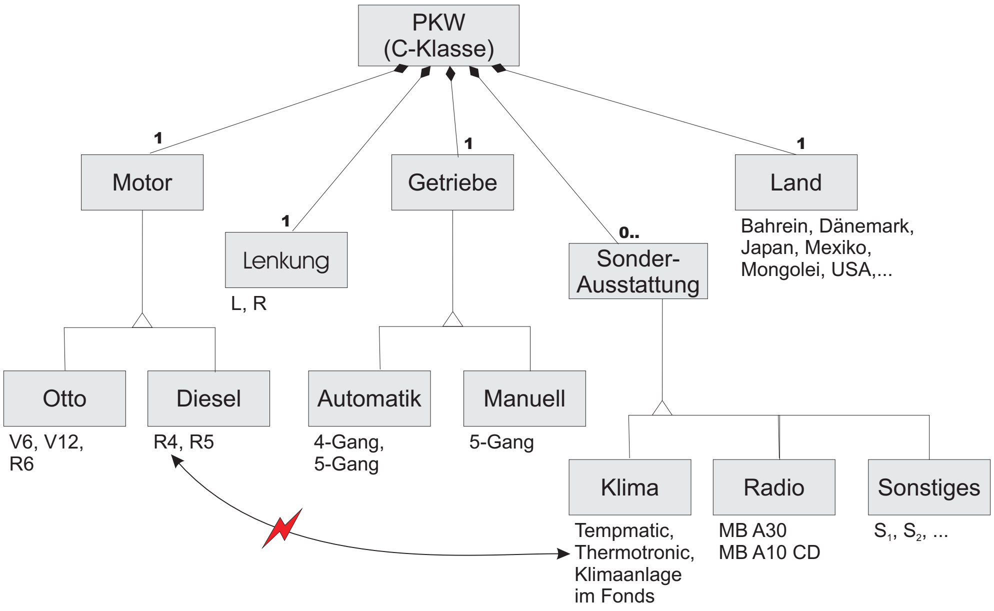


Abbildung 3.1: Vereinfachte, beispielhafte Fahrzeugkonfiguration.

Wie dieser Abbildung zu entnehmen ist, enthält ein PKW der C-Klasse genau einen Motor, ein Getriebe und verschiedene Sonderausstattungen. Außerdem muss eine Lenkungsausführung (Links- oder Rechtslenker) sowie ein Auslieferungsland angegeben werden. Für den Motor gibt es verschiedene Varianten, grob untergliedert in Otto- und Dieselmotoren, ebenso gilt es beim Getriebe zwischen automatischer und manueller Ausführung zu unterscheiden. Die Sonderausstattungen sind in verschiedene Kategorien eingeteilt, die beispielsweise die Klimatisierung oder das Radio betreffen. Für jede dieser Kategorien gibt es verschiedene konkrete Ausstattungspakete, so z.B. MB A30 und MB A10 CD für die Kategorie Radio. Zwischen diesen Ausstattungspaketen können Abhängigkeiten bestehen, von denen wir in unserem vereinfachten Schema nur eine mit aufgenommen haben, nämlich die hypothetische Annahme, dass sich der R4-Dieselmotor mit der Klimaanlage “Thermotronic” nicht vertrage.

Zusätzliche Details, wie Ressourcenbedarf einzelner Komponenten oder deren weitere Zerlegung in konstituierende Subkomponenten, werden wir in unserem Beispiel nicht betrachten; deren Umsetzung innerhalb unserer Beschreibungslogik stehen aber keine prinzipiellen Schwierigkeiten im Weg. So ist beispielsweise Felfernigs Vorgehensweise [FFJ+03] leicht auf unseren Formalismus zu übertragen.

Das Ergebnis der (manuellen) Übersetzung der Fahrzeugkonfiguration aus dem UML-Diagramm in die Beschreibungslogik DL_0 ist in Abbildung 3.2 zu sehen. Wir wollen einige Umsetzungsdetails nun näher erläutern.

Bei der Konfigurationsbeschreibung in DL_0 werden die Komponenten (Teile) des Fahrzeugs als Individuen der Beschreibungslogik dargestellt, Konzepte werden dadurch zu Mengen von Komponenten. So setzt sich in unserem beispielhaften Konfigurationsmodell die Kategorie der Ottomotoren, OttoM, aus den Individuen V6_OttoM, V12_OttoM und R6_OttoM zusammen. Sollen die individuellen Motoren nicht als Basiskomponenten angesehen werden, sondern als zusammengesetzte Teilzusammenbauten, so müssen diese auch als Kategorien dargestellt werden. Deren Einzelteile würden dann zu den Individuen der veränderten Formalisierung. Sollte also z.B. der Motor V6_OttoM nicht als atomares Einzelteil angesehen werden, so wäre dieser ähnlich der zusammengesetzten Kategorie PKW zu definieren.

Aggregationen (wie z.B. ein PKW enthält genau einen Motor) werden in DL_0 anhand der Rollenbeziehung “hatTeil” erfasst, durch welche Teilkomponenten mit dem sie (direkt) enthaltenden Teilzusammenbau oder Ganzen in Verbindung gebracht werden. Mittels der qualifizierten Mächtigkeitseinschränkungen kann die Multiplizität der Zuordnung angegeben werden. Somit lassen sich sowohl 1:1- als auch 1:n-Beziehungen modellieren. Um auszuschließen, dass nicht zum Gesamtkonzept gehörende Teile diesem zugeordnet werden, muss die Vollständigkeit der Aggregation, beipsielsweise mit Hilfe einer universellen Quantifizierung, wie in der fünften Zeile des Beispiels, spezifiziert werden.

Spezialisierung von Komponenten und deren Anordnung in einer Komponentenhierarchie kann über terminologische Axiome der Form

$$\mathsf{A} \equiv \mathsf{S}_1 \sqcup \cdots \sqcup \mathsf{S}_n$$

erreicht werden, wobei es sich bei A um das allgemeinere, bei den S_i um die spezielleren Konzepte handelt. Sind die Konzepte S_i paarweise disjunkt, was üblicherweise der Fall ist, so muss dies durch weitere Axiome, z.B. der Form

$$\mathsf{S}_i \sqcap \mathsf{S}_j \sqsubseteq \bot$$

für alle $1 \leq i < j \leq n$ angegeben werden. Ist eine festgelegte Menge von Komponenten gleicher Funktionalität vorhanden, so können diese über den Konstruktor für Individuenmengen zu einem Konzept zusammengefasst werden. Eine Spezifikation der Verschiedenheit der Individuen ist wegen der *unique name assumption* nicht erforderlich.

$$
\begin{aligned}
\mathsf{PKW} &\sqsubseteq {=}1\ \mathsf{hatTeil.Motor}\\
\mathsf{PKW} &\sqsubseteq {=}1\ \mathsf{hatTeil.Getriebe}\\
\mathsf{PKW} &\sqsubseteq {=}1\ \mathsf{hatTeil.Lenkung}\\
\mathsf{PKW} &\sqsubseteq {=}1\ \mathsf{hatTeil.Land}\\
\mathsf{PKW} &\sqsubseteq \forall \mathsf{hatTeil.}(\mathsf{Motor} \sqcup \mathsf{Getriebe} \sqcup \mathsf{Lenkung}\\
&\qquad\qquad\quad \sqcup \mathsf{Land} \sqcup \mathsf{Sonderausstattung})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Motor} &\equiv \mathsf{OttoM} \sqcup \mathsf{DieselM}\\
\mathsf{Getriebe} &\equiv \mathsf{AutomatikG} \sqcup \mathsf{ManuellesG}\\
\mathsf{Lenkung} &\equiv \{\mathsf{links}, \mathsf{rechts}\}\\
\mathsf{Land} &\equiv \{\mathsf{D\ddot{a}nemark}, \mathsf{Japan}, \mathsf{Mexiko}, \dots\}\\
\mathsf{Motor} &\equiv \mathsf{OttoM} \sqcup \mathsf{DieselM}\\
\mathsf{Sonderausstattung} &\equiv \mathsf{KlimaA} \sqcup \mathsf{RadioA} \sqcup \mathsf{SonstigeA}\\
\mathsf{OttoM} &\equiv \{\mathsf{V6\_OttoM}, \mathsf{V12\_OttoM}, \mathsf{R6\_OttoM}\}\\
\mathsf{DieselM} &\equiv \{\mathsf{R4\_DieselM}, \mathsf{R5\_DieselM}\}\\
\mathsf{AutomatikG} &\equiv \{\mathsf{4\text{-}GangAG}, \mathsf{5\text{-}GangAG}\}\\
\mathsf{ManuellesG} &\equiv \{\mathsf{5\text{-}GangMG}\}\\
\mathsf{KlimaA} &\equiv \{\mathsf{Tempmatic}, \mathsf{Thermotronic}, \mathsf{Fonds\text{-}KlimaA}\}\\
\mathsf{RadioA} &\equiv \{\mathsf{MB\_A30}, \mathsf{MB\_A10\_CD}\}\\
\mathsf{SonstigeA} &\equiv \{\mathsf{S}_1, \dots, \mathsf{S}_k\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{OttoM} \sqcap \mathsf{DieselM} &\sqsubseteq \bot\\
\mathsf{AutomatikG} \sqcap \mathsf{ManuellesG} &\sqsubseteq \bot\\
\mathsf{KlimaA} \sqcap \mathsf{RadioA} &\sqsubseteq \bot\\
\mathsf{KlimaA} \sqcap \mathsf{SonstigeA} &\sqsubseteq \bot\\
\mathsf{RadioA} \sqcap \mathsf{SonstigeA} &\sqsubseteq \bot
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{hatTeil} : \mathsf{Thermotronic} &\sqsubseteq \neg \mathsf{hatTeil} : \mathsf{R4\_DieselM}\\
\mathsf{hatTeil} : \mathsf{S}_2 &\sqsubseteq \neg \mathsf{hatTeil} : \mathsf{Mexiko} \sqcap \exists \mathsf{hatTeil.ManuellesG}
\end{aligned}
$$

Abbildung 3.2: Konfigurationsbeispiel in der Beschreibungslogik DL_0.

Nebenbedingungen, wie beispielsweise der in Abbildung 3.1 gezeigte Ausschluss, können, wenn sie auf der Ebene der Individuen greifen sollen, entweder als zusätzliche, nicht in DL, also extern formulierte Einschränkung an die Konzept- bzw. Rollenzugehörigkeiten $C(a)$ oder $R(a, b)$ angegeben werden oder direkt in DL mittels terminologischer Axiome unter Verwendung des Füllkonstruktors, wie weiter oben beschrieben. In Abbildung 3.2 haben wir diese Beschreibungsform gewählt; eine äquivalente externe Formulierung unter Verwendung von Aussageabhängigkeiten, die in Prädikatenlogik formuliert werden, könnte, wieder am Beispiel des Ausschlusses aus Abbildung 3.1 gezeigt, lauten:

$$\forall x(\mathsf{hatTeil}(x, \mathsf{Thermotronic}) \land \mathsf{hatTeil}(x, \mathsf{R4\_DieselM}) \Rightarrow \bot) \enspace .$$

Die in Abbildung 3.2 angegebene zweite Einschränkung auf Individuenebene drückt aus, dass Sonderausstattung S_2 nur zusammen mit dem manuellen Getriebe und nur, sofern das Fahrzeug nicht für Mexiko bestimmt ist, verbaut werden darf.

Zur Spezifikation eines Ausschlusses auf Konzeptebene bedient man sich ähnlicher Konstrukte wie der oben für die Disjunktheit der spezialisierten Klassen der Abstraktionshierarchie beschriebenen. So lässt sich z.B. die Einschränkung, dass USA-Fahrzeuge immer mit einer Klima-Sonderausstattung versehen sein müssen, durch die Konzeptinklusion

$$\mathsf{hatTeil}\!:\!\mathsf{USA} \sqsubseteq \exists \mathsf{hatTeil}.\mathsf{KlimaA}$$

angeben.

Sind die terminologischen Axiome einer Produktfamilie, und damit die Beschreibung der Produktstruktur, festgelegt, so kann die Auswahl eines bestimmten Produkts bezüglich dieser Terminologie auf zwei Arten geschehen: entweder über eine dieses spezielle Produkt beschreibende Formel oder durch Aussagewissen, d.h. durch eine Menge von Fakten über dieses Produkt. Ersteres sieht dann beispielsweise wie folgt aus, wenn wir die Beschreibung eines Fahrzeugs P_1 (als Konzept) bezüglich obiger Fahrzeugterminologie vornehmen wollen:

$$\begin{aligned} P_1 \equiv\ & \mathsf{PKW} \sqcap \mathsf{hatTeil} : \mathsf{V12\_OttoM} \sqcap \mathsf{hatTeil} : \mathsf{Dänemark} \sqcap \\ & \mathsf{hatTeil} : \mathsf{L} \sqcap \mathsf{hatTeil} : \mathsf{5\text{-}GangAG} \sqcap \mathsf{hatTeil} : \mathsf{Tempmatic} \sqcap \\ & \mathsf{hatTeil} : \mathsf{Fonds\text{-}KlimaA} \sqcap \mathsf{hatTeil} : \mathsf{MB\_A30} \end{aligned}$$

Verwendung der zweiten Alternative, der Beschreibung als Menge von Konzeptzugehörigkeitsaussagen, ergibt, nun für das Individuum (d.h. die Fahrzeugvariante) p_1:

$$\{\ \mathsf{PKW}(p_1), \mathsf{hatTeil}(p_1, \mathsf{V12\_OttoM}), \mathsf{hatTeil}(p_1, \mathsf{Dänemark}),\\ \mathsf{hatTeil}(p_1, \mathsf{L}), \mathsf{hatTeil}(p_1, \mathsf{5\text{-}GangAG}), \mathsf{hatTeil}(p_1, \mathsf{Tempmatic}),\\ \mathsf{hatTeil}(p_1, \mathsf{Fonds\text{-}KlimaA}), \mathsf{hatTeil}(p_1, \mathsf{MB\_A30})\ \}$$

Die noch fehlenden Konzeptzugehörigkeiten (z.B. für das Konzept Motor) lassen sich dann aus den Terminologie-Axiomen ableiten.

Auf Grund der oben bereits erwähnten Offene-Welt-Semantik lässt sich auch hier aus dem Fehlen eines Fakts nicht auf dessen Ungültigkeit schließen. Daher ist das Fahrzeug p_1 (ebenso wie das äquivalente, weiter oben definierte Konzept P_1) nicht eindeutig bestimmt, sondern steht für all die Alternativen, die zumindest die angegebenen Fakten einschließen. Da also beispielsweise über das Enthaltensein von MB_A10_CD im Auftrag nichts gesagt ist, kann es ebenso gut vorhanden sein oder fehlen. Das gleiche gilt für weitere Sonderausstattungen, also z.B. für S_1. Wollte man die Fahrzeugspezifikation von p_1 eindeutig machen, so wären dazu die Rollenzugehörigkeitsaussagen $\neg\mathsf{hatTeil}(p_1, a_i)$ für alle nicht erwähnten Individuen a_i in obige Faktenmenge mitaufzunehmen. Alternativ könnte man auch eine veränderte Semantik, wie die der minimalen Modelle [McC80], verwenden.

Um nun die Gültigkeit eines so gegeben Auftrags bezüglich einer Produktspezifikation zu überprüfen, kann ein terminologiebezogener Erfüllbarkeitstest wie

$$\mathcal{T} \models P_1 \not\equiv \bot$$

verwendet werden. Darin bezeichnet $\mathcal{T}$ die Menge der terminologischen Axiome aus dem Konfigurationsbeispiel 3.2.

Bewertung

Durch die Verwendung von Beschreibungslogiken in der Produktkonfiguration wird das Augenmerk verstärkt auf die strukturelle Zusammensetzung des Produkts gelenkt, die Gruppierung von Teilen und die Bildung von konzeptionellen Zwischenstufen (z.B. Sonderausstattung, Getriebe in obigem Beispiel) werden gefördert. Dies hat den Vorteil[4], dass es zum Aufbau der Dokumentation erforderlich wird, sich Gedanken über strukturelle Gemeinsamkeiten verschiedener Produktvarianten zu machen. Dadurch

---

[4] Durch den zusätzlich verursachten Aufwand kann dies auch zu einem Nachteil werden.

kann eventuell auch eine Zusammenführung ähnlicher Komponenten und damit eine bessere Nutzung von in verschieden Modellen verwendbaren Teilen erreicht werden. Nicht zuletzt wird auch die Formulierung von Einschränkungen und Abhängigkeiten, die auf ganze Komponentengruppen zutreffen, durch die Bildung von konzeptionellen Zwischenstufen vereinfacht.

Verglichen mit der Produktstruktur haben in Beschreibungslogiken Prozessabläufe und Vorgehensweisen (z.B. wichtig in der Produktion) einen geringen Stellenwert. Dies hängt mit der deklarativen, auf Begriffsdefinitionen zugeschnittenen Ausrichtung der Beschreibungslogiken zusammen (wir werden später in den Dynamischen Logiken eine Alternative hierzu sehen), deren Ziel nicht die Abbildung von schrittweisen Abläufen ist. Solche können aber in der Produktkonfiguration eine erhebliche Rolle spielen, nicht nur in der Produktion. Beispielsweise ist es in Beschreibungslogiken schwierig, ausgehend von einer Teilspezifikation des Kunden nicht nur irgendeine passende und vollständige Produktkonfiguration zu finden, sondern dies auch in einer deterministischen, nachvollziehbaren Form. Die Spezifikation von bevorzugten Komponenten oder Standardkomponenten ist ohne eine Erweiterung der Sprache nicht möglich[5], da zwischen mehreren nach der Dokumentation möglichen Konfigurationen keine Präferenzen angegeben werden können.

McGuinness sieht deshalb den Einsatz von Beschreibungslogiken hauptsächlich in der interaktiven Produktkonfiguration [McG03]. Dabei soll der Benutzer Schritt für Schritt Anforderungen an die von ihm gewünschte Produktvariante benennen, deren gemeinsame Erfüllbarkeit dann jeweils über einen Subsumtionstest in der Beschreibungslogik überprüft wird. Die Präferenzen müssen hierbei dem System nicht bekannt sein, da sie vom Kunden selbst ganz direkt angegeben werden.

3.1.2 Feature-Logik

Ursprünglich unter der Bezeichnung Unifikationsgrammatiken in der Computer-Linguistik eingeführt [Kay79, KB82], etablierte Smolka Ende der 80er Jahre eine Variante dieser Grammatiken mit prädikatenlogischer Semantik unter dem Begriff “Feature Logic” [Smo88]. Inzwischen wird die Feature-Logik als eine besondere Beschreibungslogik angesehen, die auf Grund ihrer Herkunft aber andere Schwerpunkte setzt, andere Formalismen und

---

[5] Siehe z.B. Junkers Arbeiten zu solchen Erweiterungsmöglichkeiten [Jun01].

Notationen verwendet, und daher eine gesonderte Betrachtung verdient.

Wie die Beschreibungslogiken kennen die Feature-Logiken Individuen, atomare Konzepte und (funktionale) Rollen. Diese werden hier aber *Atome*, *Sorten* und *Features* genannt, wobei die Features als partielle Funktionen interpretiert werden und damit also eine Spezialisierung des Rollenbegriffs der Beschreibungslogiken darstellen.

Ein zentraler Begriff der Feature-Logik ist der *Feature-Term*. Ein Feature-Term wird, ähnlich einem Konzept in den Beschreibungslogiken, aus Atomen $(a, b, c, \dots)$, Sorten $(A, B, C, \dots)$ und Feature-Ketten $(p, q, r, \dots)$ aufgebaut, wobei die folgenden Konstruktoren zur Bildung von Feature-Termen $(S, T, U, \dots)$ zur Verfügung stehen:

$p : S$	Feature-Selektion
$p \downarrow q$	Feature-Übereinstimmung
$S \sqcup T$	Vereinigung
$S \sqcap T$	Schnitt
$\neg S$	Negation

Die Feature-Selektion entspricht dem Füllkonstruktor und hat auch dieselbe Semantik wie dieser. $p : S$ bezeichnet also all diejenigen Objekte, deren p-Feature einen Wert der Sorte S einnimmt; so bezeichnet beispielsweise der Term $\mathsf{Zylinder} : \mathbf{12}$ alle Gegenstände mit 12 Zylindern. Dabei wird die **12** als einelementige Sortenmenge aufgefasst.

Feature-Übereinstimmungen entsprechen den weiter oben beschriebenen Rollen-Wert-Abbildungen [BL85], wobei wir es hier nur mit funktionalen Rollen zu tun haben. Der Konstruktor $p \downarrow q$ entspricht der Rollenkettengleichheit $p \doteq q$. Die Feature-Ketten (p, q) ergeben sich typischerweise aus der Komposition von atomaren Features, d.h. beispielsweise $p = f_1 \circ \cdots \circ f_k$ für funktionale Rollen (Features) f_i. Oft wird dabei das verbindende Relationskompositionssymbol "$\circ$" weggelassen, so dass z.B. anstelle von $\mathsf{Motor} \circ \mathsf{Zylinder} : \mathbf{12}$ häufiger noch die Schreibweise $\mathsf{Motor}\ \mathsf{Zylinder} : \mathbf{12}$ anzutreffen ist.

Die verbleibenden Booleschen Operatoren auf Feature-Termen entsprechen den analogen Konzeptkonstruktoren in den Beschreibungslogiken. Auch die Semantik der beiden Logiken ist gleich. Interpretationen bestehen hier also aus einem Grundbereich $\mathcal{I}$ und einer Zuweisungsfunktion $(\cdot)^{\mathcal{I}}$, die Atomen Elemente des Grundbereichs, Sorten Teilmengen des Grundbereichs und Features binäre funktionale Relationen über dem Grundbereich zuordnet.

Zusammenfassend ergibt sich also im Vergleich zur Beschreibungslogik DL_0 das zusätzliche Vorhandensein von Feature-Übereinstimmungen, die Einschränkung auf funktionale Rollen sowie das Fehlen qualifizierter Mächtigkeitseinschränkungen.

$$P_1 = \begin{bmatrix} \textbf{PKW} & \\ \textsf{Motor:} & \begin{bmatrix} \textbf{Otto} \\ \textsf{Hubraum: } \textbf{6,0} \\ \textsf{Zylinder: } \textbf{12} \end{bmatrix} \\ \textsf{Getriebe:} & \begin{bmatrix} \textbf{Automatik} \\ \textsf{Gänge: } \textbf{5} \end{bmatrix} \\ \textsf{Lenkung: } \textbf{L} & \\ \textsf{Land: } \textbf{Dänemark} & \\ \textsf{Klima:} & \begin{bmatrix} \textbf{Tempmatic} \\ \textbf{Fonds-KlimaA} \end{bmatrix} \\ \textsf{Radio: } \textbf{MB\_A30} & \end{bmatrix}$$

Abbildung 3.3: Feature-Term einer Fahrzeugvariante.

In Abbildung 3.3 haben wir den Feature-Term dargestellt, der annähernd obiger Fahrzeug-Konzeptdefinition P_1 in der Beschreibungslogik DL_0 entspricht. Die grafische Darstellungsweise stammt von Smolka [Smo88] und wird von diesem als Matrixschreibweise von Feature-Termen (oder Datensätzen) bezeichnet. In dieser Darstellung werden Features in normalem Schrifttyp (ohne Auszeichnung) und Sorten in Fettdruck dargestellt. Die Diagramme sind so zu interpretieren, dass untereinander stehende, geklammerte Sorten als konjunktiv verknüpft zu einer Gesamtsorte S verstanden sein wollen, und Ausdrücke der Form $f : S$ als (übliche) Feature-Selektion. Die geklammerten Ausdrücke dieser Darstellung zeigen Ähnlichkeit zu “Record”-Datenstrukturen in Programmiersprachen wie Pascal, wobei dann die Features als Variablennamen und die Sorten als Typen verstanden werden können.

Unter Verwendung der üblichen Standardnotation ergibt sich für den Ausdruck P_1 die Sorte

$$\begin{aligned} &\textsf{PKW} \sqcap \textsf{Motor} : (\textsf{Otto} \sqcap \textsf{Hubraum} : 6,0 \sqcap \textsf{Zylinder} : 12) \sqcap \\ &\quad \textsf{Getriebe} : (\textsf{Automatik} \sqcap \textsf{Gänge} : 5) \sqcap \textsf{Lenkung} : \textsf{L} \sqcap \textsf{Radio} : \textsf{MB\_A30} \sqcap \\ &\qquad\qquad \textsf{Land} : \textsf{Dänemark} \sqcap \textsf{Klima} : (\textsf{Tempmatic} \sqcap \textsf{Fonds-KlimaA}) \ . \end{aligned}$$

Im Vergleich zu DL_0 müssen für Teilkomponenten eines Zusammenbaus jeweils unterschiedliche Feature-Namen verwendet werden. Anstelle der in der Beschreibungslogik einheitlich verwendeten hatTeil-Relation, mit der sowohl das Enthaltensein des Motors, als auch das des Getriebes und weiterer Fahrzeugbestandteile im PKW ausgedrückt werden konnte, muss in der Feature-Logik stets ein neues Feature eingebracht werden. Besonders bei in der Mächtigkeit a priori unbeschränkten Relationswertebereichen (wie z.B. der im Fahrzeug enthaltenen Sonderausstattungen) führt dies zu einer umständlichen Modellierung. Einen in manchen Fällen gangbaren Ausweg haben wir in den Klima-Sonderausstattungen von Abbildung 3.3 beschritten: Dort sind die beiden Klimatisierungskomponenten Tempmatic und Fonds-KlimaA zu einer Einheit zusammengefasst. Dies ist nur möglich, wenn man auch Atome zulässt, die zu mehreren atomaren Sorten gleichzeitig gehören. D.h. es muss in einer (erfüllenden) Interpretation $\mathcal{I}$ für ein $a \in \Delta^{\mathcal{I}}$ möglich sein, dass $a \in (A \sqcap B)^{\mathcal{I}}$ für zwei atomare Sorten A und B. In unserem Beispiel müsste der Grundbereich der Interpretation $\mathcal{I}$ also auch Elemente enthalten, die für beliebige Kombinationen der Komponenten stehen, also z.B. für die Kombination aus Tempmatic und Fonds-KlimaA. Die Spezifikation von weiteren Features dieser Komponenten (z.B. wenn an die Tempmatic, nicht aber die Fonds-KlimaA weitere Anforderungen gestellt werden) bleibt aber trotzdem schwierig, und muss notfalls über Feature-Namen, die zusätzlich noch Sorteninformationen mit beinhaltenden (z.B. Tempmatic-Leistung) ausgeglichen werden. Eine Alternative wäre die Verwendung individueller Features für jede Subkomponente des Fahrzeugs, im Klimaanlagen-Beispiel also eine Spezifikation der Form

$$\begin{aligned} &\mathsf{Klima\_1} : \mathbf{Tempmatic} \\ &\mathsf{Klima\_2} : \mathbf{Fonds\text{-}KlimaA} \quad . \end{aligned}$$

Abschließend lässt sich sagen, dass durch die Anwendung der Feature-Logik für Konfigurationsaufgaben sich das Augenmerk von der Beschreibung der Produktstruktur (die in den Beschreibungslogiken im Vordergrund steht) zu den Ressourcen verschiebt, die hier einfacher – und in der Matrixschreibweise vielleicht auch intuitiver – modelliert werden können. So könnte die farbliche Übereinstimmung zweier Fahrzeugkomponenten in Feature-Logik beispielsweise durch den Term

$$\mathsf{Mittelkonsole\ Farbe} \downarrow \mathsf{Schalthebel\ Farbe}$$

relativ prägnant festgelegt werden. In der Konfiguration wird aber wahrscheinlich die Übereinstimmung von Attributen häufig nicht ausreichend sein. Durch die detaillierte Modellierung von Komponentenattributen werden Zahlenwerte und Operationen auf diesen nötig, um z.B. den Leistungsverbrauch von Komponenten aufsummieren zu können. Dies ist in der hier dargestellten einfachen Feature-Logik nicht möglich. Eine Erweiterung ähnlich den konkreten Grundbereichen der Beschreibungslogiken scheint naheliegend und durchführbar.

3.1.3 Prädikatenlogik

Die Verwendung der Prädikatenlogik erster Stufe (PL1) zur Codierung von Konfigurationsproblemen kann aus zwei unterschiedlichen Sichtweisen begründet werden. Zum einen ist PL1 eine so ausdrucksstarke Sprache, dass sich damit fast alle auftretenden Beschreibungskonstrukte formalisieren lassen.[6] Darüberhinaus ist PL1 recht weit verbreitet und im Vergleich zu anderen Logiken besser etabliert, wodurch die Bedeutung von Formeln dieser Logik von einem größeren Personenkreis direkt erschlossen werden kann. Zum zweiten sind aus fast allen anderen Formalismen, einschließlich der Beschreibungslogiken, Übersetzungen in die Prädikatenlogik bekannt, so dass sich PL1 als einheitliche "Basislogik" etabliert hat. Dies hängt auch damit zusammen, dass PL1 die in der Mathematik üblicherweise verwendete Sprache ist und somit die Semantik anderer Logiken auch meist in dieser ausgedrückt wird.

Transformation von anderen Logiken nach PL1

Wir wollen nun die Transformation in Prädikatenlogik für obige Beschreibungslogik DL_0 beschreiben. Dabei folgen wir Borgidas Ausführungen, verwenden aber eine wesentlich kompakter darstellbare Menge von Transformationsregeln. Konzeptausdrücke C und Rollenausdrücke R werden dabei durch eine rekursiv definierte Funktion τ in die Prädikatenlogik, genauer das 3-Variablen-Fragment mit zählenden Quantoren, $\mathcal{C}_3$, übersetzt. Tabelle 3.2 gibt die Definition der Übersetzungsfunktion wieder.

Die Zeilen der Tabelle sind für die Transformation so zu lesen, dass die DL_0-Ausdrücke der linken Spalte durch die $\mathcal{C}_3$-Formeln der rechten Spalte ersetzt werden. Dabei soll $F[x \rightleftharpoons y]$ das Vertauschen aller Variablenvorkommen von x, ob frei oder gebunden, durch y und umgekehrt in F

---

[6] Eine prominente Ausnahme ist der transitive Abschluss von Relationen, der sich erst in der Prädikatenlogik zweiter Stufe (PL2) ausdrücken lässt.

$C \in \Gamma$	$\tau(C) \in \mathcal{C}_3$
$\top$	$\top$
$D \sqcup E$	$\tau(D) \vee \tau(E)$
$\neg D$	$\neg\tau(D)$
$\geq n\, R.D$	$\exists_n y(\tau(R) \wedge \tau(D)[x \rightleftharpoons y])$
$\{a_1, \dots, a_n\}$	$x = a_1 \vee \dots \vee x = a_n$
$A \in \Gamma_0$	$A(x)$

$R \in \Pi$	$\tau(R) \in \mathcal{C}_3$
$S \circ T$	$\exists z(\tau(S)[y \rightleftharpoons z] \wedge \tau(T)[x \rightleftharpoons z])$
$Q \in \Pi_0$	$Q(x, y)$

Tabelle 3.2: Übersetzung τ von DL_0-Ausdrücken in das Fragment $\mathcal{C}_3$ der Prädikatenlogik.

ausdrücken, $\exists_n$ bezeichnet den weiter oben bereits verwendeten zählenden Existenzquantor. Unter Verwendung weiterer nicht in F vorkommender Variablen $x_1, \dots, x_n$ und der Gleichheitsrelation kann dieser anhand der Äquivalenz

$$\exists_n x\, F(x) \quad \Leftrightarrow \quad \exists x_1, \dots, x_n \left(F(x_1) \wedge \dots \wedge F(x_n) \wedge \bigwedge_{1 \leq i < j \leq n} x_i \neq x_j \right)$$

auch eliminiert werden. Die Übersetzung eines Konzepts C liefert immer eine PL1-Formel mit freier Variable x, die Übersetzung einer Rolle eine mit den beiden freien Variablen x und y. Man erkennt hier auch, dass unter Wegfall der Rollenkomposition die Regeln eine Übersetzung in das entscheidbare 2-Variablen-Fragment $\mathcal{C}_2$ definieren.

Anwendung der Transformation τ auf die in Abbildung 3.2 angegebene

Fahrzeugterminologie liefert dann auszugsweise

$$\begin{aligned}
\mathsf{PKW}(x) &\Rightarrow \exists_1^1 y(\mathsf{hatTeil}(x,y) \wedge \mathsf{Motor}(y)) \\
&\vdots \\
\mathsf{PKW}(x) &\Rightarrow \exists_1^1 y(\mathsf{hatTeil}(x,y) \wedge \mathsf{Land}(y)) \\
\mathsf{PKW}(x) &\Rightarrow \forall y(\mathsf{hatTeil}(x,y) \Rightarrow (\mathsf{Motor}(y) \vee \\
&\qquad \mathsf{Getriebe}(y) \vee \cdots \vee \mathsf{Sonderausstattung}(y))) \\
\mathsf{Motor}(x) &\Leftrightarrow \mathsf{OttoM}(x) \vee \mathsf{DieselM}(x) \\
\mathsf{Getriebe}(x) &\Leftrightarrow \mathsf{AutomatikG}(x) \vee \mathsf{ManuellesG}(x) \\
\mathsf{Lenkung}(x) &\Leftrightarrow x = \mathsf{links} \vee x = \mathsf{rechts} \\
&\vdots \\
\mathsf{OttoM}(x) \wedge \mathsf{DieselM}(x) &\Rightarrow \bot \\
&\vdots \\
\exists y(\mathsf{hatTeil}(x,y) \wedge & \\
y = \mathsf{Thermotronic}) &\Rightarrow \neg\exists y(\mathsf{hatTeil}(x,y) \wedge y = \mathsf{R4\_DieselM}) \\
\exists y(\mathsf{hatTeil}(x,y) \wedge y = \mathsf{S}_2) &\Rightarrow \neg\exists y(\mathsf{hatTeil}(x,y) \wedge y = \mathsf{Mexiko}) \wedge \\
&\qquad \exists y(\mathsf{hatTeil}(x,y) \wedge \mathsf{ManuellesG}(y))
\end{aligned}$$

Dabei bezeichnen wir mit $\exists_m^n x\, F$ die Aussage, dass zwischen m und n verschiedene Variablenwerte x existieren, für die F gilt, formal also $\exists_m^n x\, F \Leftrightarrow \exists_m x\, F \wedge \neg\exists_{n+1} F$. Die einzelnen Formeln sind als universell quantifiziert zu verstehen, die sich ergebende Gesamtformel ist dann die Konjunktion der universell abgeschlossenen Teilformeln jeder Zeile.

Üblicherweise wird zur Definition der Semantik der Prädikatenlogik eine Interpretation der Form

$$\mathcal{I} = (\Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}}, \nu^{\mathcal{I}})$$

verwendet, in der die nicht-leere Menge $\Delta^{\mathcal{I}}$ den Grundbereich der Interpretation (das Universum), $(\cdot)^{\mathcal{I}}$ die Interpretation von Prädikat- und Funktionssymbolen durch konkrete Prädikate und Funktionen (in unserem Fall nur Konstanten) über dem Grundbereich $\Delta^{\mathcal{I}}$, und $\nu^{\mathcal{I}}$ eine Belegung der Variablen durch Objekte des Grundbereichs darstellt. Gültigkeit einer Formel bedeutet dann Gültigkeit in allen möglichen Interpretationen.

In unserer Anwendung wäre die Einschränkung auf nur einen (“konkreten”) Grundbereich allerdings ausreichend. Dieser Grundbereich umfasste die atomaren und zusammengesetzten Komponenten, die sich z.B. aus dem

UML-Diagramm einfach ablesen ließen. In jeder im Modell vorkommenden Abstraktionshierarchie sind die konkreten Instanzen, die individuellen Komponenten ebenfalls umfassend, ausreichend. In unserem Fahrzeugbeispiel ergäbe sich für diesen Grundbereich $\Delta_{\mathrm{K}}^{\mathcal{I}}$ also

$$\begin{aligned}\Delta_{\mathrm{K}}^{\mathcal{I}} = \{&\mathsf{PKW}, \mathsf{V6\_OttoM}, \mathsf{V12\_OttoM}, \mathsf{R6\_OttoM},\\ &\mathsf{R4\_DieselM}, \mathsf{R5\_DieselM}, \mathsf{links}, \mathsf{rechts},\\ &\mathsf{4\text{-}GangAG}, \mathsf{5\text{-}GangAG}, \mathsf{5\text{-}GangMG},\\ &\mathsf{Tempmatic}, \mathsf{Thermotronic}, \mathsf{Fonds\text{-}KlimaA},\\ &\mathsf{MB\_A30}, \mathsf{MB\_A10\_CD}, \mathsf{S}_1, \ldots \mathsf{S}_k,\\ &\mathsf{Bahrein}, \mathsf{Dänemark}, \mathsf{Japan}, \mathsf{Mexiko}, \mathsf{Mongolei}, \mathsf{USA}, \ldots\ \} \enspace .\end{aligned}$$

Um automatische Beweise über nur diesem Grundbereich durchzuführen, wären dann auch die Deduktionsalgorithmen entsprechend zu modifizieren. Das dabei entstehende Beweisproblem erinnert beträchtlich an Model-Checking [CGP00], und hat mit diesem die Festlegung auf nur einen Grundbereich gemein. Abweichend ist aber die Semantik der Prädikate und Funktionen, die nun nicht mehr festgelegt, sondern variabel ist. Gültigkeit einer Formel bedeutet damit also, dass sie im festgelegten Grundbereich $\Delta_{\mathrm{K}}^{\mathcal{I}}$ unter allen möglichen Interpretationen $(\cdot)^{\mathcal{I}}$ der Prädikate und Funktionen und unter allen Variabenbelegungen $\nu^{\mathcal{I}}$ gültig ist.

Man kann dies gewissermaßen als Gegenpol zu den interpretierten Funktionssymbolen sehen, die sowohl in der Logikprogrammierung als auch in der Datenbanktheorie anzutreffen sind [KKR90, BL00]. Bei diesen bleibt der Grundbereich (zumindest zum Teil) variabel, die Bedeutung der Funktionen und Prädikate wird dahingegen eindeutig festgelegt.

Wieder zurück zu unseren interpretierten konkreten Grundbereichen; hier können zur Prüfung der Allgemeingültigkeit oder Erfüllbarkeit einer Formel spezielle Verfahren eingesetzt werden, die vom festgelegten Universum profitieren. Wir werden weiter unten im Zusammenhang mit der Aussagenlogik ein einfaches Beweisverfahren angeben, das für endliche konkrete Grundbereiche, wie sie häufig in der Konfiguration anzutreffen sind, ausreichend ist und gewinnbringend eingesetzt werden kann.

Alternative direkte Beschreibung der Konfiguration in PL1

Verzichtet man darauf, eine schon gegebene Konfigurationsbeschreibung in einer von PL1 unterschiedlichen Logik als Ausgangspunkt zu nehmen, so ergeben sich natürlich weitere Modellierungsalternativen, von denen wir eine vorstellen wollen, die sich durch die direkte Modellierung der Komponentenauswahl hervorhebt. Um diese Auswahl auszudrücken, verwenden wir das einstellige Prädikat $\mathsf{ausgewählt}(x)$, das genau dann wahr sein soll, wenn vom Kunden Komponente x ausgewählt wurde. Anstelle einer eigenständigen Beschreibung der Produktstruktur, wird diese hier nur in Verbindung mit der Korrektheit (oder Gültigkeit) eines Auftrags definiert. So ergibt sich für obiges Fahrzeugbeispiel die Gültigkeit eines PKWs als

$$\begin{aligned}
\mathsf{gültig}(\mathsf{PKW}) &\Leftrightarrow \exists^1_1 x(\mathsf{ausgewählt}(x) \land \mathsf{Motor}(x)) \land \\
&\quad \vdots \\
&\quad\; \exists^1_1 x(\mathsf{ausgewählt}(x) \land \mathsf{Land}(x)) \land \\
&\quad\; \forall x(\mathsf{ausgewählt}(x) \Rightarrow \mathsf{Motor}(x) \lor \cdots \lor \mathsf{Sonderausstattung(x)}) \\
\mathsf{Motor}(x) &\Leftrightarrow \mathsf{OttoM}(x) \lor \mathsf{DieselM}(x) \\
\mathsf{Getriebe}(x) &\Leftrightarrow \mathsf{AutomatikG}(x) \lor \mathsf{ManuellesG}(x) \\
\mathsf{Lenkung}(x) &\Leftrightarrow x = \mathsf{links} \lor x = \mathsf{rechts} \\
&\vdots
\end{aligned}$$

Die Abstraktionshierarchien wurden dabei wie in der vorangehenden Formalisierung in PL1 realisiert. Anders als bei dieser haben wir die Komponentenhierarchie aber nicht auf das zweistellige Prädikat $\mathsf{hatTeil}$ abgebildet, sondern sie komplett in die Gültigkeitsdefinition miteinbezogen. Dadurch geht zwar ein Teil der strukturellen Information verloren (es lassen sich so nur “flache” Hierarchien abbilden), andererseits kann auf das zweistellige Enthaltensein-Prädikat verzichtete werden, was sich in der weiter unten beschriebenen Transformation nach Aussagenlogik als vorteilhaft erweist (siehe Abschnitt 3.1.5).

Die Inkompatibilitäten und Abhängigkeiten aus obiger Fahrzeugproduktstruktur wollen wir der Vollständigkeit halber in Auszügen auch noch angeben, wobei wir zwei zusätzliche Hilfsprädikate ($\mathsf{inkompatibel}$ und $\mathsf{erfordert}$) der Lesbarkeit halber einführen:

$$\begin{gathered}
\neg(\mathsf{OttoM}(x) \land \mathsf{DieselM}(x)) \\
\mathsf{inkompatibel}(x, y) \;\Leftrightarrow\; \neg(\mathsf{ausgewählt}(x) \land \mathsf{ausgewählt}(y)) \\
\mathsf{erfordert}(x, y) \;\Leftrightarrow\; \mathsf{ausgewählt}(x) \Rightarrow \mathsf{ausgewählt}(y) \\
\mathsf{inkompatibel}(\mathsf{Thermotronic}, \mathsf{R4\_DieselM}) \\
\mathsf{inkompatibel}(\mathsf{S}_2, \mathsf{Mexiko}) \\
\mathsf{ausgewählt}(\mathsf{S}_2) \Rightarrow \exists y(\mathsf{ausgewählt}(y) \land \mathsf{ManuellesG}(y))
\end{gathered}$$

Eine dieser zweiten Codierungsvariante ähnliche Form haben wir zur Formalisierung der SIEMENS-Gerätestrukturen (siehe Abschnitt 2.3.2) verwendet, in einer Machbarkeitsstudie ist dies detailliert dargestellt [SK02]. Darin wird auch eine weitaus allgemeinere Methode zur Übersetzung von in XML spezifizierten Produktstrukturen nach Aussagenlogik beschrieben.

3.1.4 Constraint-Logik

Die Constraint-Logik befasst sich mit der (kompakten) Beschreibung endlicher oder unendlicher Datenmengen durch logische Formeln. So kann z.B. die unendliche Menge $U = \{1, 3, 5, 7, \dots\}$ der ungeraden natürlichen Zahlen durch die Bedingung (den *Constraint*)

$$\{x \in \mathbb{N} \mid \exists y \in \mathbb{N}\ (x = 2 \cdot y)\},$$

also durch eine kompakte, endliche Formel beschrieben werden. Die typische Fragestellung der Constraint-Logik ist die Erfüllbarkeit eines Systems von Bedingungen (*constraint satisfaction problem, CSP*).

Sehr häufig wird die Constraint-Logik zusammen mit endlichen Bereichen verwendet. Bei diesen ist jeder Variablen (z.B. x) ein endlicher Wertebereich (D_x) fest zugeordnet. Genauer ist dies in der folgenden Definition erfasst:

Definition 3.1.3 *Ein* ***Constraint-Problem*** P *ist ein Tripel* $P = (V, D, C)$*, wobei* $V = (v_1, \dots, v_n)$ *eine geordnete Variablenmenge,* $D = (D_1, \dots, D_n)$ *eine geordnete Bereichsmenge mit endlichen* D_i *und* $C = \{C_1, \dots, C_m\}$ *eine Menge von Bedingungen ist. Jede Bedingung* $C_j = (R_j, I_j)$ *besteht aus einer Relation* R_j *über einer durch die Indexmenge* $I_j = \{i_{j,1}, \dots, i_{j,k_j}\}$ *bestimmen Teilmenge der Variablenbereiche, d.h.* $R_j \subseteq D_{i_{j,1}} \times \dots \times D_{i_{j,k_j}}$*, wobei* $1 \leq i_{j,1} < i_{j,2} < \dots < i_{j,k_j} \leq n$*. Die Relationen* R_j *werden als zulässige Kombinationen von Variablenwerten interpretiert. Eine Variablenbelegung* $\beta : V \to D_1 \cup \dots \cup D_n$ *ist eine Lösung des Constraint-Problems* P*, wenn* $\beta(v_i) \in D_i$ *für* $1 \leq i \leq n$*, und für alle* j *mit* $1 \leq j \leq m$ *gilt* $(\beta(v_{i_{j,1}}), \dots, \beta(v_{i_{j,k_j}})) \in R_j$*. Das Constraint-Problem* P *heißt erfüllbar, wenn es mindestens eine Lösung* β *besitzt.*

Wir wollen die Definition anhand eines Beispiels erläutern. Sei $P = (V, D, C)$ das Constraint-Problem mit $V = (x, y, z)$, $D = (\{a, b, c\}, \{a, d\}, \{e, f\})$ und

$C = \{C_1, C_2\}$, wobei

$$\begin{aligned} C_1 &= (R_1, I_1) = (\{(a,a),(b,d)\},\{1,2\}) \\ C_2 &= (R_2, I_2) = (\{(d,e),(a,f)\},\{2,3\}) \ . \end{aligned}$$

Die Bedingung C_1 erklärt also z.B. die Variablenkombination $x = b$ und $y = d$ für gültig, die Kombination $x = b, y = a$ hingegen nicht. Die beiden einzigen Lösungen des Constraint-Problems P sind die Variablenbelegungen $\beta_1 = \{x \mapsto a, y \mapsto a, z \mapsto f\}$ und $\beta_2 = \{x \mapsto b, y \mapsto d, z \mapsto e\}$.

Anstatt die Relationen der Constraints explizit anzugeben wird häufiger noch eine implizite Darstellung mittels Formeln gewählt. Für diese Formeln wird meist eine Variante der Aussagenlogik mit den Konnektiven $\neg, \vee, \wedge, \Rightarrow$ und $\Leftrightarrow$ gewählt, in der die atomaren Prädikate Gleichheitsprädikate sind, also z.B. $x = d$ für $x \in V$ und $d \in D_x$, wobei D_x der zu x gehörige Wertebereich ist. Man findet auch oft die verkürzte Schreibweise $x \in \{d_1, \ldots, d_k\}$ anstelle von $x = d_1 \vee \cdots \vee x = d_k$. Die beiden Bedingungen $C_{1/2}$ unseres Beispiels können dann z.B. in der aussagenlogischen Sprache so ausgedrückt sein:

$$\begin{aligned} C_1 : \quad & x = a \Rightarrow y = a \\ & y = d \Leftrightarrow x \notin \{a, c\} \\ & x \neq c \\ C_2 : \quad & z = e \Leftrightarrow y = d \end{aligned}$$

Die Indexmengen I_j der Variablenbereiche ergeben sich hierbei direkt aus den in den Bedingungen vorkommenden Variablen und müssen daher nicht extra mit angegeben werden.

Konfigurationsprobleme lassen sich in der Constraint-Logik codieren, indem für jede Auswahlmöglichkeit einer Komponente eine neue Variable mit endlichem Bereich eingeführt wird. Die Werte dieses endlichen Bereichs entsprechen dann den einzelnen Wahlmöglichkeiten. So wird auch die Abstraktionshierarchie, die demnach “flach” sein muss, abgebildet. Die Komponentenhierarchie wird, sofern erforderlich, über zusätzliche Constraints konstruiert, wobei das Nichtvorhandensein einer Komponente durch die Zuweisung des Wertes undefiniert an die Variable ausgedrückt wird. Mehrfaches Vorhandensein einer Komponente muss durch Vorhandensein mehrerer Variablen desselben Wertebereichs, nämlich den der mehrfach vorhandenen Komponente, modelliert werden. Zusätzliche Nebenbedingungen werden über zusätzliche Constraints festgelegt.

So ergibt sich für unsere Fahrzeugkonfiguration aus Abbildung 3.1 das folgende Constraint-Problem $P = (V, D, C)$. Zuerst die Variablen und deren Bereiche:

i	v_i	D_i
1	Motor	{V6_OttoM, V12_OttoM, R6_OttoM, R4_DieselM, R5_DieselM}
2	Lenkung	{links, rechts}
3	Getriebe	{4-GangAG, 5-GangAG, 5-GangMG}
4	Land	{Dänemark, Japan, Mexiko, ...}
5	KlimaA$_1$	{Tempmatic, Thermotronic, undefiniert}
6	KlimaA$_2$	{Fonds-KlimaA, undefiniert}
7	RadioA	{MB_A30, MB_A10_CD, undefiniert}
8	SonstigeA$_1$	{S$_1$, undefiniert}
9	SonstigeA$_2$	{S$_2$, undefiniert}
⋮		

Die Komponentenstruktur ist zum Großteil bereits in den Variablenwertebereichen mit eincodiert. Da es z.B. für die Variable Motor keinen Wert undefiniert gibt, muss diese zwangsläufig auf eine der vorhandenen Komponentenwerte gesetzt werden, wodurch das Vorhandensein genau eines Motors erzwungen wird. Bleiben also nur noch die beiden in Abbildung 3.2 angegebenen Restriktionen zu behandeln. Diese lassen sich durch

$$C_1: \quad \text{KlimaA}_1 = \text{Thermotronic} \Rightarrow \text{Motor} \neq \text{R4\_DieselM}$$
$$C_2: \quad \text{SonstigeA}_2 = \text{S}_2 \Rightarrow \neg\text{Land} = \text{Mexiko} \wedge \text{Getriebe} = \text{5-GangMG}$$

codieren. Bei der gewählten Codierung geht leider die Information einer mehrstufigen Abstraktionshierarchie verloren, da nur die oberste und unterste Ebene repräsentiert sind. Für die Motoren lässt sich somit z.B. auch keine Unterteilung in Otto- und Dieselmotoren vornehmen.

Ein bisher nicht betrachteter Vorteil der Constraint-Logik ist die Möglichkeit andere entscheidbare Theorien zu integrieren. So kann beispielsweise durch die Erweiterung um Variablen mit reellem Grundbereich sowie Operatoren für Addition und Multiplikation eine Modellierung zusätzlicher numerischer Eigenschaften der Komponenten (wie in ressourcenbasierten Systemen erforderlich) relativ einfach realisiert werden.

3.1.5 Aussagenlogik

Da man in der Produktkonfiguration (zumindest in der hier betrachteten Form) nur mit räumlich beschränkten Produkten, die aus endlich vielen, zuvor bereits bekannten Komponenten zusammengesetzt sind, zu tun hat, lassen sich sämtliche zuvor beschriebenen kombinatorischen Formalisierungen in endlichen Strukturen interpretieren und daher in Aussagenlogik umcodieren. Anstelle einer Umcodierung kann man natürlich auch die Produktstruktur direkt in Aussagenlogik beschreiben.

Neben der Festlegung des endlichen Grundbereiches ist die grundlegende Idee bei der Übersetzung nach Aussagenlogik in allen zuvor beschriebenen Formalismen mit Ausnahme der Constraint-Logik die Anwendung der beiden Äquivalenzen

$$\forall x F(x) \Leftrightarrow \bigwedge_{k \in \Delta_\mathrm{K}} F(k) \qquad \text{und}$$

$$\exists x F(x) \Leftrightarrow \bigvee_{k \in \Delta_\mathrm{K}} F(k)$$

für den festgelegten endlichen Grundbereich Δ_K, um die Quantoren aus den Formeln zu entfernen. Funktionssymbole werden zuvor, sofern vorhanden, durch Prädikate ersetzt, wobei jedes n-stellige Funktionssymbol f durch ein neues $n+1$-stelliges Prädikatensymbol P_f ersetzt werden muss (für $n > 1$), dessen $n+1$-te Komponente dem Funktionswert entspricht. Neben der Ersetzung muss noch die Funktionalität des neuen Prädikats P_f codiert werden, z.B. durch den Satz

$$\forall x_1, \ldots, x_n, y, z \; (P_f(x_1, \ldots, x_n, y) \wedge P_f(x_1, \ldots, x_n, z) \Rightarrow y = z) \; .$$

Verschachtelte Funktionssymbole werden schon vorher wie üblich aufgelöst, also z.B. durch Ersetzen von $f(\ldots, g(x), \ldots)$ durch $f(\ldots, y, \ldots) \wedge y = g(x)$, wobei y eine neue, bisher nicht in der Gesamtformel vorkommende Variable ist. Letztendlich erhält man so quantoren- und variablenfreie Formeln, in denen nur noch die aussagenlogischen Konnektive und grundierte Prädikate vorkommen. Jedes grundierte Prädikat $P(c_1, \ldots, c_n)$ wird dann zu einer Aussagevariable $P_{c_1, \ldots, c_n}$, man erhält eine rein aussagenlogische Formel.

Im Falle universell quantifizierter Axiome ergibt sich so die Ersetzung aller Axiome durch deren sämtliche (Grund-)Instanzen. Obiges Axiom aus der Übersetzung der Fahrzeugterminologie von Beschreibungslogik nach Prädikatenlogik,

$$\mathsf{Motor}(x) \Leftrightarrow \mathsf{OttoM}(x) \vee \mathsf{DieselM}(x),$$

wird unter Verwendung des konkreten Grundbereichs $\Delta_{\mathrm{K}}^{\mathcal{I}}$ demnach zu der Grundinstanzmenge

$$\begin{aligned}
\mathsf{Motor}(\mathsf{PKW}) &\Leftrightarrow \mathsf{OttoM}(\mathsf{PKW}) \vee \mathsf{DieselM}(\mathsf{PKW}) \\
\mathsf{Motor}(\mathsf{V6\_OttoM}) &\Leftrightarrow \mathsf{OttoM}(\mathsf{V6\_OttoM}) \vee \mathsf{DieselM}(\mathsf{V6\_OttoM}) \\
&\vdots \\
\mathsf{Motor}(\mathsf{Tempmatic}) &\Leftrightarrow \mathsf{OttoM}(\mathsf{Tempmatic}) \vee \mathsf{DieselM}(\mathsf{Tempmatic}) \\
&\vdots
\end{aligned}$$

Die meisten dieser Axiome sind trivial erfüllt, da die Konstanten des Grundbereichs nicht zum jeweiligen Konzept gehören, und so beide Seiten der Äquivalenz falsch werden. Zusätzliche Betrachtung der Konzeptzugehörigkeiten kann hier eine sofortige deutliche Reduktion der Anzahl der Axiominstanzen bringen, die allerdings auch in einem nachfolgenden Vereinfachungsschritt leicht vorgenommen werden kann. Die Gleichheitsprädikate können für die schon ursprünglich in der Formel auftretenden Konstanten direkt ausgewertet werden, da nur gleiche Bezeichner aufgrund der *unique name assumption* als gleiche Objekte des Grundbereichs betrachtet werden dürfen.

Im Falle der Umcodierung von Formeln der Constraint-Logik nach Aussagenlogik, um beispielsweise SAT-Checker einsetzen zu können, ist anders vorzugehen. Hier sind die endlichen Bereiche mit einer Mächtigkeit größer als zwei auf den Bereich der Booleschen Konstanten $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ zu reduzieren. Die einfachste Möglichkeit ist, eine Variable v mit Wertebereich $D = \{d_1, \ldots, d_n\}$ durch n aussagenlogische Variablen $v_1, \ldots, v_n$ zu ersetzen, wobei v_i für die Aussage $v = d_i$ steht. Zusätzlich muss dann noch spezifiziert werden, dass die Variable v immer genau einen der möglichen Werte d_i annimmt, also

$$v_1 \vee \cdots \vee v_n \quad \text{und} \quad \bigwedge_{1 \leq i < j \leq n} \neg(v_i \wedge v_j) \ . \tag{3.2}$$

Eine andere Möglichkeit ist eine binäre Codierung der n Variablenwerte durch $k = \lceil \log n \rceil$ Aussagevariablen. Dabei wird die Aussage $v = d_i$ durch das i entsprechende Bitmuster $(b_{k-1}, \ldots, b_0)$, d.h. die k-stellige binäre Darstellung von i ersetzt. Im Gegensatz zur ersten Lösung entfällt hierbei die Codierung des Ausschlusses der verschiedenen Variablenwerte. Dafür muss nun jede Aussage $v = d_i$ durch die kompliziertere Konjunktion der k entsprechenden Aussagevariablen b_i ersetzt werden. Zum Beispiel ergibt sich bei einem 5-elementigen Wertebereich für die Aussage $v = d_2$ die äquivalente Codierung $\neg b_2 \wedge b_1 \wedge \neg b_0$ in Aussagenlogik.

Bei einer Auswahl der beiden Alternativen gilt es also abzuwägen zwischen der erhöhten Variablenanzahl bei kürzerer Codierung der Formel in der ersten Variante und einer geringeren Variablenanzahl und komplexerer Formel in der zweiten Variante. Genauer gesagt ergibt sich, ausgehend von Wertebereichen mit maximal m Werten, also $\max |D_i| = m$, für eine Formel mit n Endliche-Bereichs-Variablen und f atomaren Ausdrücken (Aussagen der Form $v_i = d_j$) in der ersten Transformationsvariante eine aussagenlogische Formel mit $n \cdot m$ Variablen und $f + n \cdot (m + \frac{1}{2}m(m-1))$ Atomen (Literalvorkommen), während in der zweiten Variante eine Formel mit $n \cdot \lceil \log m \rceil$ aussagenlogischen Variablen und $f \cdot \lceil \log m \rceil$ Literalvorkommen entsteht.

In unserer Formalisierung des im IBM System Automation Manager enthaltenen Expertensystems haben wir erstere Variante gewählt [SLSK02].

Bei der Codierung von endlichen Bereichen sowie zur Modellierung von Komponentengruppen in der Konfiguration hat sich herausgestellt, dass die Verwendung eines Mächtigkeits- oder Selektionsoperators S_a^b häufig gewinnbringend sein kann. Bei diesem handelt es sich um einen n-stelligen Operator, der zusätzlich zwei Parameter $a, b \in \mathbb{N}$ besitzt. Intuitiv drückt $\mathrm{S}_a^b(F_1, \ldots, F_n)$ aus, dass zwischen a und b der Formeln F_i wahr sind. Ein solcher Operator lässt sich dann beispielsweise in der obigen ersten Codierungsvariante zur Festlegung der Eigenschaft 3.2 verwenden, die sich damit zu

$$\mathrm{S}_1^1(v_1, \ldots, v_n)$$

verkürzt, womit aus einer Formel quadratischer Größe in n eine Formel wird, deren Größe linear in n ist.

Kaiser hat einen aussagenlogischen Beweiser entwickelt, der diesen Operator miteinschließt und in einem dem Davis-Putnam-Algorithmus ähnlichen Verfahren speziell behandelt [Kai01]. Dieser Beweiser wird auch in unserem System BIS, das wir weiter unten noch ausführlich beschreiben werden, eingesetzt [SKK00, SKK03]. In anderen Anwendungen der Verifikation hat sich dieser Operator auch als hilfreich erwiesen [SK02, SLSK02]. Bejar *et al.* haben ein ähnliches Verfahren, das jedoch auf mehrwertiger Logik basiert, vorgestellt und heben dessen Vorzüge in der aussagenlogischen Codierung von Problemen aus der Praxis hervor [BCF$^+$01].

3.1.6 Modallogiken

Modallogiken erweitern klassische Logiken um sogenannte Modalitäten wie Möglichkeit und Notwendigkeit. Häufig sind sie anzutreffen als Erweiterungen der Aussagenlogik und werden dort auch verwendet, um Aussagen über zeitliche Zusammenhänge von Propositionen, wie deren Abhängigkeit von Zukunft und Vergangenheit, zu machen (Temporallogik) oder das Wissen und Fürmöglichhalten von Aussagen auszudrücken (siehe z.B. Hintikka [Hin62]).

Typischerweise wird die Logik um zwei (duale) Operatoren, $\Diamond$ und $\Box$, erweitert, von denen ersterer z.B. die Möglichkeit und letzterer die Notwendigkeit ausdrückt. So kann der Ausdruck $\Diamond P$ z.B. als Aussage "es ist möglich, dass P gilt" verstanden werden, und analog dazu $\Box P = \neg\Diamond\neg P$ als "P gilt notwendigerweise" oder "es ist nicht möglich, dass P nicht gilt".

Auch wenn in der Konfiguration vielfältige Anwendungsgebiete für Modallogiken denkbar sind (z.B. Verwendung von Temporallogik um Produktionsprozesse zu modellieren), werden wir im folgenden nur eine spezielle Variante, die dynamische Logik, vorstellen. Diese nimmt in der Programmverifikation, speziell von regelbasierten Expertensystemen, eine herausragende Stellung ein.

3.2 Dynamische Aussagenlogik (PDL)

Dynamische Logiken [Har84, HKT00] sind Modallogiken, die die Formulierung und das Beweisen von Aussagen über Programme erlauben. Solche Aussagen können beispielsweise lauten: *"Das Programm P terminiert immer"* oder *"Nach Ablauf des Programms P' mit Eingabe $x > 0$ gilt immer $y = \sqrt{x}$"*.

In dieser Arbeit beschränken wir uns auf eine verbreitete Variante der Dynamischen Logik, die *Dynamische Aussagenlogik (propositional dynamic logic, PDL)* [FL77, FL79, Pra79]. Diese lässt als Programmvariablen nur Boolesche Variablen zu. Die Logik PDL ist einerseits so eingeschränkt, dass sie entscheidbar, und damit automatischen Beweisverfahren besser zugänglich ist, andererseits aber mächtig genug, um die in der Produktkonfiguration auftretenden Probleme und Programme zu formalisieren.

3.2.1 Syntax

Die Sprache der Dynamischen Aussagenlogik besteht aus zwei Sorten von Ausdrücken: solche für Programme und solche für Aussagen. Komplexe Programme werden aus atomaren Programmen mittels verschiedener programmkonstruierender Operatoren gebildet, analog werden komplexe Aussagen, genau wie in der klassischen Aussagenlogik, aus atomaren Aussagen aufgebaut.

Die Bestandteile der Sprache der Dynamischen Aussagenlogik sind:

Symbole für atomare Programme:	$a, b, c, \dots$
Symbole für atomare Aussagen:	$p, q, r, \dots$
Propositionale Operatoren:	$\Rightarrow$ (Implikation), $\bot$ (Falsum)
Programmkonstr. Operatoren:	$;$ (Komposition), $\cup$ (Auswahl), * (Iteration)
Vermittelnde Operatoren:	$[\cdot]$ (Modalität der Notwendigkeit), $?$ (Test)
Hilfssymbole:	( )

Wir verwenden griechische Minuskeln ($\alpha, \beta, \gamma, \dots$) zur Bezeichnung von (beliebigen) Programmen und lateinische Majuskeln ($F, G, H, \dots$) zur Bezeichnung von (beliebigen) Aussagen.

Definition 3.2.1 (PDL-Ausdrücke) *Sei* Π_0 *die Menge der atomaren Programme und* Φ_0 *die Menge der atomaren Aussagen. Dann sind die Menge der* ***Programme*** Π *und die Menge der* ***Aussagen*** Φ *definiert als die kleinsten Mengen, für die gilt:*

1. $\Pi_0 \subseteq \Pi$
2. $\Phi_0 \subseteq \Phi, \bot \in \Phi$
3. *Wenn* $\alpha, \beta \in \Pi$*, dann auch* $(\alpha\, ; \beta) \in \Pi$*,* $(\alpha \cup \beta) \in \Pi$ *und* $\alpha^* \in \Pi$.
4. *Wenn* $F, G \in \Phi$*, dann auch* $(F \Rightarrow G) \in \Phi$.
5. *Wenn* $\alpha \in \Pi$ *und* $F \in \Phi$*, dann ist* $[\alpha]F \in \Phi$ *und* $F? \in \Pi$.

Um Klammern zu sparen, vereinbaren wir die folgenden Präzedenzen: unäre Operatoren binden stärker als binäre und ; bindet stärker als $\cup$. In diesem Zusammenhang werden auch die Modaloperatoren als unäre Operatoren aufgefasst.

Die intuitive Bedeutung der programmkonstruierenden und vermittelnden Operatoren ist:

$\alpha\,;\beta$	Führe zuerst α, danach β aus.
$\alpha \cup \beta$	Wähle nicht-deterministisch entweder α oder β, und führe das gewählte Programm aus.
α^*	Führe α wiederholt aus. Die Anzahl der Wiederholungen ist nicht-deterministisch und endlich, auch überhaupt keine Ausführung von α ist erlaubt.
$[\alpha]F$	Nach allen nicht-fehlschlagenden Ausführungen von α gilt F.
$F?$	Test auf Bedingung F; ist diese nicht erfüllt, schlägt das Programm fehl, ansonsten wird es fortgesetzt.

Da der Iterationsoperator nur endliche Wiederholungen erlaubt, ist PDL echt schwächer als zum Beispiel der modale μ-Kalkül [Koz83], der durch unbeschränkte Minimierung echte μ-Rekursion erlaubt. So lässt sich beispielsweise in PDL nicht ausdrücken, dass alle möglichen Abläufe eines nicht-deterministischen Programms terminieren [Str82]. In einigen Varianten wie LPDL, RPDL [HS82] oder ΔPDL [Str82] wurde versucht, diese Einschränkung zu umgehen. Für die in dieser Arbeit untersuchten Programme ist PDL allerdings ausreichend.

Die Sprache von PDL wird typischerweise um einige bekannte logische Konstrukte erweitert. Diese werden definiert durch

$$\begin{array}{rcl}
\neg F & := & F \Rightarrow \bot \\
\top & := & \neg\bot \\
F \vee G & := & \neg F \Rightarrow G \\
F \wedge G & := & \neg(F \Rightarrow \neg G) \\
F \Leftrightarrow G & := & (F \Rightarrow G) \wedge (G \Rightarrow F) \\
\langle\alpha\rangle F & := & \neg[\alpha]\neg F
\end{array}$$

Das letztgenannte Konstrukt in dieser Tabelle, $\langle\cdot\rangle$, wird auch Modalität der Möglichkeit genannt. $\langle\alpha\rangle F$ soll informell als “Es gibt einen terminierenden (nicht-fehlschlagenden) Programmlauf, nach dem F gilt” verstanden werden. Die restlichen Operatoren sind aus der klassischen Aussagenlogik bekannt und bezeichnen Negation ($\neg$), Wahrheit ($\top$), Disjunktion ($\vee$), Konjunktion ($\wedge$) und Äquivalenz ($\Leftrightarrow$). Für diese neuen Operatoren erweitern wir unsere Präzedenzregelung: $\wedge$ bindet stärker als $\vee$, welches stärker bindet als $\Rightarrow$, welches wiederum stärker bindet als $\Leftrightarrow$. Außerdem soll die Negation ($\neg$) stärker binden als die beiden PDL-Operatoren Iteration (*) und Test ($?$). In manchen Fällen werden wir auch den Programmkompositionsoperator einfach weglassen und somit einfach $\alpha\beta$ für die Hintereinanderausführung der beiden Programme α und β schreiben.

Um eine den üblichen Programmiersprachen angenäherte Notation zu erhalten, sind darüberhinaus weitere Abkürzungen verbreitet:

$$\begin{array}{rcl}
\textbf{skip} & := & \top? \\
\textbf{fail} & := & \bot? \\
\textbf{if } F \textbf{ then } \alpha \textbf{ else } \beta & := & (F?\,;\alpha) \cup (\neg F?\,;\beta) \\
\textbf{if } F \textbf{ then } \alpha & := & \textbf{if } F \textbf{ then } \alpha \textbf{ else skip} \\
\textbf{while } F \textbf{ do } \alpha & := & (F?\,;\alpha)^*\,;\neg F? \\
\textbf{repeat } \alpha \textbf{ until } F & := & \alpha\,; \textbf{while } \neg F \textbf{ do } \alpha \\
\textbf{for } i = 1 \textbf{ to } n \textbf{ do } \alpha(i) & := & \alpha(1)\,; \ldots\,; \alpha(n) \\
\textbf{if } F_1 \rightarrow \alpha_1 | \ldots | F_n \rightarrow \alpha_n \textbf{ fi} & := & F_1?\,;\alpha_1 \cup \cdots \cup F_n?\,;\alpha_n \\
\textbf{do } F_1 \rightarrow \alpha_1 | \ldots | F_n \rightarrow \alpha_n \textbf{ od} & := & (F_1?\,;\alpha_1 \cup \cdots \cup F_n?\,;\alpha_n)^*\,; \\
 & & \qquad (\neg F_1 \wedge \cdots \wedge \neg F_n)? \\
\{F\}\alpha\{G\} & := & F \Rightarrow [\alpha]G
\end{array}$$

Von den beiden Programmen **skip** und **fail** hat ersteres keine Änderung des Programmzustands zur Folge (NOP), letzteres ist ein Programm, das immer fehlschlägt. Die Konstrukte **if-then(-else)**, **while-do** und **for-to-do** sind bekannt aus den gängigen Programmiersprachen, wobei die **for**-Schleife ein über den natürlichen Zahlen (oder zumindest dem Abschnitt $\{1, \ldots, n\}$) parametrisiertes Programm $\alpha(i)$ benötigt. Die beiden abgeschirmten Anweisungen (*guarded commands*) **if**-|-**fi** und **do**-|-**od** gehen auf Dijkstra zurück [Dij76] und werden *alternative guarded command* und *iterative guarded command* genannt. Beim **if**-|-**fi**-Konstrukt wird nicht-deterministisch ein Kommando α_i, dessen abschirmende Formel F_i erfüllt ist, ausgewählt und ausgeführt. Im Falle von **do**-|-**od** wird dieser Prozess so lange wiederholt, bis keine der Formeln F_i mehr erfüllt sind. Das Konstrukt $\{F\}\alpha\{G\}$ ist Hoares partielle Korrektheitsaussage [Hoa69]. Diese besagt, dass falls Programm α in einem Zustand, der F erfüllt, gestartet wird und terminiert, es sich nach seiner Ausführung in einem Zustand befindet, in dem G gilt.

3.2.2 Semantik

Wir geben die Semantik für PDL-Formeln in der für Modallogiken gebräuchlichen Kripkesemantik an. Dabei wird eine PDL-Formel in einer Struktur $\mathcal{K} = (W, \tau_0, \sigma_0)$ interpretiert, der so genannten *Kripke-Struktur*. W ist dabei eine nicht näher bestimmte Zustandsmenge (Zustände werden auch *Welten* genannt),

$$\tau_0 : \Pi_0 \to \mathbb{P}(W \times W) \quad \text{und} \quad \sigma_0 : \Phi_0 \to \mathbb{P}(W)$$

definieren Zustandsübergangs- bzw. Bewertungsfunktionen für atomare Programme bzw. atomare Formeln. Die Funktionen τ_0 und σ_0 werden durch die folgenden Definitionen auf die kompletten Mengen Π und Φ erweitert:

Für Programme $\alpha \in \Pi$ wird $\tau(\alpha)$ induktiv definiert durch:

$$\begin{aligned}
\tau(\beta\,;\gamma) &= \tau(\beta) \circ \tau(\gamma) = \{(s,t) \mid (\exists u)((s,u) \in \tau(\beta) \wedge (u,t) \in \tau(\gamma))\} \\
\tau(\beta \cup \gamma) &= \tau(\beta) \cup \tau(\gamma) \\
\tau(\beta^*) &= \{(s,t) \mid (\exists k \in \mathbb{N})(\exists s_0 \ldots s_k) \\
&\qquad\qquad (s_0 = s \wedge s_k = t \wedge (s_i, s_{i+1}) \in \tau(\beta) \text{ für } 0 \leq i < k)\} \\
\tau(F?) &= \{(s,s) \mid s \in \sigma(F)\}
\end{aligned}$$

Für Formeln $F \in \Phi$ wird $\sigma(F)$ induktiv definiert durch:

$$\begin{aligned}
\sigma(\bot) &= \emptyset \\
\sigma(G \Rightarrow H) &= (W \setminus \sigma(G)) \cup \sigma(H) \\
\sigma([\alpha]G) &= \{s \in W \mid (\forall t)((s,t) \in \tau(\alpha) \Rightarrow t \in \sigma(G))\}
\end{aligned}$$

Die Definition der Bewertungsfunktion kann leicht auf die abgeleiteten Operatoren übertragen werden:

$$\begin{aligned}
\sigma(\top) &= W \\
\sigma(\neg G) &= W \setminus \sigma(G) \\
\sigma(G \vee H) &= \sigma(G) \cup \sigma(H) \\
\sigma(G \wedge H) &= \sigma(G) \cap \sigma(H) \\
\sigma(\langle\alpha\rangle G) &= \{s \in W \mid (\exists t)((s,t) \in \tau(\alpha) \wedge t \in \sigma(F))\}
\end{aligned}$$

Eine PDL-Formel $F \in \Phi$ ist wahr bezüglich einer Kripke-Struktur $\mathcal{K} = (W, \tau_0, \sigma_0)$ und eines Zustands $s \in W$, wenn $s \in \sigma(F)$. Man sagt dann auch "s erfüllt F in $\mathcal{K}$" und schreibt $(\mathcal{K}, s) \models F$. $\mathcal{K}$ ergibt sich meist aus dem Zusammenhang und wird daher oft weggelassen.

Wir definieren weiter die *Gültigkeit* einer Formel F in einer Kripke-Struktur $\mathcal{K} = (W_{\mathcal{K}}, \tau_0, \sigma_0)$, in Zeichen $\mathcal{K} \models F$, falls $(\mathcal{K}, s) \models F$ für alle $s \in W_{\mathcal{K}}$. Des weiteren heißt F *gültig*, geschrieben als $\models F$, falls es in jeder Kripke-Struktur gültig ist, d.h. falls $\mathcal{K} \models F$ für alle $\mathcal{K}$. F heißt *erfüllbar*, wenn es eine Kripke-Struktur $\mathcal{K}$ und einen Zustand $s \in W_{\mathcal{K}}$ gibt, so dass $(\mathcal{K}, s) \models F$. Falls $(\mathcal{K}, s) \models F$, so wird $(\mathcal{K}, s)$ *Modell* von F genannt. Analog heißt $\mathcal{K}$ Modell von F, falls $\mathcal{K} \models F$. Zwei Formeln F und G heißen äquivalent (i.Z. $F \equiv G$), falls $\models F \Leftrightarrow G$.

3.2.3 Axiomatisierung und elementare Eigenschaften

Für die Dynamische Aussagenlogik hat Segerberg [Seg77] ein Deduktionssystem vorgestellt, dessen Korrektheit und Vollständigkeit Gabbay [Gab77] und Parikh [Par78] unabhängig voneinander bewiesen haben. Dieses Deduktionssystem ist im Stil des Hilbertkalküls gehalten und in Abbildung 3.4 dargestellt. Wir werden dieses Deduktionssystem später als Grundlage für Beweise von PDL-Formeln verwenden.

Axiome:

(i) Alle Tautologien der Aussagenlogik
(ii) $[\alpha](F \Rightarrow G) \Rightarrow ([\alpha]F \Rightarrow [\alpha]G)$
(iii) $[\alpha](F \wedge G) \Leftrightarrow [\alpha]F \wedge [\alpha]G$
(iv) $[\alpha \cup \beta]F \Leftrightarrow [\alpha]F \wedge [\beta]F$
(v) $[\alpha\,;\beta]F \Leftrightarrow [\alpha][\beta]F$
(vi) $[F?]G \Leftrightarrow (F \Rightarrow G)$
(vii) $F \wedge [\alpha][\alpha^*]F \Leftrightarrow [\alpha^*]F$
(viii) $F \wedge [\alpha^*](F \Rightarrow [\alpha]F) \Rightarrow [\alpha^*]F$

Inferenzregeln:

(I) $\dfrac{F \quad F \Rightarrow G}{G}$ modus ponens

(II) $\dfrac{F}{[\alpha]F}$ modale Generalisierung

Abbildung 3.4: Deduktionssystem für PDL (zitiert nach Harel *et al.* [HKT00]).

Das nächste Lemma hält einige elementare Eigenschaften von PDL fest, die wir später noch brauchen werden.

Lemma 3.2.2 *Es gilt:*

1. $\langle\alpha\rangle(F \wedge G) \Rightarrow \langle\alpha\rangle F \wedge \langle\alpha\rangle G$ *und* $[\alpha]F \vee [\alpha]G \Rightarrow [\alpha](F \vee G)$.
2. $[\alpha^*]F \Leftrightarrow \bigwedge_{n\in\mathbb{N}}[\alpha^n]F$
3. $\langle F?\rangle G \Leftrightarrow (F \wedge G)$ *und* $\langle F?\rangle G \Rightarrow [F?]G$
4. *Für endliche Indexmengen* I *gilt:*
   $\langle\bigcup_{i\in I}\alpha_i\rangle F \Leftrightarrow \bigvee_{i\in I}\langle\alpha_i\rangle F$ *und* $[\bigcup_{i\in I}\alpha_i]F \Leftrightarrow \bigwedge_{i\in I}[\alpha_i]F$.

Beweis:

1. Diese beiden Aussagen sind die PDL-Analogien der beiden prädikatenlogischen Tautologien $(\exists x)(F(x) \wedge G(x)) \Rightarrow (\exists x)F(x) \wedge (\exists x)G(x)$ und $((\forall x)F(x) \vee (\forall x)G(x)) \Rightarrow (\forall x)(F(x) \vee G(x))$. Unter Verwendung der prädikatenlogischen PDL-Semantik folgen die behaupteten Implikationen direkt aus diesen.
2. Für den Beweis der Implikation von links nach rechts betrachten wir ein beliebiges $n \in \mathbb{N}$. Durch $n+1$-malige Anwendung von Axiom (vii) und intermittierendes "Ausmultiplizieren" mittels Axiom (iii) erhalten wir $[\alpha^*]F \Leftrightarrow F \wedge [\alpha]F \wedge \cdots \wedge [\alpha^n]F \wedge [\alpha^{n+1}][\alpha^*]F$, also gilt insbesondere $[\alpha^*]F \Rightarrow [\alpha^n]F$. Da dies für beliebiges n der Fall ist, folgt die Behauptung. Für die Rückrichtung der Implikation nehmen wir an, dass in einem Zustand s einer Kripke-Struktur für alle $n \in \mathbb{N}$ die Formel $[\alpha^n]F$ gilt. Die Formel $(\forall n)(\forall t)((s,t) \in \tau(\alpha^n) \Rightarrow t \in \sigma(F))$, die man durch Expansion der Semantik von $[\cdot]$ erhält, ist also gültig. Sie ist auch äquivalent zu $(\forall t)((\exists k)((s,t) \in \tau(\alpha^k)) \Rightarrow t \in \sigma(F))$, was der Semantik von $[\alpha^*]F$ entspricht.
3. Es gilt $\langle F?\rangle G \Leftrightarrow \neg([F?]\neg G) \Leftrightarrow \neg(F \Rightarrow \neg G) \Leftrightarrow (F \wedge G)$, sowie $\langle F?\rangle G \Leftrightarrow (F \wedge G) \Rightarrow (F \Rightarrow G) \Leftrightarrow [F?]G$.
4. Beweis per Induktion über die Mächtigkeit $|I|$ der Indexmenge unter Verwendung von Axiom (iv) liefert die Behauptung.

□

3.2.4 Reguläre Erfüllbarkeit

Verifikation eines in PDL spezifizierten Programms erfordert die Prüfung der Gültigkeit einer PDL-Formel. Diese enthält typischerweise neben dem Programm α selbst auch eine zu prüfende Eigenschaft E sowie, optional, weitere einschränkende Bedingungen. Die verbreitete Verifikation einer Nachbedingung E unter einer gegebenen Vorbedingung P entspricht z.B. der Gültigkeit von $P \Rightarrow [\alpha]E$. Dabei umfasst P neben der Vorbedingung auch die erwünschten Eigenschaften – die Spezifikation – der atomaren Programme, welche deren erlaubte Transitionen beschreibt. P ist demnach zusammengesetzt aus der Spezifikation der atomaren Programme, P_α, und der eigentlichen Vorbedingung P_V des Programms, d.h. $P = P_\alpha \wedge P_V$. Die Spezifikation P_α muss die Transitionsrelationen nicht unbedingt eindeutig beschreiben. Unwesentliche Implementationsdetails können in der Spezifikation fehlen, wodurch sich mehrere Interpretationsmöglichkeiten der atomaren Programme ergeben.

Betrachten wir als Beispiel die Fragestellung, ob nach jedem Ablauf des Programms $\alpha = a \cup (b^* \,; x?)$ immer x gilt, sofern nach Ablauf des (atomaren) Programms a immer x und nach Ablauf von b nie x gilt. Nach Gesagtem ergibt sich daraus also die Frage nach der Gültigkeit der PDL-Formel $[a]x \wedge [b]\neg x \Rightarrow [a \cup (b^* \,; x?)]x$. P_α ist in diesem Beispiel demnach die Teilformel $[a]x \wedge [b]\neg x$. (Die Eigenschaft $[b]\neg x$ wird zum Beweis nicht benötigt.)

Zur Lösung der für die Verifikation erforderlichen Beweisaufgabe sind verschiedene Verfahren denkbar:

1. Man kann versuchen, die Aussage $\models P \Rightarrow [\alpha]E$ direkt zu beweisen. Dazu kann das Axiomensystem aus Abbildung 3.4 herangezogen werden, welches allerdings für automatische Beweiszwecke weniger geeignet ist, da das Kalkül wegen der Modus-Ponens-Inferenzregel nicht invertierbar ist. Eine weitere Möglichkeit besteht in der Generierung einer approximierenden Modellsequenz basierend auf der Filtrierung der universellen Nonstandard-Kripke-Struktur, wie von Pratt vorgeschlagen [Pra79]. Dieses Verfahren wird häufig auch in der Form eines Tableau-Beweisverfahrens dargestellt. Alternativ bietet sich der Einsatz automaten-theoretischer Methoden an, wie sie z.B. zum direkten Beweis der Entscheidbarkeit des μ-Kalküls entwickelt wurden [SE89] und heute in theoretischen Untersuchungen eine weite Verbreitung gefunden haben.

2. Man kann die Menge aller Modelle (Kripke-Strukturen) M betrach-

ten, welche die Spezifikation P_α der atomaren Programme erfüllen, d.h. solche M, für die $M \models P_\alpha$ gilt, und dann durch Model-Checking [CGP00] überprüfen, ob $M \models P_V \Rightarrow [\alpha]E$ gilt. Die Modelle M müssen dabei wegen der *small model* Eigenschaft (siehe Theorem 3.2.6 auf Seite 89) nur bis zur Größe $2^{|F|}$ betrachtet werden, wobei $|F|$ die Anzahl der in F vorkommenden Symbole bezeichnet. Wenn P_α nur ein Modell besitzt oder es ein allgemeinstes Modell gibt, dessen Eigenschaften in allen anderen Modellen ebenfalls gelten (d.h. es gibt ein M_A, so dass für alle Modelle M von P_α und alle Formeln F aus $M_A \models F$ folgt, dass $M \models F$), hat man das Verifikationsproblem auf ein einziges Model-Checking-Problem reduziert. Die weiter unten (auf Seite 93) definierte Kripke-Struktur $\mathcal{K}^{\mathcal{S}}_{\Box}$ zur Verifikation Boolescher Zuweisungsprogramme ist ein Beispiel einer solchen allgemeinsten Struktur.

3. Man kann in manchen Fällen das gesamte Programm α in ein äquivalentes Modell, nennen wir es M_α, (einen Automaten) übersetzen, in dem insbesondere die atomaren Programme die gewünschten Eigenschaften haben, also $M_\alpha \models P_\alpha$ gilt. Dann kann man per Model-Checking prüfen, ob für alle Zustände s dieses Modells aus $(M_\alpha, s) \models P_V$ die Eigenschaft $(M_\alpha, s) \models [\alpha]E$ folgt (z.B. durch eine Erreichbarkeitsanalyse). Eine solche Übersetzung ist beispielsweise in der Hardwareverifikation üblich. Bei der weiter unten betrachteten Verifikation Boolescher Zuweisungsprogramme ist eine solche Übersetzung ebenfalls denkbar.

Die zuerst beschriebene Variante, das auf Pratt zurückgehende Tableau-Beweisverfahren, hat den Nachteil, dass es ein Modell generiert, in dem jeder Zustand (annähernd) einer Teilmenge der Subformeln (genauer einer Teilmenge des Fischer-Ladner-Abschlusses $\mathrm{FL}(F)$ [FL77]) der zu prüfenden Formel F entspricht. Das so generierte Modell hat im schlimmsten Fall also eine exponentielle Anzahl von Zuständen in der Formelgröße. Für die von uns betrachteten Probleme, die Formeln mit mehr als 100.000 Symbolen enthalten, stößt ein solches Verfahren schnell an seine Grenzen. Der in dem automaten-theoretischen Verfahren von Streett und Emerson generierte Automat bearbeitet als Eingabe einen Baum, dessen Knoten ebenfalls Teilmengen von $\mathrm{FL}(F)$ sind, also wiederum auf einer entsprechend großen Struktur. Auf konkrete Implementation ausgerichtete Veröffentlichungen sind bei diesen Verfahren spärlich gesät (eine Ausnahme ist z.B. [DGM96]).

In der als zweites beschriebenen Variante kann die Anzahl der Modelle von P_α die limitierende Größe sein. Außerdem sind die meisten Model-Checking-Verfahren auf große Modelle aber kleine zu prüfende Formeln spezialisiert: Ihre Laufzeit ist linear in der Größe des Modells, aber exponentiell in der Größe der Formel. Die bei uns zu prüfende Formel $P_V \Rightarrow [\alpha]E$ enthält aber das gesamte zu verifizierende Programm α, und ist daher recht groß.

Problem der dritten Variante ist es, herauszufinden, ob es einen einfachen, dem Programm äquivalenten Automaten gibt und falls ja, wie dieser aussieht. In den oben erwähnten Fällen mag es noch möglich sein, einen solchen direkt aus dem Programm abzulesen, im allgemeinen Fall beliebiger Programme P_α ist dem Autor allerdings kein solches Verfahren bekannt.

Wir schlagen hier einen neuen Weg vor, der methodisch zwischen der zweiten und dritten Variante liegt und ebenso wie diese die Beweisaufgabe auf eine oder mehrere Model-Checking-Aufgaben zurückführt. Als Beitrag zur zweiten Variante liefert unser Verfahren (in manchen Fällen) eine Methode zur Generierung des oben skizzierten allgemeinsten Modells M_A. Zur dritten Variante kann unser Verfahren ein Kriterium beisteuern, wann die Transformation des Programms in einen einfachen Automaten möglich ist.

Unser Verfahren lässt sich nicht auf beliebige Verifikationsaufgaben anwenden, sondern nur auf solche, bei denen die atomaren Programme bestimmten Randbedingungen genügen (dadurch ergibt sich auch das erwähnte Kriterium für die dritte Variante). Diese zusätzlichen Bedingungen an die atomaren Programme formulieren wir nicht direkt als Bedingungen an P_α (aus Gründen, die wir weiter unten, auf Seite 84, näher erläutern werden), sondern als Einschränkung der erlaubten Modelle, der Kripke-Strukturen. Wir definieren dazu im folgenden den Begriff der “regulären Kripke-Struktur” und damit zusammenhängend den der “regulären Erfüllbarkeit” (der Erfüllbarkeit in einer regulären Kripke-Struktur). Reguläre Kripke-Strukturen sind ein Spezialfall der allgemeinen Kripke-Strukturen, anhand derer die Semantik von PDL gewöhnlich definiert wird. Durch die Einschränkung auf reguläre Strukturen erhalten wir eine semantisch eingeschränkte Variante von PDL, die sich für viele automatischen Beweisaufgaben als noch ausreichend erweist, so auch für das Beweisen von Eigenschaften Boolescher Zuweisungsprogramme, wie sie zur Verifikation von regelbasierten Expertensystemen oder Konfiguratoren benötigt werden.[7]

Unsere Vorgehensweise ist grob skizziert die folgende: Wir zeigen (in Ko-

---

[7] In der Literatur wurden unter den Begriffen Caucal-Basis und Selbst-Bisimulation der Regularität verwandte Konzepte beschrieben [Cau90].

rollar 3.2.7), dass eine (bezüglich einer Menge von Aussagevariablen Φ_0) regulär erfüllbare Formel F in einem “kleinen” Modell mit höchstens $2^{|\Phi_0|}$ Zuständen erfüllbar ist. Satz 3.2.9 liefert darüberhinaus die Erfüllbarkeit in einem universellen, nur von den Transitionsrelationen der atomaren Programme abhängigen “Standardmodell” S mit $2^{|\Phi_0|}$ Zuständen. Sind die Transitionsrelationen durch die Eigenschaften P_α eindeutig festgelegt, so ergibt sich genau ein Standardmodell. Somit kann in diesem Fall das Gültigkeitsproblem (in allen regulären Strukturen) auf ein Model-Checking-Problem in S reduziert werden. Im folgenden betrachten wir dann den Spezialfall Boolescher Zuweisungsprogramme. In Lemma 3.2.12 zeigen wir, dass (gewöhnliche) Erfüllbarkeit mit regulärer Erfüllbarkeit übereinstimmt, falls alle atomaren Programme Boolesche Zuweisungsprogramme sind. Daraus ergibt sich für die Gültigkeitsprüfung einer PDL-Formel, die nur Boolesche Zuweisungsprogramme als atomare Programme enthält, die Möglichkeit, diese Eigenschaft in einem speziellen Modell (bezeichnet mit $\mathcal{K}_\Box^\mathcal{S}$, siehe S. 93) anhand von Model-Checking nachzuweisen.

Wir beginnen die detaillierte Beschreibung unseres Verfahrens mit einer einfachen Feststellung: Jedem Zustand $s \in W_\mathcal{K}$ einer Kripke-Struktur $\mathcal{K} = (W_\mathcal{K}, \tau_0, \sigma_0)$ ist eine eindeutige Belegung $\beta_s : \Phi_0 \to \mathbb{B}$ der Aussagevariablen zugeordnet, die sich aus σ_0 ergibt:

$$\beta_s(p) = \begin{cases} \textbf{true} & \text{falls } s \in \sigma_0(p), \\ \textbf{false} & \text{sonst.} \end{cases}$$

Zustände s_1, s_2 gleicher Variablenbelegung, d.h. mit $\beta_{s_1} \equiv \beta_{s_2}$, bezeichnen wir als *$\beta$-gleich* (in Zeichen: $s_1 =_\beta s_2$). Diese Äquivalenzrelation induziert auf $\mathcal{K}$ eine Quotientenalgebra, die wir mit $\mathcal{K}/_{=_\beta}$ bezeichnen. Wir verwenden für die Quotientenstruktur die folgenden Bezeichnungen:

$$\begin{aligned} [s]_{=_\beta} = [s] &= \{t \mid t =_\beta s\} \qquad \text{(Äquivalenzklasse von } s) \\ W/_{=_\beta} &= \{[s] \mid s \in W\} \\ \sigma_0/_{=_\beta}(p) &= \{[s] \mid s \in \sigma_0(p)\} \\ \tau_0/_{=_\beta}(a) &= \{([s],[t]) \mid (s,t) \in \tau_0(a)\} \end{aligned}$$

Damit ist $\mathcal{K}/_{=_\beta} = (W/_{=_\beta}, \sigma_0/_{=_\beta}, \tau_0/_{=_\beta})$. Die Funktionen $\sigma_0/_{=_\beta}$ und $\tau_0/_{=_\beta}$ lassen sich wiederum durch oben angegebene induktive Definition zu $\sigma/_{=_\beta}$ und $\tau/_{=_\beta}$ auf ganz Φ bzw. Π erweitern.

Definition 3.2.3 *Eine Kripke-Struktur* $\mathcal{K} = (W, \sigma_0, \tau_0)$ *heißt* Φ_0-***regulär,*** *falls es für alle atomaren Programme* $a \in \Pi_0$ *und alle Zustände* $s, s', t \in W$

mit $(s,t) \in \tau_0(a)$ *und* $s' =_\beta s$ *ein* $t' \in W$ *gibt, so dass* $(s',t') \in \tau_0(a)$ *und* $t' =_\beta t$.

In einer regulären Kripke-Struktur sind also die möglichen Zustandsübergänge aller atomaren Programme bereits durch die Belegung der Aussagevariablen festgelegt; oder anders herum: zwei Zustände, die sich in den möglichen atomaren Transitionen unterscheiden, können auch schon anhand der Belegung der Aussagevariablen unterschieden werden.

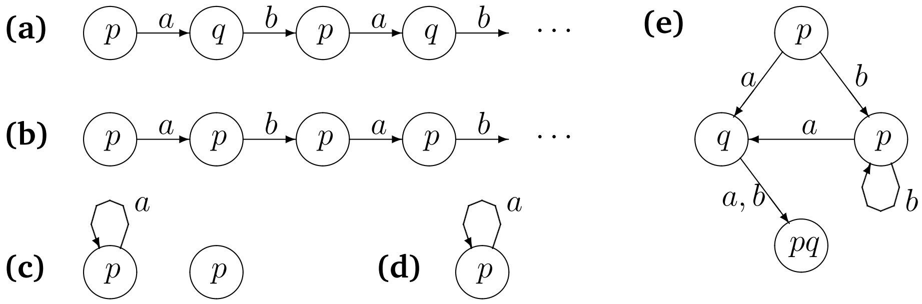


Abbildung 3.5: Reguläre und irreguläre Kripke-Strukturen: (a),(d),(e) regulär; (b),(c) irregulär.

Wir wollen den Begriff der Regularität anhand einiger Beispiele erläutern. Nehmen wir an, dass $\Phi_0 = \{p,q\}$. Dann sind die Strukturen (a), (d) und (e) aus Abbildung 3.5 Φ_0-regulär, (b) und (c) hingegen nicht. In der graphischen Notation der Abbildung haben wir dabei in die Kreise, welche die Zustände repräsentieren, genau jene Aussagevariablen geschrieben, die in diesem Zustand wahr sind.

Die Regularitätsbedingung kann auch in klassischer Prädikatenlogik ausgedrückt werden, dann als Gültigkeit der Formel

$$\forall xyz \left(R_a xy \wedge \bigwedge_{P \in \Phi_0} (Px \Leftrightarrow Pz) \;\Rightarrow\; \exists u \Big(R_a zu \wedge \bigwedge_{P \in \Phi_0} (Py \Leftrightarrow Pu) \Big) \right) \tag{3.3}$$

für alle $a \in \Pi_0$, wobei R_a die Sichtbarkeits- oder Transitionsrelation des Programms a bezeichnet und P die prädikative Bewertungsfunktion. Diese Bedingung lässt sich nicht in PDL ausdrücken, wie man durch ein Standardargument der Korrespondenztheorie leicht belegen kann: Segerbergs

Generierungstheorem[8] ([Seg71], siehe auch Theorem 2.1.3 in [vB84]) ist verletzt, da Bedingung 3.3 in der in Abbildung 3.5 (c) dargestellten Kripke-Struktur nicht gilt, jedoch in dem von dieser generierten, in (d) dargestellten Untermodell erfüllt ist. Indem man nur reguläre Kripke-Strukturen betrachtet, erhält man also eine wirkliche semantische, nicht syntaktisch formulierbare Einschränkung.

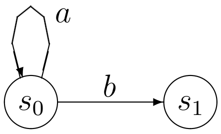
Man kann in der Einschränkung auf reguläre Strukturen gewisse Ähnlichkeiten zu hybriden Logiken [Pri67] sehen. Dies sind Modallogiken, in denen Zustände durch besondere, eindeutige Bezeichner identifiziert und unterschieden werden können. In Formeln dürfen diese Bezeichner direkt verwendet werden, um auf konkrete Zustände Bezug zu nehmen. In der zu einer regulären Struktur $\mathcal{K}$ gehörigen Quotientenstruktur $\mathcal{K}/_{=_\beta}$ kann ein Zustand durch seine Variablenbelegung eindeutig identifiziert werden.

Definition 3.2.4 *Eine Formel $F \in \Phi$ heißt* ***regulär erfüllbar bezüglich*** Φ_0*, wenn sie in einer Φ_0-regulären Kripke-Struktur erfüllbar ist.*

Wiederum wollen wir diesen Begriff zuerst an einem Beispiel erläutern. Sei $\Phi_0 = \emptyset$, $\Pi_0 = \{a, b\}$ und $F = \langle a\rangle\top \wedge \langle b\rangle([a \cup b]\bot)$. Dann ist F erfüllbar, da für die Kripke-Struktur $\mathcal{K} = (W, \sigma_0, \tau_0)$ mit

$$W = \{s_0, s_1\}, \quad \tau_0(a) = \{(s_0, s_0)\} \quad \text{und} \quad \tau_0(b) = \{(s_0, s_1)\},$$

$(\mathcal{K}, s_0) \models F$ gilt. Die Übergangsrelation von $\mathcal{K}$ sieht graphisch dargestellt so aus:



$\mathcal{K}$ ist nicht regulär, da $s_0 =_\beta s_1$, $(s_0, s_0) \in \tau_0(a)$, aber $(s_1, t) \notin \tau_0(a)$ für alle $t \in W$. Man überprüft leicht, dass F in keiner regulären Kripke-Struktur erfüllbar, also auch nicht regulär erfüllbar ist. Erlaubt man jedoch durch hinzufügen einer Aussagevariable eine Unterscheidung von s_0 und s_1, so wird $\mathcal{K}$ dadurch regulär und F regulär erfüllbar. Sei nun also $\Phi_0 = \{p\}$, Π_0

---

[8] Dieses Theorem besagt, dass für alle generierten Untermodelle $\mathcal{K}' = (W', \tau_0', \sigma_0')$ einer Kripke-Struktur $\mathcal{K} = (W, \tau_0, \sigma_0)$, alle $s' \in W'$ und alle Formeln $F \in \Phi$ die Relation $(\mathcal{K}', s') \models F$ genau dann gilt, wenn $(\mathcal{K}, s') \models F$.

und F wie oben. Dann ist $\mathcal{K}' = (W', \sigma_0', \tau_0')$ mit

$$W' = \{s_0, s_1\} \qquad \sigma_0'(p) = \{s_0\} \qquad \begin{array}{l} \tau_0'(a) = \{(s_0, s_0)\} \\ \tau_0'(b) = \{(s_0, s_1)\} \end{array}$$

regulär, und F ist regulär erfüllbar, denn es gilt wie in $\mathcal{K}$ auch $(\mathcal{K}', s_0) \models F$.

Indem man genug Variablen zur Menge Φ_0 hinzufügt, lässt sich jede Formel regulär erfüllbar bezüglich Φ_0 machen. Unter Verwendung von Fischer und Ladners Abschluss $\mathrm{FL}(F)$ einer Formel F [FL77] und der zugehörigen kollabierten Modell-Konstruktion kann man zeigen, dass eine erfüllbare Formel F immer Φ_0-regulär erfüllbar ist, sofern $|\Phi_0| \geq |\mathrm{FL}(F)|$. Unsere Zielsetzung ist es aber nicht, durch Hinzufügen zusätzlicher Variablen eine Formel erfüllbar zu machen, sondern Bedingungen zu entwickeln, unter denen die gegebenen Variablen ausreichend sind.

Der nächste Satz zeigt, dass reguläre Erfüllbarkeit unter Bildung obiger Quotientenstruktur erhalten bleibt.

Satz 3.2.5 *Sei* $\mathcal{K} = (W, \sigma_0, \tau_0)$ *eine reguläre Kripke-Struktur. Dann ist auch* $\mathcal{K}/_{=_\beta} = (W/_{=_\beta}, \sigma_0/_{=_\beta}, \tau_0/_{=_\beta})$ *eine reguläre Kripke-Struktur, und für alle* $F \in \Phi$ *gilt* $(\mathcal{K}, s) \models F$ *genau dann, wenn* $(\mathcal{K}/_{=_\beta}, [s]_{=_\beta}) \models F$.

Beweis: Da $[s] =_\beta [s']$ genau dann, wenn $[s] = [s']$, ist mit $([s], [t]) \in \tau_0/_{=_\beta}$ auch $([s'], [t]) \in \tau_0/_{=_\beta}$ und $\mathcal{K}/_{=_\beta}$ somit regulär. Bleibt zu zeigen, dass $(\mathcal{K}, s) \models F \Leftrightarrow (\mathcal{K}/_{=_\beta}, [s]) \models F$. Wir zeigen dies, indem wir mittels simultaner struktureller Induktion über $[\alpha]F$ die beiden folgenden Aussagen beweisen[9]:

(i) $s \in \sigma(F) \Leftrightarrow [s] \in \sigma/_{=_\beta}(F)$ für alle $F \in \Phi$.

(ii) Für alle $[\alpha]F \in \Phi$ gilt:
  (a) $(s, t) \in \tau(\alpha) \Rightarrow ([s], [t]) \in \tau/_{=_\beta}(\alpha)$ und
  (b) $([s], [t]) \in \tau/_{=_\beta}(\alpha) \wedge s \in \sigma([\alpha]F) \Rightarrow t \in \sigma(F)$.

Aus (i) folgt dann direkt die Behauptung des Lemmas. Zum Beweis von (i) unterscheiden wir nach dem Aufbau von F:

$F = p \in \Phi_0$**:** Ist $s \in \sigma_0(p)$, so ist nach Def. von $\sigma_0/_{=_\beta}$ auch $[s] \in \sigma_0/_{=_\beta}(p)$. Gilt umgekehrt $[s] \in \sigma_0/_{=_\beta}(p)$, so gibt es ein s' mit $s' =_\beta s$ und $s' \in \sigma_0(p)$. Da $s' \in \sigma_0(p) \Leftrightarrow s \in \sigma_0(p)$, ist auch $s \in \sigma_0(p)$.

---

[9] Genauer verwenden wir zum Induktionsbeweis die lexikographische Ordnung zuerst nach $|\alpha|$ und dann nach $|F|$.

$F = \bot$: Nach Def. ist sowohl $\sigma(\bot) = \emptyset$ als auch $\sigma/_{=_\beta}(\bot) = \emptyset$.

$F = G \Rightarrow H$: Nach Induktionsvoraussetzung gilt (i) sowohl für G als auch für H, also:

$$\begin{aligned} s \in \sigma(G \Rightarrow H) &\Leftrightarrow s \in \sigma(G) \Rightarrow s \in \sigma(H) \\ &\Leftrightarrow s \in \sigma/_{=_\beta}(G) \Rightarrow s \in \sigma/_{=_\beta}(H) \quad \text{nach I.V.} \\ &\Leftrightarrow s \in \sigma/_{=_\beta}(G \Rightarrow H)\ . \end{aligned}$$

$F = [\gamma]G$: Nach Induktionsvoraussetzung gilt (i) für G und (ii) für $[\gamma]G$ (da $[\gamma]G$ bezüglich der Induktionsordnung kleiner als $[\gamma]F = [\gamma]([\gamma]G)$ ist) . Wenn also $s \in \sigma([\gamma]G)$, so gilt $(\forall t)(([s],[t]) \in \tau/_{=\beta}(\gamma) \Rightarrow t \in \sigma(G))$ nach (ii)(b). Umgekehrt folgt nach (ii)(a) aus $(\forall t)(([s],[t]) \in \tau/_{=\beta}(\gamma) \Rightarrow t \in \sigma(G))$, dass $(\forall t)((s,t) \in \tau/_{=\beta}(\gamma) \Rightarrow t \in \sigma(G))$, daher $s \in \sigma([\gamma]G)$ nach Def. von σ. Insgesamt erhalten wir:

$$\begin{aligned} s \in \sigma([\gamma]G) &\Leftrightarrow (\forall t)(([s],[t]) \in \tau/_{=_\beta}(\gamma) \Rightarrow t \in \sigma(G)) \\ &\Leftrightarrow (\forall t)(([s],[t]) \in \tau/_{=_\beta}(\gamma) \Rightarrow [t] \in \sigma/_{=_\beta}(G)) \quad \text{nach (i)} \\ &\Leftrightarrow [s] \in \sigma/_{=_\beta}([\gamma]G) \quad \text{nach Def. von } \sigma/_{=_\beta}. \end{aligned}$$

Nun beweisen wir (ii), wobei wir fünf Fälle, je nach Aufbau von α, unterscheiden:

$\alpha = a \in \Pi_0$: Für (a) ist $(s,t) \in \tau_0(a) \Rightarrow ([s],[t]) \in \tau_0/_{=_\beta}(a)$ nachzuweisen, was auf Grund der Definition von $\tau_0/_{=_\beta}$ gilt. Für (b) argumentieren wir so: Wenn $([s],[t]) \in \tau_0/_{=_\beta}(a)$, so existieren $s' =_\beta s$ und $t' =_\beta t$ mit $(s',t') \in \tau_0(a)$. Da $\mathcal{K}$ regulär ist, gibt es ein t^*, für das $(s,t^*) \in \tau_0(a)$ und $t^* =_\beta t'$. Ist nun $s \in \sigma([a]F)$, so gilt nach Definition von σ insbesondere $t^* \in \sigma(F)$. Da $t =_\beta t^*$, ist $[t] = [t^*]$ und durch zweimalige Anwendung von I.V. (i) für F erhalten wir $t \in \sigma(F)$.

$\alpha = G?$: Eigenschaft (a) folgt aus Induktionsvoraussetzung (i) für G und der Definition von σ; (b) gilt auf Grund von

$$\begin{aligned} &([s],[s]) \in \tau/_{=_\beta}(G?) \wedge s \in \sigma([G?]F) \\ &\quad \Rightarrow [s] \in \sigma/_{=_\beta}(G) \wedge s \in \sigma(G \Rightarrow F) \quad \text{nach Axiom (vi)} \\ &\quad \Rightarrow s \in \sigma(G) \wedge s \in \sigma(G \Rightarrow F) \quad \text{nach I.V. (i) für } G \\ &\quad \Rightarrow s \in \sigma(F)\ . \end{aligned}$$

$\alpha = \gamma \cup \delta$**:** Aus der Induktionsvoraussetzung (ii)(a) für γ und δ und der Def. von τ folgt sofort (a). Außerdem gilt

$$\begin{aligned}
&([s],[t]) \in \tau/_{=_\beta}(\gamma \cup \delta) \wedge s \in \sigma([\gamma \cup \delta]F) \\
&\quad \Rightarrow \big(([s],[t]) \in \tau/_{=_\beta}(\gamma) \vee ([s],[t]) \in \tau/_{=_\beta}(\delta)\big) \wedge \\
&\qquad\qquad s \in \sigma([\gamma]F) \wedge s \in \sigma([\delta]F) \quad \text{nach Axiom (iv)} \\
&\quad \Rightarrow t \in \sigma(F) \quad \text{nach I.V. für } [\gamma]F \text{ und } [\delta]F,
\end{aligned}$$

und somit auch (b).

$\alpha = \gamma \,;\delta$**:** Behauptung (a) gilt wegen

$$\begin{aligned}
(s,t) \in \tau(\gamma\,;\delta) &\Rightarrow (\exists u)((s,u) \in \tau(\gamma) \wedge (u,t) \in \tau(\delta)) \\
&\Rightarrow (\exists u)(([s],[u]) \in \tau(\gamma) \wedge ([u],[t]) \in \tau(\delta)) \\
&\qquad\qquad \text{nach I.V. (ii)(a)} \\
&\Rightarrow ([s],[t]) \in \tau(\gamma\,;\delta)\ ,
\end{aligned}$$

(b) ergibt sich aus

$$\begin{aligned}
&([s],[t]) \in \tau/_{=_\beta}(\gamma\,;\delta) \wedge s \in \sigma([\gamma\,;\delta]F) \\
&\quad \Rightarrow (\exists u)(([s],[u]) \in \tau/_{=_\beta}(\gamma) \wedge ([u],[t]) \in \tau/_{=_\beta}(\delta)) \wedge \\
&\qquad\qquad s \in \sigma([\gamma][\delta]F) \quad \text{nach Axiom (v)} \\
&\quad \Rightarrow (\exists u)(([u],[t]) \in \tau/_{=_\beta}(\delta) \wedge u \in \sigma([\delta]F)) \quad \text{nach I.V. (ii)(b)} \\
&\quad \Rightarrow t \in \sigma(F) \quad \text{nach I.V. (ii)(b).}
\end{aligned}$$

$\alpha = \gamma^*$**:** Zuerst zeigen wir (a). Falls $(s,t) \in \tau(\gamma^*)$, so existieren $t_0, \ldots, t_n$ mit $s = t_0$, $(t_i, t_{i+1}) \in \tau(\gamma)$ für $0 \leq i < n$ und $t_n = t$. Nach Induktionsvoraussetzung (ii)(a) ist dann $([t_i],[t_{i+1}]) \in \tau/_{=_\beta}(\gamma)$ für $0 \leq i < n$, also $([s],[t]) = ([t_0],[t_n]) \in \tau/_{=_\beta}(\gamma^*)$. Nehmen wir nun für den Beweis von (b) an, dass die Voraussetzung $(s,t) \in \tau/_{=_\beta}(\gamma^*) \wedge s \in \sigma([\gamma^*]F)$ gilt. Dann existieren $t_0, \ldots, t_n$ mit $t_0 = s$, $t_n = t$ und $([t_i],[t_{i+1}]) \in \tau/_{=_\beta}(\gamma)$. Wir zeigen nun per Induktion, dass $t_i \in \sigma([\gamma^*]F)$ für $0 \leq i \leq n$. Nach Voraussetzung gilt die Behauptung für $t_0 = s$. Nehmen wir also an, sie gelte für ein $i < n$. Dann ist $t_i \in \sigma([\gamma^*]F)$, und wegen Axiom (vii) auch $t_i \in \sigma([\gamma][\gamma^*]F)$. Anwendung der Induktionsvoraussetzung (ii)(b) für $[\gamma][\gamma^*]F$ liefert $t_{i+1} \in \sigma([\gamma^*]F)$. Dies beschließt unsere innere Induktion, aus der wir $t = t_n \in \sigma([\gamma^*]F)$ erhalten. Daraus ergibt sich nach Def. von σ oder mittels Axiom (vii) sofort $t \in \sigma(F)$.

□

Der Beweis von Satz 3.2.5 ist angelehnt an den Beweis des Filtrierungs-Lemmas von Fischer und Ladner in der Version von Harel *et al.* [HKT00]. Fischer und Ladner können mit ihrem Filtrierungs-Lemma das “Theorem der kleinen Modelle” beweisen, nach dem eine erfüllbare PDL-Formel immer schon in einer “kleinen” Kripke-Struktur erfüllbar ist:

Theorem 3.2.6 *(Small Model Theorem [FL77]) Sei* F *eine erfüllbare PDL-Formel. Dann gibt es eine Kripke-Struktur* $\mathcal{K} = (W_\mathcal{K}, \tau_0, \sigma_0)$ *und einen Zustand* $s \in W_\mathcal{K}$*, so dass* $(\mathcal{K}, s) \models F$ *und* $|\mathcal{K}| \leq 2^{|F|}$.

Dabei ist $|\mathcal{K}|$ die Mächtigkeit der Trägermenge, d.h. $|\mathcal{K}| = |W_\mathcal{K}|$, $|F|$ bezeichnet die Anzahl der in F vorkommenden Symbole.

Satz 3.2.5 erlaubt die Formulierung eines analogen Resultats für reguläre Kripke-Strukturen, wobei durch die Einschränkung auf Φ_0-reguläre Strukturen eine Verschärfung des Ergebnisses von Fischer und Ladner möglich wird: Während deren Schranke von der Größe der Formel F abhängt, ist unsere Schranke unabhängig von F. Dadurch kann auch die Größe des “kleinen” Modells weiter reduziert werden. Dieses ist dann nicht mehr exponentiell in $|F|$, sondern nur noch exponentiell in der Anzahl $|\Phi_0|$ der Aussagevariablen.

Korollar 3.2.7 (zu Satz 3.2.5) *Sei* $F \in \Phi$ *eine regulär erfüllbare Formel über den Aussagevariablen* Φ_0*. Dann gibt es eine Kripke-Struktur* $\mathcal{K} = (W_\mathcal{K}, \tau_0, \sigma_0)$ *und einen Zustand* $s \in W_\mathcal{K}$*, so dass* $(\mathcal{K}, s) \models F$ *und* $|\mathcal{K}| \leq 2^{|\Phi_0|}$.

Beweis: Nach Satz 3.2.5 gibt es eine reguläre Quotientenstruktur $\mathcal{K}/_{=_\beta} = (W/_{=_\beta}, \sigma_0/_{=_\beta}, \tau_0/_{=_\beta})$, in der F erfüllbar ist. Der Index $|W/_{=_\beta}|$ von $=_\beta$ ist aber $\leq 2^{|\Phi_0|}$, da es höchstens $2^{|\Phi_0|} = |\Phi_0 \to \mathbb{B}|$ verschiedene Belegungsfunktionen gibt und zwei Zustände $s_{1/2}$ genau dann gleich sind, wenn deren Belegungsfunktionen $\beta_{s_{1/2}} : \Phi_0 \to \mathbb{B}$ gleich sind. □

Analog zu Fischer und Ladners Vorgehen ließe sich auch aus diesem Ergebnis ein Algorithmus zur Erfüllbarkeitsprüfung von PDL-Formeln ableiten, der ebenso wie deren Algorithmus in seiner trivialen Form eine deterministisch doppelt exponentielle Laufzeit, diesmal in der Anzahl der Aussagevariablen besitzt. In den Grundzügen sieht der Algorithmus wie folgt aus: Generiere alle Kripke-Strukturen $\mathcal{K}$ bis zur Größe $2^{|\Phi_0|}$ und prüfe in jeder dieser Strukturen für alle Zustände $s \in W_\mathcal{K}$, ob die zu prüfende Formel F erfüllt ist, also $(\mathcal{K}, s) \models F$. Dieser Test ist in polynomieller Zeit in $|\mathcal{K}|$ durchzuführen [FL77], der gesamte Algorithmus also doppelt exponentiell.

Unser Interesse geht allerdings in eine andere Richtung. Wir wollen für eine fest vorgegebene Semantik der atomaren Programme eine einfache Prüfmöglichkeit der Erfüllbarkeit erreichen. Dazu definieren wir:

Definition 3.2.8 *Sei* $W^{\mathcal{S}} = \Phi_0 \to \mathbb{B}$ *und* $\sigma_0^{\mathcal{S}} : \Phi_0 \to \mathbb{P}(W^{\mathcal{S}})$ *mit* $\sigma_0^{\mathcal{S}}(p) = \{s \mid s(p) = \mathbf{true}\}$ *für alle* $p \in \Phi_0$*. Für eine Semantik* $\tau_0 : \Pi_0 \to \mathbb{P}(W^{\mathcal{S}} \times W^{\mathcal{S}})$ *der atomaren Programme heißt die Kripke-Struktur* $\mathcal{K}^{\mathcal{S}} = (W^{\mathcal{S}}, \sigma_0^{\mathcal{S}}, \tau_0)$ *die* Φ_0***-reguläre Standardstruktur von*** τ_0*.*

Die Zustände einer regulären Standardstruktur sind also Bewertungsfunktionen. Durch diese Konstruktion kann jeder Zustand eindeutig anhand der in ihm gültigen Prädikate identifiziert werden. Man überprüft leicht, dass $\mathcal{K}^{\mathcal{S}}$ immer eine Φ_0-reguläre Kripke-Struktur ist.

Satz 3.2.9 *Sei* $F \in \Phi$ *regulär erfüllbar bezüglich* Φ_0*. Dann gibt es ein* τ_0*, so dass* F *in der* Φ_0*-regulären Standardstruktur von* τ_0 *erfüllbar ist.*

Beweis: Da F regulär erfüllbar bezüglich Φ_0 ist, gibt es eine Φ_0-reguläre Kripke-Struktur $\mathcal{K} = (W, \tau_0, \sigma_0)$ und ein $s \in W$, so dass $(\mathcal{K}, s) \models F$. Nach Satz 3.2.5 gilt $(\mathcal{K}/_{=_\beta}, [s]) \models F$ dann auch in der Quotientenstruktur $\mathcal{K}/_{=_\beta} = (W/_{=_\beta}, \sigma_0/_{=_\beta}, \tau_0/_{=_\beta})$. Wir konstruieren nun eine neue Kripke-Struktur $\mathcal{K}^* = (W^*, \tau_0^*, \sigma_0^*)$ basierend auf $\mathcal{K}/_{=_\beta}$ durch

$$\begin{aligned} W^* &= \{\beta_s \mid [s] \in W/_{=_\beta}\} \\ \tau_0^*(a) &= \{(\beta_s, \beta_t) \mid ([s],[t]) \in \tau_0/_{=_\beta}(a)\} \\ \sigma_0^*(p) &= \{\beta_s \mid [s] \in \sigma_0/_{=_\beta}(p)\} \quad . \end{aligned}$$

$\mathcal{K}^*$ ist isomorph zu $\mathcal{K}/_{=_\beta}$, wie man leicht überprüft. Daher gilt $(\mathcal{K}^*, \beta_s) \models F$. Durch Hinzufügen der noch fehlenden Elemente zu W^* lässt sich $\mathcal{K}^*$ leicht zur regulären Standardstruktur von τ_0^* erweitern. Dieses τ_0^* ist dann die atomare Übergangsfunktion, deren Existenz vom Lemma behauptet wird. Die erforderliche Eigenschaft von σ_0^*, dass $\sigma_0^*(p) = \{\beta_s \mid \beta_s(p) = \mathbf{true}\}$, ist dabei nach Definition von β_s erfüllt. □

Falls die Semantik der atomaren Programme in Bezug auf die Aussagevariablen eindeutig festgelegt ist, so liefert Satz 3.2.9 einen Algorithmus, mit dessen Hilfe sich die Erfüllbarkeit einer regulär erfüllbaren Formel F in exponentieller Zeit in der Anzahl der Aussagevariablen $|\Phi_0|$ entscheiden lässt. Man prüft (per Model-Checking) einfach, ob F in der regulären Standardstruktur erfüllbar ist. Dies verbessert bekannte Komplexitätsschranken [Pra79] nur insofern, als diese bekannten Schranken exponentiell von der Größe $|F|$ der zu prüfenden Formel abhängen. Da das Erfüllbarkeitsproblem für PDL jedoch EXPTIME-vollständig ist [FL79] und da unsere Φ_0-regulären Strukturen zur Lösung des allgemeinen Problems für $|\Phi_0| \geq |\mathrm{FL}(F)|$ ausreichend sind, liegt dieses Resultat im Rahmen des Erwarteten. Unter praktischen Gesichtspunkten kann unser Ansatz allerdings vorteilhaft sein, da die Anzahl $|\Phi_0|$ der aussagenlogischen Variablen meist deutlich kleiner ist als die Formelgröße $|F|$.

Wir werden in den nächsten Abschnitten die Anwendung von Satz 3.2.9 durch die Spezifikation einer relativ allgemeinen Transitionsinterpretation illustrieren und versuchen, deren Nutzung zur Erfüllbarkeitsprüfung zu verdeutlichen.

3.2.5 Boolesche Zuweisungsprogramme

Wir werden nun eine Menge von speziellen atomaren PDL-Programmen, die Booleschen Zuweisungsprogramme, genauer untersuchen. Diese treten z.B. in regelbasierten Expertensystemen auf, oder, allgemeiner gesagt, in imperativen Programmen, die ausschließlich aussagenlogische Variablen oder Variablen mit endlichem Wertebereich verwenden. Wir betrachten atomare Zuweisungsprogramme der Form $x := b$ für eine Boolesche Variable x und einen Wert $b \in \mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$. Die Zuweisungsprogramme sollen die folgenden PDL-Eigenschaften haben:

Definition 3.2.10 *Sei $x \in \Phi_0$ und $b \in \mathbb{B}$. Dann ist das **Zuweisungsprogramm** $x := b$ ein Programm mit folgenden Eigenschaften für beliebiges $y \in \Phi_0$:*

1. $[x := b]y \Leftrightarrow \langle x := b\rangle y$
2. $[x := b]x$, *falls* $b = \mathbf{true}$, $[x := b]\neg x$, *falls* $b = \mathbf{false}$
3. *Falls* $y \neq x$, *so gilt* $y \Leftrightarrow [x := b]y$.

Wir bezeichnen die Konjunktion dieser Programmeigenschaften mit $F^{\mathrm{BA}}_{x,b}$.

Die erste Eigenschaft besagt, dass Zuweisungsprogramme deterministisch sind; durch die zweite Eigenschaft wird der beabsichtigte Effekt einer Zuweisung festgelegt, dass nämlich die Variable nach Ausführung der Zuweisung den gewünschten Wert angenommen hat; durch die letzte schließlich wird eine Änderung anderer Variablen als der Zuzuweisenden x ausgeschlossen.

Durch diese Definition wird die Transitionsrelation, die in diesem Fall sogar eine Funktion ist (s.u.), für die atomaren Programme festgelegt. Wir wollen nun die Eigenschaften betrachten, die eine Kripke-Struktur, in der $F^{\mathrm{BA}}_{x,b}$ gilt, haben muss.

Lemma 3.2.11 *Sei* $x \in \Phi_0$*,* $b \in \mathbb{B}$*,* $x := b \in \Pi_0$ *und* $\mathcal{K} = (W, \sigma_0, \tau_0)$ *eine Kripke-Struktur mit* $\mathcal{K} \models F^{\mathrm{BA}}_{x,b}$*. Dann gibt es für alle* s *ein* t^**, so dass* $(s, t^*) \in \tau_0(x{:=}b)$*, und es gilt für alle* $(s,t) \in \tau_0(x{:=}b)$

$$t \in \sigma_0(x) \Leftrightarrow b = \mathbf{true} \quad \textit{und} \tag{a}$$

$$t \in \sigma_0(y) \Leftrightarrow s \in \sigma_0(y) \quad \textit{für alle } y \in \Phi_0 \textit{ mit } y \neq x. \tag{b}$$

Beweis: Expansion der Semantik der drei Eigenschaften aus Def. 3.2.10 für $\mathcal{K} = (W, \tau_0, \sigma_0)$ liefert für beliebiges $y \in \Phi_0$:

$$(\forall t)((s,t) {\in} \tau_0(x{:=}b) \Rightarrow t {\in} \sigma_0(y)) \Leftrightarrow (\exists t)((s,t) {\in} \tau_0(x{:=}b) \wedge t {\in} \sigma_0(y)), \tag{i}$$

$$(\forall t)((s,t) \in \tau_0(x{:=}\mathbf{true}) \Rightarrow t \in \sigma_0(x)), \tag{ii}$$

$$(\forall t)((s,t) \in \tau_0(x{:=}\mathbf{false}) \Rightarrow t \notin \sigma_0(x)) \quad \text{und} \tag{iii}$$

$$s \in \sigma_0(y) \Leftrightarrow (\forall t)((s,t) \in \tau_0(x{:=}b) \Rightarrow t \in \sigma_0(y)) \quad \text{für } y \neq x. \tag{iv}$$

Äquivalenz (i), von links nach rechts gelesen, kann zu $(\forall s)(\exists t)((s,t) \in \tau_0(x := b))$ vereinfacht werden, wodurch die Existenz des geforderten t^* bezeugt ist. Aus (ii) zusammen mit (iii) ergibt sich direkt Bedingung (a), aus (iv) ergibt sich (b). □

Werden die Behauptungen dieses Lemmas zusammen genommen, so erkennt man leicht, dass die Transitionsrelation eines Zuweisungsprogramms $x := b$ funktional in der oben definierten Quotientenstruktur ist, und dass jede Kripke-Struktur, in der $F^{\mathrm{BA}}_{x,b}$ gilt, Φ_0-regulär ist.

Lemma 3.2.12 *Falls alle atomaren Programme Zuweisungsprogramme an Variablen in* Φ_0 *sind, d.h. falls* $\Pi_0 \subseteq \{x := b \mid x \in \Phi_0, b \in \mathbb{B}\}$*, so ist jede erfüllbare Formel* Φ_0*-regulär erfüllbar.*

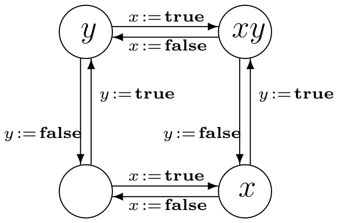
Beweis: Sei $F \in \Phi$ eine erfüllbare Formel. Dann gibt es eine Kripke-Struktur $\mathcal{K} = (W, \sigma_0, \tau_0)$ und ein $s^* \in W$ mit $(\mathcal{K}, s^*) \models F$. Da alle atomaren Programme Zuweisungsprogramme sind, gilt $\mathcal{K} \models F^{\mathrm{BA}}_{x,b}$ für alle $x := b \in \Pi_0$.

Sei $a = x := b$ ein beliebiges Zuweisungsprogramm aus Π_0, $s, s', t \in W$ mit $s =_\beta s'$ und $(s,t) \in \tau_0(a)$. Nach Lemma 3.2.11 gibt es dann ein t^* mit $(s', t^*) \in \tau_0(a)$. Außerdem ist nach Lemma 3.2.11(b) $\beta_s(y) = \beta_t(y)$ und $\beta_{s'}(y) = \beta_{t^*}(y)$ für $y \neq x$, also auch $\beta_t(y) = \beta_{t^*}(y)$ für $y \neq x$. Wegen Lemma 3.2.11(a) ist $\beta_t(x) = \beta_{t^*}(x)$, also $t^* =_\beta t$. Demnach muss $\mathcal{K}$ regulär sein und deshalb auch F Φ_0-regulär erfüllbar. $\Box$

Korollar 3.2.13 *Falls alle atomaren Programme Zuweisungsprogramme an Variablen in* Φ_0 *sind, so ist jede erfüllbare Formel in der* Φ_0*-regulären Standardstruktur erfüllbar, die aus der Summe*[10] *der individuellen* $\tau_0(a)$ *aller atomaren Programme* $a \in \Pi_0$ *resultiert.*

Beweis: Lemma 3.2.12 und Satz 3.2.9. $\Box$

Um nun Eigenschaften Boolescher Zuweisungsprogramme zu prüfen, können wir dies direkt in der von diesen generierten Φ_0-regulären Standardstruktur tun. Diese Struktur ist ein $|\Phi_0|$-dimensionaler Hyperwürfel, in dem atomare Zuweisungsprogramme benachbarte Ecken verbinden. Für den 2-dimensionalen Fall mit $\Phi_0 = \{x, y\}$ sieht der Hyperwürfel wie folgt aus:



Die Spezialisierung auf Boolesche Zuweisungsprogramme macht nun die Anwendung der folgenden Vereinfachung für das automatische Beweisen einer Eigenschaft F möglich: Falls F nur Boolesche Zuweisungsprogramme enthält, dann gilt

$$\models \left(\bigwedge_{x\in\Phi_0, b\in\mathbb{B}} F^{\mathrm{BA}}_{x,b} \right) \Rightarrow F \qquad \text{gdw.} \qquad \mathcal{K}^{\mathcal{S}}_{\Box} \models F \ ,$$

wobei $\mathcal{K}^{\mathcal{S}}_{\Box}$ die Φ_0-reguläre Standard-Struktur bezeichnet, in der τ_0 durch die Zuweisungsprogramme generiert ist (τ_0 ist also der $|\Phi_0|$-dimensionale Hyperwürfel). Für die Prüfung der Allgemeingültigkeit einer solchen PDL-Formel braucht man also lediglich *eine* Kripke-Struktur zu betrachten, wodurch die Beweisaufgabe zu einer Model-Checking-Aufgabe wird.

---

[10] Mengenvereinigung der Relationen.

3.3 Konfiguration in PDL

Ziel dieses Abschnitts ist die Formalisierung des Konfigurationsprogramms der Mercedes-Benz-Fahrzeuge, welches in Abschnitt 2.3.1 ausführlich dargestellt wurde, in PDL. Die dort angegebenen Algorithmen zur Zusteuerung, Baubarkeitsprüfung und Teilebedarfsermittlung werden wir nun nach PDL konvertieren. Diese eignen sich besonders deswegen für eine Übersetzung nach PDL, da sie im Gegensatz zu vielen anderen Formalismen nicht rein deklarativ, sondern zustandsbasiert sind, und weil die imperativen Programme zur Auftragsbearbeitung relativ einfach sind.

Für die Transformation nach PDL gehen wir von einer geordneten Menge $\mathcal{C} = (c_1, \ldots, c_n)$ von möglichen Auftragscodes und einer geordneten Menge $\mathcal{T} = (t_1, \ldots, t_k)$ von möglichen Teilen aus. Für die Codes sind in der Dokumentation (möglicherweise) Zusteuerungs- und Baubarkeitsregeln hinterlegt, für die Teile Selektionsregeln.

```
ALOGRITHMUS: Auftragsbearbeitung
EINGABE: $(x_1, \ldots, x_n) \in \mathbb{B}^n$        // Kundenauftrag
AUSGABE: $(b_1, \ldots, b_k) \in \mathbb{B}^k$        // Teilebedarf

do                                                    // Zusteuerung $Z$
    $Z_1 \wedge \neg x_1 \;\rightarrow\; x_1 := \mathbf{true}$
    $| \cdots |$
    $Z_n \wedge \neg x_n \;\rightarrow\; x_n := \mathbf{true}$
od

for $i = 1$ to $n$ do                                 // Baubarkeitsprüfung $B$
    if $x_i \wedge \neg B_i$ then fail

for $i = 1$ to $k$ do                                 // Teilebedarfsermittlung $T$
    if $T_i$ then $b_i := \mathbf{true}$
          else $b_i := \mathbf{false}$
```

Abbildung 3.6: PDL-Programm Auftragsbearbeitung

Anstelle der konkreten und doch recht komplexen Programme aus Abschnitt 2.3.1 wollen wir ein abstrakteres Auftragsbearbeitungsprogramm verwenden, das wir in Abbildung 3.6 dargestellt haben und das bereits in PDL formuliert ist. Dieses Programm nimmt eine Menge $E = \{e_1, \ldots, e_i\} \subseteq$

$\mathcal{C}$ von Bestellcodes als Eingabe und generiert als Ausgabe eine Liste der für diesen Auftrag benötigten Teile $A = \{a_1, \ldots, a_j\} \subseteq \mathcal{T}$. Sowohl die Menge E der Bestellcodes als auch die Menge A der benötigten Teile werden als charakteristische Funktion bezüglich $\mathcal{C}$ bzw. $\mathcal{T}$ dargestellt, also als Boolesche Vektoren der Dimension n bzw. k.

Das Programm besteht aus drei Teilen, die die drei aufeinanderfolgenden Stufen der Auftragsverarbeitung widerspiegeln. Wenn wir später auf den abstrakten Algorithmus Bezug nehmen, werden wir diese Teile mit Z, B und T bezeichnen, so dass das Gesamtprogramm aus Abbildung 3.6 sich als $Z\,;B\,;T$ darstellen lässt. Im Programm werden ferner abstrakte Zusteuerungsformeln Z_i, Baubarkeitsformeln B_i und Teileselektionsformeln T_i verwendet, die die folgende intuitive Bedeutung haben sollen:

- Ist die abstrakte Zusteuerungsformel Z_i unter der durch den Auftrag gegebenen Belegung wahr, so wird Code c_i dem Auftrag hinzugefügt.
- Kommt Code c_i im (eventuell durch die Zusteuerung modifizierten) Auftrag vor, so muss die Baubarkeitsformel B_i unter der durch den Auftrag gegebenen Belegung zu $\mathbf{true}$ evaluieren, damit der Auftrag akzeptiert wird.
- Sofern die abstrakte Teileselektionsformel T_i vom Auftrag erfüllt ist, wird das Teil t_i der Teileliste hinzugefügt. Ist dies nicht der Fall, so wird das Teil auch nicht in die Liste aufgenommen.

Wir wollen nun die Verbindung zwischen dem abstrakten Programm in PDL und dem konkreten, von DaimlerChrysler verwendeten Verfahren näher untersuchen. Dies betrifft zum einen die Generierung der abstrakten Formeln Z_i, B_i und T_i aus den gegebenen Formeln bzw. Regeln der Datenbank, zum anderen aber auch die Zuordnung von abstrakten Programmläufen zu konkreten, wobei insbesondere der Indeterminismus des Zusteuerungsteils im PDL-Programm zu berücksichtigen ist. Ein- und Ausgabe der beiden Algorithmen stimmen überein, so dass wir zur Untersuchung der Abstraktionseigenschaft uns darauf beschränken können, zu prüfen, ob jedem Programmlauf des konkreten Programms ein Programmlauf des abstrakten Programms zugeordnet werden kann.

Wenden wir uns nun also der ersten Frage, der Generierung der abstrakten Formeln zu. Im konkreten Programm hängt die Zusteuerung von den Formeln $\mathrm{ZRegel}(c, g, p, v)$, $\mathrm{BRegel}(c, g, p, v)$ und $\mathrm{PCRegel}(c)$ der Datenbank ab.

Außerdem gibt es verschiedene Zusteuerungsläufe (CG, BG) und eine Begrenzung der Zusteuerungsschritte. Beides werden wir im abstrakten Algorithmus nicht übernehmen, so dass wir nur eine Art von Zusteuerungsläufen kennen und die Begrenzung der Anzahl dieser Läufe aufheben, indem Zusteuerungsschritte so lange wie möglich durchgeführt werden.

Fassen wir, dies alles berücksichtigend, die konkreten Formeln für einen Code c_i zusammen, so erhalten wir unter Verwendung der Hilfsformeln

$$\mathrm{H}(c_i, l) \;:=\; \bigvee_{\substack{g \in \mathrm{Gruppen}(c_i) \\ p \in \mathrm{Positionen}(c_i, g) \\ o \in \{\mathrm{CG}, \mathrm{BG}\} \\ v \in \mathrm{Varianten}(c_i, g, p, l, o)}} \mathrm{ZRegel}(c_i, g, p, v) \wedge \mathrm{BRegel}(c_i, g, p, v)$$

für die abstrakte Zusteuerungsformel Z_i die folgende Definition:

$$Z_i \;:=\; \mathrm{PCRegel}(c_i) \wedge \bigvee_{l \in \{\mathrm{L}, \mathrm{R}\}} l \wedge \mathrm{H}(c_i, l) \;\;.$$

Dabei identifizieren wir in der Formel Z_i vorkommende Codes c_j mit den entsprechenden Booleschen Variablen x_j des in Abbildung 3.6 angegebenen Algorithmus. Außerdem verwenden wir die in Abschnitt 2.3.1 beschriebene Eigenschaft, dass immer genau einer der beiden Lenkungscodes L und R in einem Auftrag vorhanden sein muss. Diese lässt sich z.B. durch das Hinzufügen der Baubarkeits- und Zusteuerungsregeln

$$\begin{aligned} Z_L &:= \neg R \qquad & B_L &:= \neg R \\ Z_R &:= \neg L \qquad & B_R &:= \neg L \end{aligned}$$

für die Codes L und R erreichen.

Die abstrakten Baubarkeitsformeln B_i ergeben sich auf ähnliche Weise aus den konkreten Formelbausteinen $\mathrm{BRegel}(c, g, p, v)$ und $\mathrm{PCRegel}(c)$ der Produktdokumentation, diesmal unter Verwendung der Hilfsformeln

$$\mathrm{G}(c_i, l) \;:=\; \bigwedge_{\substack{g \in \mathrm{Gruppen}(c_i) \\ p \in \mathrm{Positionen}(c_i, g)}} \; \bigvee_{v \in \mathrm{Varianten}(c_i, g, p, l)} \mathrm{BRegel}(c, g, p, v) \;\;.$$

Damit kann man die Baubarkeitsformeln B_i definieren als

$$B_i \;:=\; \mathrm{PCRegel}(c_i) \wedge \bigwedge_{l \in \{\mathrm{L}, \mathrm{R}\}} (l \Rightarrow \mathrm{G}(c_i, l)) \;\;.$$

Schlussendlich betrachten wir noch die Teileselektionsformeln T_i, für die sich eine einfache, direkte Übersetzung ergibt. Bezeichnen wir die lange Coderegel (siehe Abschnitt 2.3.1) eines an Positionsvariante t_i dokumentierten Teils mit $\mathrm{LCRegel}(t_i)$, so ergibt sich für die abstrakte Teileselektionsformeln die Definition

$$T_i \ := \ \mathrm{LCRegel}(t_i) \ .$$

Wir wollen hier noch, da für die spätere Verifikation wichtig, anmerken, dass die Formeln T_i nur Codevariablen x_i, aber keine Teilevariablen b_i enthalten.

Auf eine detaillierte Darstellung des Beweises, dass der konkrete, in DIALOG implementierte Algorithmus tatsächlich eine Spezialisierung des abstrakten Algorithmus darstellt, wollen wir verzichten (dieser ist in [KS00] dargestellt) und stattdessen die Voraussetzungen und Annahmen, die zur Einhaltung dieser Eigenschaft notwendig sind, genauer herausarbeiten, uns ansonsten aber mit informellen, skizzenhaften Begründungen zufriedengeben.

Erste Voraussetzung ist, dass die Anzahl der CG- und BG-Läufe des konkreten Algorithmus für alle theoretisch möglichen Aufträge ausreichend sind und somit immer eine vollständige Zusteuerung erreicht wird. Dies muss in der Implementation nicht zwangsläufig der Fall sein, die Annahme vereinfacht die Untersuchung aber deutlich, da man dann immer von einer kompletten Durchführung der Zusteuerung ausgehen kann. Dies ermöglicht auch erst die PDL-Formalisierung der Zusteuerung durch *iterative guarded commands*. Ist diese Voraussetzung gegeben, lässt sich jedem Programmlauf des konkreten Zusteuerungsalgorithmus (Abbildung 2.9), welcher durch die Sequenz der zugesteuerten Codes $(z_1, \ldots, z_l)$ eindeutig beschrieben werden kann, ein Programmlauf des abstrakten Zusteuerungsalgorithmus zuordnen. Dieser ist durch dieselbe Zusteuerungssequenz charakterisiert; die Möglichkeit der Ausführung der einzelnen Zusteuerungsschritte im abstrakten Algorithmus, sofern sie tatsächlich zu einer Modifikation des Auftrags beitragen, lässt sich anhand der Formeln Z_i leicht überprüfen.

Für die Baubarkeitsprüfung und die Teilebedarfsermittlung kommt man zu einer ähnlichen Übereinstimmung der abstrakten und konkreten Programmläufe, so dass sich insgesamt ergibt, dass jeder Programmlauf des konkreten Auftragsbearbeitungsprogramms durch einen Lauf des abstrakten Programms simuliert werden kann, vorausgesetzt obige Annahme über die Anzahl der Zusteuerungsläufe trifft zu.

Wir beschließen diesen Abschnitt mit Lemmata, die wichtige Eigenschaften des Auftragsbearbeitungsprogramms festhalten. Wir beginnen mit zwei Eigenschaften des Zusteuerungsprogramms, zuerst eines zur Termination:

Lemma 3.3.1 *Das Zusteuerngsprogramm Z ist terminierend, d.h. es gilt $\mathcal{K}_\Box^\mathcal{S} \models \langle Z \rangle \top$ und $\mathcal{K}_\Box^\mathcal{S} \models \neg\Delta((Z_1 \wedge \neg x_1?x_1 := \mathbf{true}) \cup \cdots \cup (Z_n \wedge \neg x_n?x_n := \mathbf{true}))$.*

In diesem Lemma haben wir die Notation $\Delta\alpha$ verwendet, die aus der Erweiterung ΔPDL [Str82] stammt und der Syntax von PDL das Divergenzsymbol Δ hinzufügt. Für Programme $\alpha \in \Pi$ besitzt dieses Symbol die folgende, nicht in gewöhnlichem PDL ausdrückbare Semantik (für die Kripke-Struktur $\mathcal{K} = (W, \tau_0, \sigma_0)$):

$$\sigma(\Delta\alpha) \;=\; \{s_0 \in W \mid (\exists s_1, s_2, \dots)(\forall i \geq 0)((s_i, s_{i+1}) \in \tau(\alpha))\} \;.$$

Es drückt also aus, dass für das Programm α^* ein unendlicher, d.h. nicht terminierender Programmlauf existiert, so dass die Behauptung des Lemmas ist, dass jeder Lauf des Zusteuerungsprogramms endlich ist, das Zusteuerungsprogramm also terminiert.

Beweis: Wir werden nur die Idee des Terminationsbeweises skizzieren. Wir betrachten eine Zustandsfolge $s_0, \dots, s_k, \dots$, die sich bei Durchlauf des Zusteuerungsprogramms jeweils am Ende der Schleife ergibt, und für jeden dieser Zustände die Mächtigkeit m_i der in s_i mit falsch bewerteten Variablen der Menge $\{x_1, \dots, x_n\}$, d.h. $m_i = |\{x_j \mid 1 \leq j \leq n, s_i \notin \sigma(x_j)\}|$. In jedem Schleifendurchlauf wird eine zuvor mit falsch belegte Variable x_j auf wahr gesetzt, so dass m_i in jedem Durchlauf kleiner wird. Da $m_i \geq 0$ für alle i gelten muss, ist die Termination damit bewiesen. □

Das nun folgende Lemma drückt die leicht aus dem Programmtext abzulesende Eigenschaft aus, dass durch das Zusteuerungsprogramm keine der Variablen x_i von **true** nach **false** abgeändert werden kann:

Lemma 3.3.2 *Für das Zusteuerungsprogramm Z, das Baubarkeitsprüfprogramm B und alle $x_i \in \mathcal{C}$ gilt:*

1. $\mathcal{K}_\Box^\mathcal{S} \models x_i \Rightarrow [Z; B]x_i$ *und*
2. $\mathcal{K}_\Box^\mathcal{S} \models \langle Z; B \rangle x_i \Rightarrow x_i$*, falls* $Z_i \equiv \bot$.

Beweis:

1. Sei $s_0 \in \sigma(x_i)$. Wir zeigen, dass für jede Zustandsfolge $s_0, \dots, s_k$ eines Programmlaufs des Zusteuerungs- und Prüfprogramms und für alle j mit $0 \leq j \leq k$ gilt, dass $s_j \in \sigma(x_i)$. Da die einzigen atomaren Programme in Z und B Zuweisungen der Form $x_i := \mathbf{true}$ sind, kann nach Definition 3.2.10 für zwei aufeinanderfolgende Zustände s_j und s_{j+1} nur gelten, dass entweder $s_{j+1} \in \sigma(x_i) \Leftrightarrow s_j \in \sigma(x_i)$ oder $s_{j+1} \in \sigma(x_i)$, womit die Behauptung bereits bewiesen ist.

2. In einer der vorhergehenden analogen Argumentation stellt man fest, dass im Fall $Z_i \equiv \bot$ alle atomaren Programme den Wert der Variablen x_i unverändert lassen, so dass $s_{j+1} \in \sigma(x_i) \Leftrightarrow s_j \in \sigma(x_i)$ für alle Berechnungssequenzen gelten muss. Daraus folgt auch schon die Behauptung.

□

Letztendlich geben wir noch ein Lemma zum Baubarkeitsprüfprogramm an, das dessen Äquivalenz zu einem Programm bestehend nur aus einem einzigen Test feststellt:

Lemma 3.3.3 *Das Baubarkeitsprüfprogramm B ist äquivalent zu dem PDL-Programm $((x_1 \Rightarrow B_1) \wedge \dots \wedge (x_n \Rightarrow B_n))?$.*

Beweis: Expansion der syntaktischen Abkürzungen in Programm B unter Berücksichtigung der Semantik σ liefert:

$$\begin{aligned}
B &\equiv \textbf{for } i = 1 \textbf{ to } n \textbf{ do if } x_i \wedge \neg B_i \textbf{ then fail} \\
&\equiv \textbf{if } x_1 \wedge \neg B_1 \textbf{ then fail}; \dots ; \textbf{if } x_n \wedge \neg B_n \textbf{ then fail} \\
&\equiv ((x_1 \wedge \neg B_1)?; \bot? \cup \neg(x_1 \wedge \neg B_1)?; \top?); \dots ; \\
&\qquad\qquad ((x_n \wedge \neg B_n)?; \bot? \cup \neg(x_n \wedge \neg B_n)?; \top?) \\
&\equiv \neg(x_1 \wedge \neg B_1)?; \dots ; \neg(x_n \wedge \neg B_n)? \\
&\equiv ((x_1 \Rightarrow B_1) \wedge \dots \wedge (x_n \Rightarrow B_n))?
\end{aligned}$$

□

3.4 Transformation nach Aussagenlogik

Wir wenden uns nun der Transformation von Aussagen über PDL-Programme in Aussagen der reinen Propositionallogik zu. Dabei bezeichnen wir in diesem Abschnitt mit $\Phi_A \subseteq \Phi$ die Formeln der reinen Aussagenlogik, also diejenigen PDL-Formeln, die keine Modalitäten enthalten.

Ziel ist es, Gültigkeitsaussagen über Boolesche Zuweisungsprogramme, also Aussagen der Form

$$\models \left(\bigwedge_{x \in \Phi_0, b \in \mathbb{B}} F^{\mathrm{BA}}_{x,b} \right) \Rightarrow F$$

für PDL-Formeln F, die als atomare Programme nur Boolesche Zuweisungsprogramme enthalten, zu beweisen. Nach obigen Vorarbeiten aus Abschnitt 3.2.5 ist dies äquivalent zu $\mathcal{K}^{\mathcal{S}}_{\Box} \models F$, wobei $\mathcal{K}^{\mathcal{S}}_{\Box}$ die Φ_0-reguläre Standard-Struktur der Zuweisungsprogramme bezeichnet. Für diese spezielle Form von Aussagen werden wir nun die Transformation nach Aussagenlogik beschreiben.

Definition 3.4.1 *Sei $F \in \Phi_A$ eine aussagenlogische Formel, $x \in \Phi_0$ und $b \in \mathbb{B}$. Die **Einschränkung** $F|_{x=b}$ ist dann rekursiv definiert durch*

$$\begin{aligned} y|_{x=b} &= \begin{cases} \top & \textit{falls } x = y \textit{ und } b = \mathbf{true}, \\ \bot & \textit{falls } x = y \textit{ und } b = \mathbf{false}, \\ y & \textit{falls } x \neq y \end{cases} \\ \bot|_{x=b} &= \bot \\ (G \Rightarrow H)|_{x=b} &= G|_{x=b} \Rightarrow H|_{x=b} \end{aligned}$$

Einschränkungen werden wir verwenden, um die Effekte von Zuweisungsprogrammen zu beschreiben.

Lemma 3.4.2 *Sei $\mathcal{K} = (W, \tau_0, \sigma_0)$ eine Kripke-Struktur, in der alle atomaren Programme Zuweisungsprogramme sind (d.h. es gilt insbesondere $\mathcal{K} \models F^{\mathrm{BA}}_{x,b}$ aus Definition 3.2.10), $x \in \Phi_0$ undb $\in \mathbb{B}$. Ferner sei $s, t \in W$, $(s,t) \in \tau(x\!:=\!b)$ und $F \in \Phi_A$ eine aussagenlogische Formel. Dann gilt $(\mathcal{K}, s) \models F|_{x=b} \Leftrightarrow (\mathcal{K}, t) \models F$.*

Beweis: Wir beweisen die Behauptung per Induktion über die Struktur von F. Angenommen F ist atomar, d.h. $F = p \in \Phi_0$. Wir unterscheiden drei Fälle: Erstens, falls $p \neq x$, so ist $p|_{x=p} = p$, und nach Lemma 3.2.11(b) gilt die Behauptung. Zweitens, falls $p = x$ und $b = \mathbf{true}$, so ist $p|_{x=b} = \top$ und somit $(\mathcal{K}, s) \models F|_{x=b}$. Durch Lemma 3.2.11(a) wissen wir, dass $(\mathcal{K}, t) \models F$, die Behauptung stimmt also. Drittens, falls $p = x$ und $b =$ **false**, dann ist $p|_{x=b} = \bot$, so dass $(\mathcal{K}, s) \not\models F|_{x=b}$. Wiederum reicht nun Lemma 3.2.11(a)aus, um die Behauptung zu beweisen. Die anderen Fälle der Induktion folgen aus der Tatsache, dass die Restriktion eine homomorphe Erweiterung des atomaren Falls darstellt. $\square$

Lemma 3.4.2 liefert die Grundlage zur Übersetzung von Erfüllbarkeitsproblemen aus der dynamischen Aussagenlogik in die (pure) Aussagenlogik. Durch Anwendung der folgenden Ersetzungsregeln $\mathcal{R}$ (an der am weitest innenliegenden Stelle einer Formel zuerst, also “innermost”) auf eine PDL-Formel $G \in \Phi$ kann diese in eine aussagenlogische Formel $G_A \in \Phi_A$ transformiert werden:

$$\begin{array}{rcl}
[x{:=}b]F & \rightsquigarrow & F|_{x=b} \\
[\alpha \cup \beta]F & \rightsquigarrow & [\alpha]F \wedge [\beta]F \\
[\alpha\,;\beta]F & \rightsquigarrow & [\alpha][\beta]F \\
[\alpha^*]F & \rightsquigarrow & \nu G\ .\ F \wedge [\alpha]G \\
[G?]F & \rightsquigarrow & G \Rightarrow F
\end{array}$$

Die erste Regel dieses Termersetzungssystems lässt sich durch Lemma 3.4.2 rechtfertigen, alle anderen, bis auf die Fixpunktregel für α^*, sind PDL-Äquivalenzen, die aus Segerbergs Axiomatisierung von PDL (siehe Abbildung 3.4) stammen. Die Fixpunktregel, in der $\nu G.f(G)$ den größten Fixpunkt von f bezeichnet, also die größte Formel (hinsichtlich Anzahl von Modellen) für die $G \equiv f(G)$ gilt, ist auch von diesem Axiomensystem inspiriert. Fixpunktberechnung und Regelanwendung sollen dabei intermittierend erfolgen, so dass ein neu enstandener ν-Operator durch die Fixpunktberechnung sofort wieder eliminiert wird (siehe Beispiel 2 weiter unten).

Die durch dieses Termersetzungssystem induzierte Transformationsprozedur besitzt die folgende wichtige Erhaltungseigenschaft:

Lemma 3.4.3 *Sei $F \in \Phi$ eine PDL-Formel, in der alle atomaren Programme Boolesche Zuweisungsprogramme sind, und $F_A \in \Phi_A$ die aussagenlogische Formel, die dadurch entsteht, dass man F bezüglich der Ersetzungsregeln $\mathcal{R}$ normalisiert. Dann ist F (in PDL) genau dann erfüllbar, wenn F_A (aussagenlogisch) erfüllbar ist.*

Beweis: Betrachten wir eine beliebige Reduktionsregel $L \rightsquigarrow R$ aus $\mathcal{R}$. Wir werden zeigen, dass $\models L \Leftrightarrow R$ in PDL für jede dieser Regeln gilt. Die sich aus der ersten Regel ergebende Äquivalenz folgt aus Lemma 3.4.2 zusammen mit der Definition von σ. Die Äquivalenzen der anderen Regeln, mit Ausnahme der Fixpunktregel, sind auf Grund von Segerbergs Axiomatisierung gültig. Für den Fixpunktausdruck auf der rechten Seite der Regel für α^* stellen wir fest, dass

$$G = \bigwedge_{n \in \mathbb{N}} [\alpha^n]F$$

eine Lösung dieser Fixpunktgleichung ist. Diese ist darüberhinaus äquivalent zur linken Regelseite $[\alpha^*]F$, wie sich aus Eigenschaft 2 aus Lemma 3.2.2 ergibt. Da die Funktion f der Fixpunktgleichung $G \equiv f(G)$ monoton ist, ergibt sich aus dem Knaster-Tarski-Theorem [Tar55] die Eindeutigkeit des Fixpunkts, so dass dieser auch der größte sein muss. □

Da in PDL eine Formel F genau dann eine Tautologie ist, wenn $\neg F$ nicht erfüllbar ist (siehe z.B. Fischer und Ladner [FL77]), gilt das Lemma auch, wenn man "Erfüllbarkeit" durch "Gültigkeit" ersetzt.

Korollar 3.4.4 *Sei $F \in \Phi$ eine PDL-Formel, in der alle atomaren Programme Boolesche Zuweisungsprogramme sind, und $F_A \in \Phi_A$ die aussagenlogische Formel, die dadurch entsteht, dass man F bezüglich der Ersetzungsregeln $\mathcal{R}$ normalisiert. Dann ist F (in PDL) genau dann allgemeingültig, wenn F_A (aussagenlogisch) allgemeingültig ist.*

In einer tatsächlichen Implementierung des Regelsystems und der Transformationsprozedur könnte die Fixpunktberechnung entweder unter Verwendung Binärer Entscheidungsdiagramme (*binary decision diagrams, BDDs* [Bry92]) realisiert werden oder durch Approximation des Fixpunkts, indem die (monotone) Fixpunktgleichung bis zu einer vorgegebenen Schranke $k^{\max} \in \mathbb{N}$ entfaltet wird (*unfolding*). In diesem Fall wäre die Fixpunktregel des Termersetzungssystems durch die folgenden drei Regeln zu ersetzen:

$$\begin{array}{rcl}
[\alpha^*]F & \rightsquigarrow & [\alpha^{\leq k_{\max}}]F \\
[\alpha^{\leq 0}]F & \rightsquigarrow & F \\
[\alpha^{\leq k}]F & \rightsquigarrow & [\alpha^{\leq k-1}]F \wedge [\alpha^k]F \quad \text{für } k > 0
\end{array}$$

Ist dabei $k^{\max}$ groß genug gewählt, so ist G erfüllbar genau dann, wenn G_A erfüllbar ist. Selbst wenn dies nicht der Fall sein sollte, liefert G_A eine gangbare Näherung für den Fall, dass nur positive oder nur negative Vorkommen des Iterationsoperators in der Formel auftreten. Die erhaltene Methode erinnert an das *Bounded Model-Checking* [CBRZ01], mit dem sie auch verbindet, dass die so generierten Probleme gut durch effiziente aussagenlogische Erfüllbarkeitsbeweiser (SAT-Checker) entschieden werden können.

Wir wollen nun die Transformation nach Aussagenlogik anhand zweier Beispiele eingehender erläutern.

Beispiel 1: Sei G_1 die durch $[\textbf{if}\ \neg x \wedge \neg y\ \textbf{then}\ x := \textbf{true}](x \vee y)$ gegebene PDL-Formel. Anwendung der Ersetzungsregeln und einfache Boolesche Umformungen liefern dann Schritt für Schritt:

$$\begin{array}{rl}
G_1 & \equiv\ [(\neg x \wedge \neg y)?\,;x{:=}\textbf{true} \cup (x \vee y)?\,;\top?](x \vee y) \\
& \rightsquigarrow^+ [(\neg x \wedge \neg y)?][x{:=}\textbf{true}](x \vee y)\ \wedge\ [(x \vee y)?](x \vee y) \\
& \rightsquigarrow^+ \neg x \wedge \neg y\ \Rightarrow\ [x{:=}\textbf{true}](x \vee y) \\
& \rightsquigarrow\ \neg x \wedge \neg y\ \Rightarrow\ (x \vee y)|_{x=\textbf{true}} \\
& \equiv\ \neg x \wedge \neg y\ \Rightarrow\ \top \\
& \equiv\ \top
\end{array}$$

G_1 ist also erfüllbar (und sogar allgemeingültig).

Beispiel 2: Sei α_2 das Programm $\textbf{while}\ x\ \textbf{do}\ y{:=}\textbf{false}$, wobei $x \neq y$, und G_2 die Aussage, dass α_2 unter der Annahme, dass x gilt, nie terminiert, d.h. $G_2 = x \Rightarrow \neg\langle\alpha_2\rangle\top$. Wir überprüfen dazu die Erfüllbarkeit der Negation $\neg G_2$:

$$\begin{array}{rl}
\neg G_2 & =\ \neg(x \Rightarrow \neg\langle\textbf{while}\ x\ \textbf{do}\ y{:=}\textbf{false}\rangle\top) \\
& \equiv\ x \wedge \neg[(x?\,;y{:=}\textbf{false})^*\,;\neg x?]\bot \\
& \rightsquigarrow\ x \wedge \neg[(x?;y{:=}\textbf{false})^*][\neg x?]\bot \\
& \rightsquigarrow\ x \wedge \neg[(x?;y{:=}\textbf{false})^*]x \\
& \rightsquigarrow\ x \wedge \neg(\nu H\ .\ x \wedge [x?;y{:=}\textbf{false}]H) \\
& \rightsquigarrow\ x \wedge \neg(\nu H\ .\ x \wedge [x?][y{:=}\textbf{false}]H) \\
& \rightsquigarrow^+ x \wedge \neg(\nu H\ .\ x \wedge (x \Rightarrow H|_{y=\textbf{false}})) \\
& \equiv\ x \wedge \neg(\nu H\ .\ x \wedge H|_{y=\textbf{false}}) \\
& \equiv\ x \wedge \neg x \\
& \equiv\ \bot
\end{array}$$

Die zweitletzte Äquivalenz gilt aufgrund der Lösung x der Fixpunktgleichung $\nu H\ .\ x \wedge H|_{y=\textbf{false}}$, die sich durch das bekannte Approximationsverfahren, startend mit $H_0 = \top$ und weiter mit $H_1 = x \wedge H_0|_{y=\textbf{false}} \equiv x$, $H_2 = x \wedge H_1|_{y=\textbf{false}} \equiv x$, errechnen lässt.

Alternativ ergibt die Transformation mittels Approximation für $k_{\max} = 2$:

$$\begin{aligned}
\neg G_2 \;&\rightsquigarrow^+ x \wedge \neg[(x?;y{:=}\mathbf{false})^*]x\\
&\rightsquigarrow^+ x \wedge \neg(x \wedge [x?;y{:=}\mathbf{false}]x \wedge [x?;y{:=}\mathbf{false}][x?;y{:=}\mathbf{false}]x)\\
&\rightsquigarrow^+ x \wedge \neg(x \wedge (x \Rightarrow x|_{y=\mathbf{false}}) \wedge [x?][y{:=}\mathbf{false}][x?][y{:=}\mathbf{false}]x)\\
&\rightsquigarrow^+ x \wedge \neg(x \wedge (x \Rightarrow [y{:=}\mathbf{false}](x \Rightarrow x|_{y=\mathbf{false}})))\\
&\equiv \;\; x \wedge \neg(x \wedge (x \Rightarrow [y{:=}\mathbf{false}]\top))\\
&\rightsquigarrow \;\; x \wedge \neg(x \wedge (x \Rightarrow \top))\\
&\equiv \;\; \bot
\end{aligned}$$

Die Formel $\neg G_2$ ist also unerfüllbar, G_2 daher allgemeingültig. Unter der Annahme, dass x gilt, terminiert das Programm α_2 also nie (in weniger als $k_{\max} = 2$ Schleifendurchläufen). Man überprüft leicht, dass die entfaltete Iteration für $k_{\max} > 2$ dieselbe Formel wie für $k_{\max} = 2$ liefert, und somit die Nichttermination auch im Allgemeinen gilt (was auch schon die direkte Berechnung des ν-Operators ergab).

Abschließend wollen wir noch die Erfüllbarkeitsäquivalenz zweier PDL-Formeln definieren und einige sich daraus ergebende PDL-Äquivalenzen vorstellen, die später bei der Verifikation eine entscheidende Problemreduktion ermöglichen werden.

Definition 3.4.5 *Zwei PDL-Formeln $F, G \in \Phi$ heißen **erfüllbarkeitsäquivalent** in einer Kripke-Struktur $\mathcal{K} = (W, \tau_0, \sigma_0)$, wenn F genau dann erfüllbar ist in $\mathcal{K}$, wenn auch G erfüllbar in $\mathcal{K}$ ist, d.h. wenn*

$$(\exists s_0 \in W)\,((\mathcal{K}, s_0) \models F) \quad \Leftrightarrow \quad (\exists s_1 \in W)\,((\mathcal{K}, s_1) \models G) \ .$$

Wir schreiben dann auch $F \overset{\text{ERF}}{\equiv}_{\mathcal{K}} G$ (oder einfach nur $F \overset{\text{ERF}}{\equiv} G$, sofern $\mathcal{K}$ sich aus dem Zusammenhang ergibt).

Man prüft leicht, z.B. indem man $F = \neg x \vee x$ und $G = x$ wählt, dass aus $F \overset{\text{ERF}}{\equiv} G$ nicht $\neg F \overset{\text{ERF}}{\equiv} \neg G$ folgen muss, dass Erfüllbarkeitsäquivalenz also unter Negation nicht erhalten bleibt.

Lemma 3.4.6 *Für alle Kripke-Strukturen $\mathcal{K} = (W, \tau_0, \sigma_0)$, $F, G \in \Phi$, $\alpha \in \Pi$ gilt:*

1. *Falls $\langle \mathbf{while}\ F\ \mathbf{do}\ \alpha \rangle G$ erfüllbar ist in $\mathcal{K}$, so auch $\langle \neg F? \rangle G$.*

2. $\langle \neg F?\rangle G \;\Rightarrow\; \langle \mathbf{while}\ F\ \mathbf{do}\ \alpha\rangle G$

3. $\langle \mathbf{while}\ F\ \mathbf{do}\ \alpha\rangle G \overset{\mathit{ERF}}{\equiv} \langle \neg F?\rangle G$

4. *Falls* $[\mathbf{while}\ F\ \mathbf{do}\ \alpha]G$ *erfüllbar ist in* $\mathcal{K}$ *und* $\mathbf{while}\ F\ \mathbf{do}\ \alpha$ *terminiert, d.h.* $\mathcal{K} \models \langle \mathbf{while}\ F\ \mathbf{do}\ \alpha\rangle \top$*, so ist auch* $[\neg F?]G$ *erfüllbar in* $\mathcal{K}$*.*

5. *Falls* $\mathcal{K} \models [\neg F?]G$*, so ist* $[\mathbf{while}\ F\ \mathbf{do}\ \alpha]G$ *erfüllbar in* $\mathcal{K}$*.*

Beweis: Zur Verkürzung der Schreibweise verwenden wir im folgenden die Definition $\beta := \mathbf{while}\ F\ \mathbf{do}\ \alpha$.

1. Angenommen es gibt ein $s \in W$, so dass $(\mathcal{K}, s) \models \langle\beta\rangle G$. Dann ergibt sich nach Definition der Semantik von $\langle\cdot\rangle$, dass $(\exists t \in W)((s,t) \in \tau(\beta) \wedge t \in \sigma(G))$, und nach Expansion des **while**-Ausdrucks $(\exists t \in W)((s,t) \in \tau((F?\,;\alpha)^*\,;\neg F?) \wedge t \in \sigma(G))$ und weiter $(\exists t \in W)((s,t) \in \tau((F?\,;\alpha)^*) \wedge t \in \sigma(\neg F) \wedge t \in \sigma(G))$. Somit gilt also $(\mathcal{K}, t) \models \neg F \wedge G$, und nach Eigenschaft 3 aus Lemma 3.2.2 demnach $(\mathcal{K}, t) \models \langle \neg F?\rangle G$.

2. Sei $s \in \sigma(\langle\neg F?\rangle G)$. Dann ist nach Eigenschaft 3 aus Lemma 3.2.2 auch $s \in \sigma(\neg F)$, also auch $(s,s) \in \tau((F?\,;\alpha)^*\,;\neg F?)$. Außerdem gilt, wiederum nach Lemma 3.2.2, $s \in \sigma(G)$. Insgesamt erhalten wir also $(\mathcal{K}, s) \models \langle \mathbf{while}\ F\ \mathbf{do}\ \alpha\rangle G$, das gewünschte Resultat.

3. Folgt direkt aus 1. und 2.

4. Wenn $\beta = \mathbf{while}\ F\ \mathbf{do}\ \alpha$ terminiert, so gilt $(\exists t)((s,t) \in \tau(\beta))$ für alle $s \in W$. Des weiteren gilt

$$\begin{array}{ll} & s \in \sigma([\beta]G) \\ \Leftrightarrow & (\forall t)((s,t) \in \tau(\beta) \Rightarrow t \in \sigma(G)) \\ \Leftrightarrow & (\forall t)((s,t) \in \tau((F?\,;\alpha)^*) \wedge t \in \sigma(\neg F) \Rightarrow t \in \sigma(G)) \end{array}$$

sowie

$$s \in \sigma([\neg F?]G) \quad \Leftrightarrow \quad s \in \sigma(\neg F) \Rightarrow s \in \sigma(G)$$

für alle $s \in W$. Nehmen wir nun also an, dass es ein s_0 gibt mit $(\mathcal{K}, s_0) \models [\beta]G$. Dann gilt auf Grund obiger Äquivalenz

$$(\forall t)((s_0,t) \in \tau((F?\,;\alpha)^*) \wedge t \in \sigma(\neg F) \Rightarrow t \in \sigma(G)),$$

und da wegen der Termination von β ein t^* existiert mit $(s_0, t^*) \in \tau((F?\,;\alpha)^*)$, gilt $t^* \in \sigma(\neg F) \Rightarrow t^* \in \sigma(G)$, also auch $t^* \in \sigma([\neg F?]G)$, so dass $(\mathcal{K}, t^*) \models [\neg F?]G$.

5. Nach Voraussetzung gilt für alle $s \in W$, dass $(\mathcal{K}, s) \models [\neg F?]G$. Dann gilt nach zweiter obiger Äquivalenz $s \in \sigma(\neg F) \Rightarrow s \in \sigma(G)$. Nun angenommen, es gibt ein s^* mit $s^* \in \sigma(\neg F)$. Dann gilt neben $s^* \in \sigma(G)$ auch $(s^*, s^*) \in \tau((F?\,;\alpha)^*)$ und außerdem $(s^*, t) \notin \tau((F?\,;\alpha)^*)$ für alle $t \neq s^*$. Insgesamt erhält man also

   $$(\forall t)((s^*, t) \in \tau((F?\,;\alpha)^*) \wedge t \in \sigma(\neg F) \Rightarrow t \in \sigma(G)),$$

   und daher nach obiger ersten Äquivalenz $(\mathcal{K}, s^*) \models [\beta]G$. Wenn es kein solches s^* gibt, so gilt $s \in \sigma(F)$ für alle $s \in W$. In obiger ersten Äquivalenz ist die Voraussetzung der Implikation der letzten Zeile daher nie erfüllt, die gesamte Implikation also immer gültig (das Programm β terminiert nicht). Daher gilt in diesem Fall sogar das stärkere, vom Zustand unabhängige $\mathcal{K} \models [\beta]G$.

$\square$

4

Verifikation

Verifikation dient dazu, gewünschte Eigenschaften sicherzustellen bzw. unerwünschte mit Sicherheit auszuschließen. In unserem Fall sind dies Konsistenz- oder Validitätseigenschaften, die die Dokumentation betreffen, also die formale Darstellung aller Produktvarianten und deren Struktur. Als Beispiel sei die Prüfung der Eigenschaft genannt, dass es zu jeder Komponente mindestens eine Produktvariante gibt, in der die Komponente Verwendung findet.

Wir beschränken uns in der Darstellung der Verifikation auf die bereits ausführlich erläuterte Dokumentation der Mercedes-Benz-Fahrzeuge in DIALOG. Für diese Dokumentation haben wir mit dem Vorliegen ihrer PDL-Formalisierung auch schon eine wichtige Voraussetzung zur logisch exakten und eindeutigen Untersuchung erfüllt.

Wie oben bereits erwähnt, besteht die Dokumentation einer Fahrzeugbaureihe in DIALOG aus einer so großen Menge an aussagenlogischen Regeln (siehe Tabelle 2.1), dass eine manuelle Prüfung einer Konsistenzeigenschaft für alle möglichen Produktvarianten praktisch nicht durchführbar ist. So besteht die Dokumentation der Limousinen der C-Klasse (C202/FW) aus 606 Codes, 951 Baubarkeits-, 202 Zusteuerungs- und 16773 Teileauswahlregeln; um alle theoretisch möglichen Aufträge zu betrachten, müsste man also zwischen mehr als $2 \cdot 10^{182}$ Varianten unterscheiden. Die Regeln der Dokumentation sind darüberhinaus einer hohen Änderungsrate unterworfen, da selbst an marktreifen Produkten fortwährend Weiterentwicklungen betrieben werden, die zu zusätzlich möglich werdenden Bestelloptionen oder einer Lockerung bestehender technischer Einschränkungen führen können. Häufig kommt es auch zu einem Austausch einzelner Komponenten durch weiterentwickelte Nachfolgekomponenten, beispielsweise bei einem Modelljahrwechsel.

Auch Testverfahren, die zur Vermeidung von Dokumentationsfehlern eine nicht unbeträchtliche Rolle spielen, stoßen hier an ihre Grenzen. So können selbst durch eine hohe Anzahl von Tests Fehler nicht zu 100% ausgeschlossen werden; außerdem betrachtet man bei Tests vorwiegend “typische” Aufträge, weniger gebräuchliche fallen unter den Tisch; darüberhinaus ist es zur Prüfung einiger Konsistenzeigenschaften zwingend erforderlich, alle möglichen Aufträge zu betrachten, so z.B. für den oben erwähnten Test auf Überflüssigkeit einer Komponente.

Die Erstellung und Anpassung der Produktdokumentation erfordert eine genaue Kenntnis der Produktstruktur, und selbst wenn dieses Wissen vorhanden ist, können in der Dokumentation des Regelsystems neben den beabsichtigten Eigenschaften unerwünschte Nebeneffekte auftreten und lange Zeit unentdeckt bleiben. Fehler in den Regeln können vielfältige und zum Teil gravierende Auswirkungen haben:

- **Inkorrekte Klassifikation von Aufträgen:** Korrekte Aufträge werden fälschlicherweise als fehlerhaft eingestuft und daher abgelehnt, oder noch schlimmer, fehlerhafte Bestellungen werden als korrekt eingestuft und akzeptiert. Dies kann zu einem überflüssigen Abstimmungsprozess zwischen Händler bzw. Kunde und Werk führen.
- **Überflüssige Teile in der Stückliste:** Komponenten, die von keinem möglichen Auftrag ausgewählt werden können, werden nicht erkannt und deshalb unnötigerweise in der Stückliste geführt und eventuell sogar im Lager vorgehalten.
- **Fehlerhafte Teilebedarfsermittlung:** Dies kann neben Auswirkungen auf die Produktionsplanung und Teilefertigung auch in der Montage zu unnötigen Wartezeiten durch Rückfragen führen, falls z.B. laut Dokumentation an einer Einbauposition mehrere sich ausschließende Teile eingebaut werden sollen.

Durch den Einsatz von Verifikationsmethoden lassen sich bestimmte Fehler in der Dokumentation vermeiden. Dies wird erreicht, indem Konsistenzkriterien aufgestellt werden, von denen dann automatisch überprüft werden kann, ob diese für alle theoretisch möglichen Produktvarianten gelten. Man erreicht dadurch eine im Vergleich zu Testverfahren höhere Aussagekraft, da Verifikation eine umfassende Betrachtung aller Eventualitäten ermöglicht.

Wir werden im folgenden eine Reihe von Eigenschaften herausarbeiten, die für eine “korrekte” Dokumentation erforderlich sind. Diese werden wir weiter in “harte” Korrektheitsanforderungen und “weiche” Validitätseigenschaften unterteilen. Bei ersteren handelt es sich dabei um ohne zusätzliches Produktwissen erkennbar notwendige Eigenschaften der Dokumentation, letztere sind eher ungewöhnliche “Auffälligkeiten”, für die nur mit zusätzlichem Fachwissen die Unterscheidung zwischen Fehlerfall oder beabsichtigtem Spezialfall getroffen werden kann.

Diese beiden Klassen von Verifikations- und Validitätseigenschaften betrachten wir getrennt für die “statische” Dokumentation zu einem fest vorgegebenen Zeitpunkt, wie er typischerweise in den Entwicklungsabteilungen betrachtet wird, und für die “dynamische” Dokumentation, die auch zeitliche Entwicklungen des Produkts, sowie dessen Produktionsbedingungen und Dokumentation mitberücksichtigt, was besonders in der Produktion einen hohen Stellenwert hat.

4.1 Statische Aspekte

4.1.1 Konsistenzkriterien

Unabhängig von den tatsächlichen Produkteigenschaften lassen sich Bedingungen aufstellen, von denen man in einer konsistenten Dokumentation ausgehen kann. So sollte z.B. jede Komponente in mindestens einem gültigen Auftrag vorkommen und jede Bestelloption mit mindestens einer konkreten Produktinstanz verträglich sein. Diese Kriterien nennen wir auch *a priori*-Bedingungen, da sie kein explizites Wissen über das Produkt und dessen interne Struktur – wie bestehende Abhängigkeiten zwischen Komponenten – erfordern.

Wir haben die folgenden Konsistenzbedingungen der Produktdokumentation in DIALOG als relevant empfunden:

1. **Keine unzulässigen Codes:** Als unzulässige Codes bezeichnen wir solche, die in keinem möglichen baubaren Auftrag vorkommen können. Es handelt sich also um Bestelloptionen, deren Auswahl, egal mit welchen anderen Optionen auch immer kombiniert, nie (nach Zusteuerung) in einem gültigen Auftrag resultiert. Die Wahlmöglichkeit einer solchen Bestelloption ist in der Praxis also nicht gegeben.

2. **Konsistenz der Zusteuerung:** Darunter verstehen wir zweierlei: Erstens, dass kein ursprünglich baubarer Auftrag durch die Zusteuerung so abgeändert wird, dass er nicht mehr baubar ist (**Stabilität der**

Zusteuerung). Und zweitens, dass für baubare Aufträge das Ergebnis der Zusteuerung nicht von der (mehr oder weniger zufälligen) Reihenfolge, in der Codes hinzugefügt werden, abhängt (**Reihenfolgeunabhängigkeit der Zusteuerung**). Erstere Konsistenzeigenschaft ist dadurch begründet, dass Hauptaufgabe der Zusteuerung die Vervollständigung nur teilweise spezifizierter Aufträge ist, und ein initial schon korrekter Auftrag nicht zwangsläufig abgeändert werden muss.

3. **Keine überflüssigen Teile:** In der Stückliste sollten nur Teile aufgeführt sein, die auch wirklich von irgendeinem theoretisch möglichen, gültigen Auftrag benötigt werden. Ist dies nicht der Fall, so sind sie überflüssig und können daher auch ohne Schaden aus der Gesamtstückliste entfernt werden.

4. **Keine Mehrdeutigkeiten in der Teileliste:** In einem gültigen Auftrag sollten sich gegenseitig ausschließende Teile nie gleichzeitig ausgewählt sein. Auf Grund der Eigenschaft der DIALOG-Dokumentationsmethode, dass an einer Stücklistenposition sich gegenseitig ausschließende Alternativen, die Positionsvarianten (siehe 2.3.1, Abschnitt über Disambiguierung), angegeben sind, macht es möglich zu prüfen, ob wirklich höchstens eine dieser Varianten gleichzeitig selektierbar ist. Die Verwendung der “kurzen Coderegel” soll diesen Fehler zwar verhindern helfen, kann aber eine Mehrfachauswahl auch nicht gänzlich ausschließen.

Neben diesen “harten”, auf Fehler hindeutenden Konsistenzbedingungen haben wir weitere Eigenschaften identifiziert, die eher einen informativen und synoptischen Charakterzug tragen:

1. **Notwendige Codes:** Dies sind Codes, die nach Zusteuerung und erfolgreicher Baubarkeitsprüfung zwangsläufig in jedem Auftrag vorkommen müssen. Das hat zur Konsequenz, dass für die Teileauswahl (und auch die Produktion) nur Produktvarianten berücksichtigt werden müssen, die den entsprechenden Code enthalten, nicht aber solche, in denen er fehlt.

2. **Codegruppen:** Unter einer Codegruppe verstehen wir eine Menge von Codes, so dass in jedem baubaren Auftrag genau (oder höchstens) ein Code dieser Menge enthalten ist. Ein automatischer Konsistenztest kann nun entweder darin bestehen, solche Gruppen automatische aufzufinden, oder eine vorgegebene Menge von Codes auf die Gruppeneigenschaft hin zu überprüfen.

3. **Mögliche zusätzliche Ausstattungscodes:** Dies sind Ausstattungscodes, die zu einem teilweise spezifizierten baubaren Auftrag unter Erhalt der Baubarkeit noch hinzugefügt werden können. Der teilspezifizierte Auftrag wird dabei als eine aussagenlogische Formeleinschränkung an die Auftragscodes angegeben. Dieser Test kann als Verkaufshilfe Verwendung finden, da er erlaubt, Kunden Erweiterungsvorschläge zu einem bereits teilweise spezifizierten Auftrag zu machen.

Neben den beschriebenen Tests, die alle in unserem Baubarkeits-Informationssystem BIS (siehe Anhang A) implementiert sind, sind natürlich noch weitere denkbar. Insbesondere kann man beliebige manuell codierte (Meta-)Eigenschaften der Produktdokumentation, sofern als PDL-Formel darstellbar, dazunehmen. Diese zusätzlichen Eigenschaften werden aber meist spezielles Produktwissen erfordern, und lassen sich daher nicht so allgemein und systematisch darstellen wie die oben genannten Tests.

4.1.2 Formalisierung

Wir wollen uns nun der Formalisierung der im vorangehenden Abschnitt genannten Tests zuwenden. Grundlage ist dabei obige Formalisierung des Auftragsverarbeitungssystems in PDL (Abbildung 3.6). Auf die drei Teile dieses Programms (Zusteuerung, Baubarkeitsprüfung, Teilebedarfsermittlung) nehmen wir, wie oben bereits erwähnt, mittels Z, B und T Bezug. Damit können wir bereits eine Formalisierung der einzelnen Konsistenzkriterien vornehmen.

Unzulässige Codes: Obige intuitive Formulierung eines unzulässigen Codes lässt sich wie folgt fassen:

$$\text{Falls } \mathcal{K}^{\mathcal{S}}_{\Box} \models x \Rightarrow \neg\langle Z; B\rangle\top\text{, so ist Code } x \text{ unzulässig.} \tag{4.1}$$

Damit ist ausgedrückt, dass unter der Vorbedingung x, d.h. falls Code x im Auftrag vorkommt, kein terminierender Programmlauf des Zusteuerungs- und Baubarkeitsprüfprogramms existiert. Da es für jeden Startzustand einen terminierenden Lauf des Zusteuerungsprogramms gibt (Lemma 3.3.1), kann dies nur durch ein Fehlschlagen der Baubarkeitsprüfung verursacht worden sein.

Eine alternative Formulierung der Unzulässigkeit von Code x ist durch die Gültigkeit von

$$\mathcal{K}^{\mathcal{S}}_{\Box} \models [Z; B]\neg x \tag{4.2}$$

gegeben. In dieser Fassung ist Code x unzulässig, wenn alle (erfolgreich) terminierenden Programmläufe zu einer Situation führen, in der $\neg x$ gilt, wenn Code x also nach Zusteuerung und erfolgreicher Baubarkeitsprüfung in keinem Auftrag mehr vorkommt.

Letztere Sicht ist eher die der Produktion, für die die Teilebedarfsermittlung im Vordergrund steht, erstere die des Vertriebs bzw. des Kunden, für die der initiale Auftrag relevant ist.

Weiter unten untersuchen wir ausführlich die Zusammenhänge zwischen den beiden Varianten. Dabei stellt sich heraus, dass ein unzulässiger Code aus ‘Produktionssicht’ (4.2) immer auch aus ‘Kundensicht’ (4.1) ein unzulässiger Code ist. Die Umkehrung muss nicht zwangsläufig gelten, so dass es aus ‘Kundensicht’ mehr unzulässige Codes geben kann.

Notwendige Codes: Diese sind aufgrund ihrer Ähnlichkeit zu den unzulässigen Codes auch ähnlich formalisierbar:

$$\text{Falls } \mathcal{K}^{\mathcal{S}}_{\Box} \models \neg x \Rightarrow \neg\langle Z; B\rangle\top\text{, so ist Code } x \text{ notwendig.} \tag{4.3}$$

Schlagen also für jeden Kundenauftrag, der x nicht enthält, die ersten beiden Teile (Z und B) der Auftragsbearbeitung fehl, so ist Code x notwendig, d.h. er muss zwangsweise vom Kunden angegeben werden, um einen baubaren Auftrag zu erhalten.

Auch hier gibt es wieder eine zweite Formalisierungsmöglichkeit, wonach Code x notwendig ist, falls

$$\mathcal{K}^{\mathcal{S}}_{\Box} \models [Z; B]x \ . \tag{4.4}$$

Damit wird ausgedrückt, dass ein notwendiger Code nach jedem erfolgreichen Durchlauf des Zusteuerungs- und Prüfprogramms zwangsläufig im Auftrag auftreten muss.

Wieder ist die zweite Sicht der Produktion näher, die erste dem Kunden bzw. Vertrieb. Und auch hier besteht eine Abhängigkeit zwischen den beiden Formalisierungsvarianten, die durch Lemma 4.1.1 beschrieben ist: Ein notwendiger Code aus ‘Kundensicht’ ist immer auch ein notwendiger Code aus ‘Produktionssicht’.

Stabilität der Zusteuerung: Wie oben erwähnt, verstehen wir darunter, dass ein ursprünglich baubarer Auftrag auch nach der Zusteuerung noch baubar ist. Dies lässt sich als Gültigkeit der Formel

$$\mathcal{K}^{\mathcal{S}}_{\Box} \models \langle B\rangle\top \Rightarrow [Z]\langle B\rangle\top \tag{4.5}$$

ausdrücken: Wenn also ein nicht-fehlschlagender Programmlauf der Baubarkeitsprüfung existiert, so gilt dies auch noch nach allen möglichen Zusteuerungsläufen. Verschiedene Zusteuerungsläufe sind möglich, da das Zusteuerungsprogramm nicht-deterministisch den nächsten zuzusteuernden Code auswählt.

Reihenfolgeunabhängigkeit der Zusteuerung: Verursacht durch die Unbestimmtheit der Auswahl des als nächstes zuzusteuernden Codes kann es zu Mehrdeutigkeiten in der Zusteuerung kommen. Sobald für zwei Zusteuerungsformeln Z_i und Z_j mit $i \neq j$ deren Konjunktion $Z_i \wedge Z_j$ erfüllbar ist, können beide Regeln in ein und derselben Situation ausgeführt werden, was zunächst einmal zu unterschiedlichen zugesteuerten Aufträgen führt. Ob sich dieser Unterschied später wieder aufhebt (durch Zusteuerung des jeweils anderen Codes) oder gar irrelevant ist (weil die zugesteuerten Aufträge sowieso nicht baubar sind) kann sich eventuell erst zu einem späteren Zeitpunkt im Ablauf des Auftragsbearbeitungsprogramms zeigen.

Die Unabhängigkeit des Ergebnisses von der Zusteuerungsreihenfolge kann in PDL ausgedrückt werden als Gültigkeit der Formel

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \langle Z; B \rangle p \Leftrightarrow [Z; B] p \tag{4.6}$$

für beliebige atomare Aussagen p. Damit wird sogar gefordert, dass Zusteuerungs- und Baubarkeitsprüfprogramm unabhängig vom jeweiligen Programmlauf immer in demselben Zustand enden, das Zusteuerungsprogramm also einen funktionalen Zusammenhang zwischen dem initialen und dem baubaren zugesteuerten Auftrag herstellt, und demnach insgesamt deterministisch ist (was prinzipiell eine sinnvolle Anforderung an die Zusteuerung sein kann).

Eine alternative Formulierung der Reihenfolgeunabhängigkeit, zunächst einmal ohne Berücksichtigung der Baubarkeit, besteht in der Forderung, dass die Transitionsrelation des Zusteuerungsprogramms konfluent sein muss.[1] Dazu betrachten wir die Einzelschritt-Zusteuerungsprogramme $E_i = (Z_i \wedge \neg x_i)?; x_i := \mathbf{true}$ und deren nicht-deterministische Auswahl $E = \bigcup_{1 \leq i \leq n} E_i$. Mit diesen Definitionen ist die Konfluenz der Zusteuerung äquivalent zur Gültigkeit von

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \langle E^* \rangle [E^*] p \Rightarrow [E^*] \langle E^* \rangle p \tag{4.7}$$

---

[1] Eine Relation R heißt konfluent, falls aus R^*xy und R^*xz folgt, dass es ein u gibt mit R^*yu und R^*zu, wobei R^* der reflexiv-transitive Abschluss von R ist.

für alle $p \in \Phi_0$. Da Z darüberhinaus die Terminationseigenschaft besitzt, kann man sich auf die Forderung nach lokaler Konfluenz, d.h. auf die Gültigkeit von

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \langle E \rangle [E^*] p \Rightarrow [E] \langle E^* \rangle p \tag{4.8}$$

für alle atomaren Aussagen p beschränken [New42]. Möchte man die Baubarkeit noch mit berücksichtigen und die Konfluenz nur für komplett zugesteuerte Zustände fordern, so ergibt sich

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \langle E \rangle [E^*] (p \wedge \langle B \rangle \top \wedge \bar{Z}) \Rightarrow [E] \langle E^* \rangle p \tag{4.9}$$

für alle $p \in \Phi_0$, wobei wir mit $\bar{Z} := \neg(Z_1 \wedge \neg x_1) \wedge \cdots \wedge \neg(Z_n \wedge x_n) = (Z_1 \Rightarrow x_1) \wedge \cdots \wedge (Z_n \Rightarrow x_n)$ die Terminationsbedingung der Schleife des Zusteuerungsprogramms abkürzen.

Wir werden weiter unten noch eine relativ allgemeine Anmerkung zur einschränkenden Wirkung von PDL-Formeln auf Transitionsrelationen von Programmen machen. Diese kann auf den speziellen Fall der Konfluenz angewandt werden und liefert dann eine weitere Begründung der gewählten Formalisierung.

Überflüssige Teile: Ein Teil ist überflüssig, wenn es von keinem möglichen baubaren Auftrag ausgewählt wird, falls also

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \neg \langle Z; B; T \rangle b_i \tag{4.10}$$

für die dem Teil t_i zugeordnete Variable b_i gilt. Ist diese Bedingung erfüllt, so gibt es keinen Programmlauf der Auftragsbearbeitung, nach dem b_i gilt, Teil t_i also in die Stückliste des Auftrags mit aufgenommen wird. Auf Grund der eindeutigen Belegung der Variablen b_i abhängig von Formel T_i des Teilebedarfsermittlungsprogramms ist dies äquivalent zu

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \neg \langle Z; B \rangle T_i, \tag{4.11}$$

wie nachfolgendes Lemma 4.1.2 zeigt. Letztere Formulierung hat den Vorteil, dass man auf die Berücksichtigung sämtlicher Variablen b_i verzichten kann.

Mehrdeutigkeiten in der Teileliste: Falls für zwei sich gegenseitig ausschließende Teile t_i und t_j, repräsentiert durch die beiden Programmvariablen b_i und b_j,

$$\langle Z; B; T \rangle (b_i \wedge b_j) \tag{4.12}$$

erfüllbar in $\mathcal{K}^{\mathcal{S}}_{\Box}$ ist, so können diese gleichzeitig von einem Auftrag ausgewählt werden, es handelt sich um eine Mehrdeutigkeit in der Stückliste. Auch dies lässt sich, analog dem vorhergehenden Test, reduzieren zur Erfüllbarkeit von

$$\langle Z; B \rangle (T_i \wedge T_j) \tag{4.13}$$

in $\mathcal{K}^{\mathcal{S}}_{\Box}$. Wiederum kann man dadurch auf die teilerepräsentierenden Booleschen Variablen b_i verzichten.

Mögliche Codes: Ein Code x wird möglicher Code genannt, wenn er einem Auftrag hinzugefügt werden kann, so dass dieser weiterhin baubar bleibt. Der schon bestehende, teilspezifizierte Auftrag wird über eine aussagenlogische Formelbedingung A angegeben, die die vom Kunden bereits gemachten Einschränkungen enthält. Unter diesen Voraussetzungen ist dann Code x möglich, wenn

$$A \wedge x \Rightarrow \langle Z; B \rangle \top \tag{4.14}$$

erfüllbar in $\mathcal{K}^{\mathcal{S}}_{\Box}$ ist. Wir werden weiter unten noch Genaueres zu den Formeleinschränkungen sagen.

Codegruppen: Wir beschreiben hier nur den Test, ob es sich bei einer gegebenen Menge von Codes um eine Codegruppe handelt, und nicht die Möglichkeit, solche Gruppen automatisch zu erkennen. Dazu bezeichen wir, wie schon in Abschnitt 3.1.5, mit S^b_a den Selektionsoperator. Dieser n-stellige Operator (für variables n) ist in einem Ausdruck $\mathrm{S}^b_a(F_1, \ldots, F_n)$ genau dann wahr, wenn zwischen a und b der Formeln F_i wahr sind. Unter Verwendung des Selektionsoperators handelt es sich bei einer Menge $G = \{g_1, \ldots, g_k\} \subseteq \mathcal{C}$ dann um eine Codegruppe, falls

$$\mathcal{K}^{\mathcal{S}}_{\Box} \models [Z; B] \mathrm{S}^1_a(g_1, \ldots, g_k), \tag{4.15}$$

wobei $a \in \{0, 1\}$ und $a = 0$, falls höchstens einer der Codes aus der Gruppe in einem Auftrag vorkommen darf, sowie $a = 1$, falls genau ein Code in jedem Auftrag vorkommen muss.

Neben der direkten Anwendung dieser Konsistenzbedingungen auf die Menge aller möglichen Aufträge, tritt oft auch eine erweiterte Fragestellung auf, in der die Ausgangsmenge der betrachteten Aufträge reduziert bzw. eingeschränkt ist. Man kann so beispielsweise regionale Einschränkungen machen, um die Tests nur für Fahrzeuge, die in ein bestimmtes Land geliefert werden, durchzuführen. Oder man kann sich auf solche Fahrzeugvarianten beschränken, die in einem bestimmten Werk produziert werden.

Diese Einschränkungen können als Formelbedingungen formuliert werden, die die zulässigen Kombinationen von Codes einschränken. So kann man durch die Einschränkung $x \vee y$ nur solche Aufträge betrachten, in denen mindestens einer der Codes x und y vorkommt.

Einschränkungen können dabei an zwei Stellen im Auftragsbearbeitungsprozess von Nutzen sein: (a) als Einschränkung der betrachteten (initialen) Kundenaufträge oder (b) als Einschränkung nach Zusteuerung und Baubarkeitsprüfung, also direkt vor der Teilebedarfsermittlung. Nehmen wir an, dass die so einschränkenden Formeln mit A_1 und A_2 bezeichnet sind, so ergibt sich für die einfache Frage nach der Existenz einer gültigen Fahrzeugkonfiguration mit diesen Einschränkungen die PDL-Übersetzung:

$$\text{Ist } A_1 \wedge \langle Z; B \rangle A_2 \text{ erfüllbar in } \mathcal{K}_{\Box}^{\mathcal{S}}? \tag{4.16}$$

Wir werden die Behandlung dieses erweiterten Tests nicht weiter vertiefen. Stattdessen werden wir die oben nur informell begründeten Eigenschaften des Auftragsbearbeitungsprogramms nun durch die nachfolgenden Lemmas exakt nachweisen.

Lemma 4.1.1 *In der regulären Standardstruktur $\mathcal{K}_{\Box}^{\mathcal{S}}$ gilt für das Zusteuerungsprogramm Z (mit enthaltenen Zusteuerungsformeln Z_i), das Baubarkeitsprüfprogramm B und alle $x_i \in \mathcal{C}$:*

1. *Wenn $[Z; B]\neg x_i$, so gilt auch $x_i \Rightarrow \neg\langle Z; B \rangle \top$.*
2. *Falls $Z_i \equiv \bot$, so gilt $[Z; B]\neg x_i$ genau dann, wenn $x_i \Rightarrow \neg\langle Z; B \rangle \top$.*
3. *Wenn $\neg x_i \Rightarrow \neg\langle Z; B \rangle \top$, so gilt auch $[Z; B]x_i$.*
4. *Falls $Z_i \equiv \bot$, so gilt $\neg x_i \Rightarrow \neg\langle Z; B \rangle \top$ genau dann, wenn $[Z; B]x_i$.*

Beweis:

1. Angenommen, es gilt $[Z; B]\neg x_i$. Dann gilt $(\forall s, t)((s, t) \in \tau(Z; B) \Rightarrow t \notin \sigma(x_i))$ nach Def. von σ. Aus Lemma 3.3.2 wissen wir, dass $x_i \Rightarrow [Z; B]x_i$, also $(\forall s)(s \in \sigma(x_i) \Rightarrow (\forall t)((s, t) \in \tau(Z; B) \Rightarrow t \in \sigma(x_i)))$. Insgesamt ergibt sich also $(\forall s)(s \in \sigma(x_i) \Rightarrow (\forall t)((s, t) \in \tau(Z; B) \Rightarrow \bot))$ oder $(\forall s)(s \in \sigma(x_i) \Rightarrow (\forall t)(s, t) \notin \tau(Z; B))$.

2. Die Implikation von links nach rechts folgt aus dem ersten Teil des Lemmas. Für die andere Richtung nehmen wir an, dass $x_i \Rightarrow \neg\langle Z; B\rangle\top$ gilt, und damit nach Def. von σ auch $(\forall s)(s \in \sigma(x_i) \Rightarrow (\forall t)(s,t) \notin \tau(Z;B))$, oder andersherum $(\forall s)((\exists t)(s,t) \in \tau(Z;B) \Rightarrow s \notin \sigma(x_i))$. Da wegen $Z_i \equiv \bot$ nach Lemma 3.3.2 außerdem $\langle Z; B\rangle x_i \Rightarrow x_i$, gilt auch $(\forall s)(((\exists t)((s,t) \in \tau(Z;B) \wedge t \in \sigma(x_i))) \Rightarrow s \in \sigma(x_i))$. Nehmen wir nun beliebige s, t mit $(s,t) \in \tau(Z;B)$. Dann haben wir nach Voraussetzung $s \notin \sigma(x_i)$ und nach Lemma 3.3.2 $t \notin \sigma(x_i)$, da ansonsten ein Widerspruch auftritt. Damit haben wir $(\forall s,t)((s,t) \in \tau(Z;B) \Rightarrow t \notin \sigma(x_i)))$ und somit $[Z;B]\neg x_i$ gezeigt.

3. Angenommen, es gilt $\neg x_i \Rightarrow [Z;B]\bot$. Dann gilt $(\forall s)(s \notin \sigma(x_i) \Rightarrow (\forall t)(s,t) \notin \tau(Z;B))$ nach Def. von σ. Lemma 3.3.2 liefert $(\forall s)(s \in \sigma(x_i) \Rightarrow (\forall t)((s,t) \in \tau(Z;B) \Rightarrow t \in \sigma(x_i)))$, so dass sich insgesamt $(\forall s,t)((s,t) \in \tau(Z;B) \Rightarrow t \in \sigma(x_i)))$ und damit $[Z;B]x_i$ ergibt.

4. Nehmen wir an, dass $[Z;B]x_i$, dass also $(\forall s,t)((s,t) \in \tau(Z;B) \Rightarrow t \in \sigma(x_i))$. Wählen wir nun irgendein s (sofern vorhanden), für das es ein t gibt mit $(s,t) \in \tau(Z;B)$. Dann gilt $t \in \sigma(x_i)$ nach Voraussetzung. Nach Lemma 3.3.2 gilt auch $s \in \sigma(x_i)$, so dass wir $((\exists t)(s,t) \in \tau(Z;B)) \Rightarrow s \in \sigma(x_i)$ für alle s erhalten. Dies ist äquivalent zu $s \notin \sigma(x_i) \Rightarrow (\forall t)(s,t) \notin \tau(Z;B))$, also $\neg x_i \Rightarrow [Z;B]\bot$. Zusammen mit dem 3. Teil des Lemmas folgt die Behauptung.

□

Das nächste Lemma erlaubt die Vereinfachung von Aussagen über die Teilebedarfsermittlung:

Lemma 4.1.2 *In der regulären Standardstruktur $\mathcal{K}^{\mathcal{S}}_{\Box}$ gilt für das Teilebedarfsermittlungsprogramm T (mit enthaltenen Selektionsformeln T_i) und alle teilerepräsentierenden Variablen b_i:*

$$\langle T\rangle b_i \;\Leftrightarrow\; T_i$$

Beweis: Zuerst wollen wir zeigen, dass

$$\langle \textbf{if } T_j \textbf{ then } b_j := \textbf{true} \textbf{ else } b_j := \textbf{false}\rangle F \;\Leftrightarrow\; F, \qquad (*)$$

falls in F die Variable b_j nicht vorkommt. Außerdem zeigen wir, dass

$$\langle \textbf{if } T_i \textbf{ then } b_i := \textbf{true} \textbf{ else } b_i := \textbf{false}\rangle b_i \;\Leftrightarrow\; T_i\ . \qquad (**)$$

Wie in Abschnitt 3.3 angemerkt, enthalten die Formeln T_j keine Teilevariablen b_i, sondern nur Codevariablen x_i. Dies, zusammen mit den beiden vorhergehenden Aussagen, begründet dann die Behauptung, denn es gilt unter Verwendung der Abkürzung α_i für das Programm **if** T_i **then** $b_i :=$ **true else** $b_i :=$ **false**:

$$\begin{aligned}
\langle T\rangle b_i &\Leftrightarrow \langle \alpha_1; \ldots; \alpha_i; \ldots; \alpha_k\rangle b_i \\
&\Leftrightarrow \langle \alpha_1\rangle \ldots \langle \alpha_i\rangle \ldots \langle \alpha_k\rangle b_i \\
&\Leftrightarrow \langle \alpha_1\rangle \ldots \langle \alpha_i\rangle b_i && \text{wegen Behauptung (*)} \\
&\Leftrightarrow \langle \alpha_1\rangle \ldots \langle \alpha_{i-1}\rangle T_i && \text{wegen Behauptung (**)} \\
&\Leftrightarrow T_i && \text{wegen Behauptung (*)}
\end{aligned}$$

Wir beweisen nun die beiden Behauptungen. Es gilt

$$\begin{aligned}
&\langle \mathbf{if}\ T_j\ \mathbf{then}\ b_j := \mathbf{true}\ \mathbf{else}\ b_j := \mathbf{false}\rangle F \\
\Leftrightarrow\ &\langle (T_j?; b_j := \mathbf{true}) \cup (\neg T_j?; b_j := \mathbf{false})\rangle F && \text{nach Def. von } \mathbf{if\text{-}then\text{-}else} \\
\Leftrightarrow\ &\langle T_j?; b_j := \mathbf{true}\rangle F \vee \langle \neg T_j?; b_j := \mathbf{false}\rangle F && \text{nach Axiom (iv)} \\
\Leftrightarrow\ &\langle T_j?\rangle\langle b_j := \mathbf{true}\rangle F \vee \langle \neg T_j?\rangle\langle b_j := \mathbf{false}\rangle F && \text{nach Axiom (v)} \\
\Leftrightarrow\ &\langle T_j?\rangle F \vee \langle \neg T_j?\rangle F && \text{nach Def. 3.2.10, 1. und 3.} \\
\Leftrightarrow\ &(T_j \wedge F) \vee (\neg T_j \wedge F) && \text{nach Axiom (vi)} \\
\Leftrightarrow\ &F,
\end{aligned}$$

also auch Behauptung (*). Außerdem ist mit analogen Begründungen

$$\begin{aligned}
&\langle \mathbf{if}\ T_i\ \mathbf{then}\ b_i := \mathbf{true}\ \mathbf{else}\ b_i := \mathbf{false}\rangle b_i \\
\Leftrightarrow\ &\langle T_i?\rangle\langle b_i := \mathbf{true}\rangle b_i \vee \langle \neg T_i?\rangle\langle b_i := \mathbf{false}\rangle b_i \\
\Leftrightarrow\ &\langle T_i?\rangle\top \vee \langle \neg T_i?\rangle\bot && \text{nach Def. 3.2.10, 1. und 2.} \\
\Leftrightarrow\ &T_i,
\end{aligned}$$

und daher gilt auch Behauptung (**). □

Die Formalisierung der Reihenfolgeunabhängigkeit der Zusteuerung kann auf den ersten Blick überraschend und intuitiv wenig verständlich sein. Wir wollen deshalb hier eine ausführlichere Begründung, die sich auf Korrespondenztheorie [vB84] beruft, nachreichen. Die Korrespondenztheorie beschäftigt sich mit der Frage, welche Auswirkungen modale Axiome auf die Sichtbarkeitsrelation der Kripke-Struktur haben. So ist beispielsweise das modale Axiom $\Box p \Rightarrow p$ genau in den Kripke-Strukturen mit reflexiver Sichtbarkeitsrelation gültig.

Wir schauen uns nun die Auswirkungen der Formel 4.7,

$$\langle E^*\rangle[E^*]p \Rightarrow [E^*]\langle E^*\rangle p,$$

auf die Sichtbarkeits- oder Transitionsrelation R_E des Programmes E an. Weiter oben haben wir behauptet, dass mit dieser Formel die Konfluenz der Relation R_E sichergestellt wird; dies wollen wir jetzt mit Methoden der Korrespondenztheorie untermauern.

Durch die Übersetzung des PDL-Ausdrucks nach Prädikatenlogik erhält man unter Verwendung von R_E^* und dem Prädikat P die Formel zweiter Ordnung

$$(\forall P)(\forall x)((\exists y)(R_E^* xy \wedge (\forall z)(R_E^* yz \Rightarrow Pz)) \Rightarrow \\ (\forall y)(R_E^* xy \Rightarrow (\exists z)(R_E^* yz \wedge Pz))).$$

In dieser Formel ist die Quantifizierung zweiter Ordnung über P deshalb vorhanden, weil die ursprüngliche Aussage für alle Prädikate p gelten soll. Umformung liefert das äquivalente

$$(\forall x,y)(\forall P)(R_E^* xy \wedge (\forall z)(R_E^* yz \Rightarrow Pz) \Rightarrow \\ (\forall z)(R_E^* xz \Rightarrow (\exists u)(R_E^* zu \wedge Pu))).$$

Falls der Antezedent in dieser Formel falsch ist, so gilt die Formel trivialerweise, so dass wir das Prädikat P so wählen, dass wir den stärkstmöglichen Antezedenten erhalten. Für $Px = R_E^* yx$ ist dies der Fall. Damit ergibt sich:

$$(\forall x,y)(R_E^* xy \Rightarrow (\forall z)(R_E^* xz \Rightarrow (\exists u)(R_E^* zu \wedge R_E^* yu)),$$

was äquivalent ist zu

$$(\forall x,y,z)(R_E^* xy \wedge R_E^* xz \Rightarrow (\exists u)(R_E^* zu \wedge R_E^* yu)),$$

der Aussage, dass die Relation R_E konfluent ist.

4.1.3 Konvertierung nach Aussagenlogik

Wir wenden uns nun der Frage zu, wie die im letzten Abschnitt generierten Konsistenzeigenschaften automatisch überprüft werden können. Neben einer direkten Erfüllbarkeits- oder Gültigkeitsprüfung der Formeln steht auch die in Abschnitt 3.4 entwickelte Transformation nach Aussagenlogik zur

Verfügung. Diese wollen wir nun auf die Entscheidungsprobleme, soweit möglich, anwenden.

Die meisten Konsistenztests – unzulässige Codes (4.2), notwendige Codes (4.4), überflüssige Teile (4.11), Mehrdeutigkeiten in der Teileliste (4.13), Codegruppen (4.15) – lassen sich als PDL-Erfüllbarkeitsprobleme der Form

$$\langle Z; B\rangle F \tag{4.17}$$

für eine aussagenlogische Testformel F und die Standardstruktur $\mathcal{K}^{\mathcal{S}}_{\Box}$ schreiben. Des weiteren können die iterativen *guarded commands* äquivalent auch als **while**-Schleifen dargestellt werden, wobei das Programm **do** $F_1 \rightarrow \alpha_1 | \ldots | F_n \rightarrow \alpha_n$ **od** äquivalent ist zu

$$\textbf{while } F_1 \vee \cdots \vee F_n \textbf{ do if } F_1 \;\rightarrow\; \alpha_1 \mid \;\cdots\; \mid F_n \;\rightarrow\; \alpha_n \textbf{ fi } . \tag{4.18}$$

Wendet man nun Lemma 3.4.6(3) auf diese Schleife an, so erhält man

$$\langle \textbf{do } F_1 \rightarrow \alpha_1 | \ldots | F_n \rightarrow \alpha_n \textbf{ od}\rangle G \overset{\text{ERF}}{\equiv}_{\mathcal{K}} \langle (\neg F_1 \wedge \cdots \wedge \neg F_n)?\rangle G$$

für beliebige Kripke-Strukturen $\mathcal{K}$. Spezialisierung dieser Äquivalenz auf das Zusteuerungsprogramm Z erlaubt somit eine Vereinfachung des Erfüllbarkeitstest 4.17:

$$\begin{aligned}
\langle Z; B\rangle F &\equiv \quad \langle Z\rangle\langle B\rangle F \\
&\overset{\text{ERF}}{\equiv}_{\mathcal{K}^{\mathcal{S}}_{\Box}} \langle (\neg(Z_1 \wedge \neg x_1) \wedge \cdots \wedge \neg(Z_n \wedge \neg x_n))?\rangle\langle B\rangle F \\
&\equiv \quad \neg(Z_1 \wedge \neg x_1) \wedge \cdots \wedge \neg(Z_n \wedge \neg x_n) \;\wedge\; \langle B\rangle F \\
&\equiv \quad (Z_1 \Rightarrow x_1) \wedge \cdots \wedge (Z_n \Rightarrow x_n) \;\wedge\; \langle B\rangle F
\end{aligned}$$

Anwendung von Lemma 3.3.3 erlaubt eine weitere Vereinfachung zu

$$(Z_1 \Rightarrow x_1) \wedge \cdots \wedge (Z_n \Rightarrow x_n) \wedge \; (x_1 \Rightarrow B_1) \wedge \cdots \wedge (x_n \Rightarrow B_n) \wedge \; F, \tag{4.19}$$

so dass wir nun den PDL-Erfüllbarkeitstest aus Formel 4.17 durch den aussagenlogischen Erfüllbarkeitstest 4.19 ersetzen können. Damit ist diese Sorte von Erfüllbarkeitstests gänzlich ohne Zugriff auf einen Zustandsbegriff durchführbar, womit sich eine erhebliche Vereinfachung des Entscheidungsproblems und die Möglichkeit fortgeschrittene SAT-Checker direkt einzusetzen ergibt.

Die drei verbleibenden Konsistenztests – Stabilität der Zusteuerung (4.5), Reihenfolgeunabhängigkeit der Zusteuerung (4.8), mögliche Codes (4.14) – lassen sich nicht als Erfüllbarkeitsprobleme der Form $\langle Z; B\rangle F$ darstellen und müssen daher mit anderen Mitteln behandelt werden.

Stabilität der Zusteuerung. Für die Stabilität der Zusteuerung betrachten wir eine stärkere, aber automatischen Beweisverfahren besser zugängliche Bedingung als die durch 4.5 gegebene, indem wir für jeden einzelnen Zusteuerungsschritt den Erhalt der Baubarkeit fordern. Unter Verwendung der oben definierten Einzelschritt-Zusteuerungsprogramme E_i ergibt sich damit als Konsistenzbedingung die Gültigkeit der Formel

$$\mathcal{K}^{\mathcal{S}}_{\Box} \models \langle B\rangle\top \Rightarrow [E_i]\langle B\rangle\top \tag{4.20}$$

für alle $i \in \{1, \ldots, n\}$. Falls diese Bedingung erfüllt ist, so folgt daraus, wie man leicht per Induktion zeigen kann, auch die Gültigkeit der ursprünglichen Stabilitätsbedingung 4.5. Die Gültigkeit der Formel 4.20 ist äquivalent zur Unerfüllbarkeit von $\neg(\langle B\rangle\top \Rightarrow [E_i]\langle B\rangle\top)$ in $\mathcal{K}^{\mathcal{S}}_{\Box}$, was wiederum äquivalent ist zur Unerfüllbarkeit in $\mathcal{K}^{\mathcal{S}}_{\Box}$ von $\langle B\rangle\top \wedge \langle E_i\rangle[B]\bot$. Diese Formel lässt sich mittels der (erfüllbarkeitserhaltenden) Reduktionsregeln des Termersetzungssystems $\mathcal{R}$ und unter Verwendung der Äquivalenz aus Lemma 3.3.3 umformen:

$$\begin{aligned}
& \langle B\rangle\top \wedge \langle E_i\rangle[B]\bot \\
\equiv\quad & \langle B\rangle\top \wedge \langle (Z_i \wedge \neg x_i)?\rangle\langle x_i := \mathbf{true}\rangle[B]\bot \\
\equiv\quad & \langle B\rangle\top \wedge \langle (Z_i \wedge \neg x_i)?\rangle\langle x_i := \mathbf{true}\rangle\neg((x_1 \Rightarrow B_1) \wedge \cdots \wedge (x_n \Rightarrow B_n)) \\
\leadsto^{+}\quad & \langle B\rangle\top \wedge (Z_i \wedge \neg x_i \wedge \neg((x_1 \Rightarrow B_1) \wedge \cdots \wedge (x_n \Rightarrow B_n))|_{x_i=\mathbf{true}}) \\
\equiv\quad & \bigwedge_{1\le k\le n} (x_k \Rightarrow B_k) \wedge (Z_i \wedge \neg x_i \wedge \neg B_i|_{x_i=\mathbf{true}} \wedge \bigvee_{1\le j\le n, j\ne i} (x_j \wedge \neg B_j|_{x_i=\mathbf{true}}))
\end{aligned}$$

Die Formel der letzten Zeile lässt sich noch verkürzen, indem man von der Eigenschaft $F \Rightarrow F|_{x_i=\mathbf{true}}$ Gebrauch macht, die gilt, falls die Variable x_i nicht negativ in F vorkommt (abgekürzt mit $x_i \notin \mathrm{negocc}(F)$) . Man erhält somit

$$\bigwedge_{1\le k\le n} (x_k \Rightarrow B_k) \wedge \left(Z_i \wedge \neg x_i \wedge \neg B_i|_{x_i=\mathbf{true}} \wedge \bigvee_{\substack{1\le j\le n, j\ne i \\ x_i \notin \mathrm{negocc}(B_j)}} (x_j \wedge \neg B_j|_{x_i=\mathbf{true}}) \right) \tag{4.21}$$

als zu prüfende Kondition. Ist diese Formel unerfüllbar für alle i, so folgt daraus die Stabilität der Zusteuerung.

Reihenfolgeunabhängigkeit der Zusteuerung. Ausgangspunkt unserer Untersuchungen ist hier die Formalisierung der lokalen Konfluenz mittels Formel 4.8. Zur einfacheren Codierung in Aussagenlogik und zur Vermeidung der ansonsten für die Iteration erforderlichen Fixpunktberechnung verwenden wir eine schärfere Form dieser Bedingung, die wir Einschritt-Konfluenz nennen wollen und die dem von van Benthem verwendeten Begriff der “directedness” nahe kommt [vB84] (siehe dazu auch [KS00]). Bei dieser Verschärfung müssen sich zwei divergente Zustände in höchstens einem Schritt (und nicht in beliebig vielen wie bei der lokalen Konfluenz) wieder zusammenführen lassen. Man erhält hierfür als PDL-Äquivalent die Formel

$$\mathcal{K}_{\Box}^{\mathcal{S}} \models \langle E\rangle[E^{\leq 1}]p \Rightarrow [E]\langle E^{\leq 1}\rangle p \tag{4.22}$$

mit $E^{\leq 1} \equiv \textbf{skip} \cup E$, deren Gültigkeit die Einschritt-Konfluenz beinhaltet. Das Programm $E = \bigcup_{1\leq i\leq n} E_i$ setzt sich, wie oben angemerkt, aus elementaren Einzelschritt-Zusteuerungsprogrammen zusammen. Somit ergibt sich der komplexere Ausdruck

$$\left\langle \bigcup_{1\leq i\leq n} E_i \right\rangle \left[\textbf{skip} \cup \bigcup_{1\leq k\leq n} E_k\right] p \Rightarrow \left[\bigcup_{1\leq j\leq n} E_j\right] \left\langle \textbf{skip} \cup \bigcup_{1\leq l\leq n} E_l \right\rangle p$$

für die Formel aus Eigenschaft 4.22, der nach Lemma 3.2.2(4) äquivalent ist zu

$$\left(\bigvee_{1\leq i\leq n} \langle E_i\rangle \left(p \wedge \bigwedge_{1\leq k\leq n} [E_k]p\right)\right) \Rightarrow \bigwedge_{1\leq j\leq n} [E_j]\left(p \vee \bigvee_{1\leq l\leq n} \langle E_l\rangle p\right) . \tag{†}$$

Nehmen wir nun an, dass

$$\langle E_{i^*}\rangle[E_{j^*}]p \Rightarrow [E_{j^*}]\langle E_{i^*}\rangle p \quad \text{und} \quad \langle E_{i^*}\rangle p \Rightarrow [E_{i^*}]p \tag{††}$$

für alle i^*, j^* mit $i^* \neq j^*$ gelte. Wir wollen zeigen, dass unter dieser Voraussetzung bereits (†) gilt: Für den Fall $i = j$ erhalten wir unter Verwendung von (††) die Implikationsfolge

$$\langle E_i\rangle p \wedge \bigwedge_{1\leq k\leq n} \langle E_i\rangle[E_k]p \;\Rightarrow\; \langle E_i\rangle p \;\Rightarrow\; [E_i]p \;\Rightarrow\; [E_j]p \vee \bigvee_{1\leq l\leq n} [E_j]\langle E_l\rangle p \;.$$

Für $i \neq j$ können wir wegen des ersten Teils von (††) dassselbe Resultat über folgende Schlusskette erreichen:

$$\begin{aligned}\langle E_i\rangle p \wedge \bigwedge_{1\leq k\leq n} \langle E_i\rangle[E_k]p \;&\Rightarrow\; \langle E_i\rangle[E_j]p \\ &\Rightarrow\; [E_j]\langle E_i\rangle p \;\Rightarrow\; [E_j]p \vee \bigvee_{1\leq l\leq n} [E_j]\langle E_l\rangle p \;.\end{aligned}$$

Damit erhalten wir unter Verwendung von Lemma 3.2.2(1) für beliebige $i, j \in \{1, \ldots, n\}$:

$$\begin{aligned}\langle E_i\rangle\Bigl(p \wedge \bigwedge_{1\le k\le n} [E_k]p\Bigr) &\Rightarrow \langle E_i\rangle p \wedge \bigwedge_{1\le k\le n} \langle E_i\rangle[E_k]p \\ &\Rightarrow [E_j]p \vee \bigvee_{1\le l\le n} [E_j]\langle E_l\rangle p \\ &\Rightarrow [E_j]\Bigl(p \vee \bigvee_{1\le l\le n} \langle E_l\rangle p\Bigr) \ .\end{aligned}$$

Unter der Voraussetzung (††) gilt also immer auch Eigenschaft (†), und demnach ist es zum Beweis der Einschritt-Konfluenz-Eigenschaft ausreichend, die Allgemeingültigkeit von (††) für alle i^*, j^* mit $i^* \neq j^*$ zu zeigen.[2]

Wenden wir uns nun also der Transformation der Bedingung (††) nach Aussagenlogik zu. Da wir nur an der Gültigkeit in der regulären Standardstruktur interessiert sind (siehe 4.5), dürfen wir in unserer Transformation auch Lemma 3.4.2 verwenden. Beginnend mit dem ersten Teil von (††), erhalten wir für die Aussage $H(p) := \langle E_i\rangle[E_j]p \Rightarrow [E_j]\langle E_i\rangle p$ unter Berücksichtigung der Definition der Einzelschritt-Zusteuerungsprogramme E_i:

$$\begin{aligned}H(p) &\equiv [E_i]\langle E_j\rangle\neg p \vee [E_j]\langle E_i\rangle p \\ &\equiv [(Z_i \wedge \neg x_i)?x_i := \mathbf{true}]\langle(Z_j \wedge \neg x_j)?x_j := \mathbf{true}\rangle\neg p \ \vee \\ &\qquad\qquad [(Z_j \wedge \neg x_j)?x_j := \mathbf{true}]\langle(Z_i \wedge \neg x_i)?x_i := \mathbf{true}\rangle p \\ &\equiv (Z_i \wedge \neg x_i \Rightarrow (Z_j \wedge \neg x_j \wedge \neg p|_{x_j=\mathbf{true}})|_{x_i=\mathbf{true}}) \ \vee \\ &\qquad\qquad (Z_j \wedge \neg x_j \Rightarrow (Z_i \wedge \neg x_i \wedge \neg p|_{x_i=\mathbf{true}})|_{x_j=\mathbf{true}})\end{aligned}$$

für alle $p \in \Phi_0$ und damit insbesondere

$$\begin{aligned}H(x_i) &\equiv \neg(Z_i \wedge \neg x_i) \ \vee \ (Z_j \wedge \neg x_j \Rightarrow (Z_i \wedge \neg x_i)|_{x_j=\mathbf{true}}) \\ &\equiv Z_i \wedge \neg x_i \wedge Z_j \wedge \neg x_j \ \Rightarrow \ (Z_i \wedge \neg x_i)|_{x_j=\mathbf{true}}, \\ H(x_j) &\equiv Z_i \wedge \neg x_i \wedge Z_j \wedge \neg x_j \ \Rightarrow \ (Z_j \wedge \neg x_j)|_{x_i=\mathbf{true}} \ .\end{aligned}$$

Für alle $p \notin \{x_i, x_j\}$ gilt außerdem $H(x_i) \wedge H(x_j) \Rightarrow H(p)$, wie man anhand der dann gültigen Äquivalenz

$$\begin{aligned}H(p) \ \equiv \ Z_i \wedge \neg x_i \wedge Z_j \wedge \neg x_j \ \Rightarrow& \\ &(Z_j \wedge \neg x_j)|_{x_i=\mathbf{true}} \wedge \neg p \vee (Z_i \wedge \neg x_i)|_{x_j=\mathbf{true}} \wedge p\end{aligned}$$

---

[2] Die bewiesene Eigenschaft stellt die PDL-Formulierung einer allgemeinen Aussage über die Vertauschbarkeit bei Summen von Relationen dar.

leicht überprüft. Gültigkeit der Formel $H(x_i) \wedge H(x_j)$, also Gültigkeit von

$$Z_i \wedge \neg x_i \wedge Z_j \wedge \neg x_j \;\Rightarrow\; (Z_j \wedge \neg x_j)|_{x_i=\mathbf{true}} \wedge (Z_i \wedge \neg x_i)|_{x_j=\mathbf{true}} \qquad (+)$$

für alle $i, j \in \{1, \ldots, n\}$ mit $i \neq j$, ist demnach äquivalent zur Gültigkeit des ersten Teils von (††) für alle p in der Struktur $\mathcal{K}^{\mathcal{S}}_{\Box}$. Der zweite Teil von (††) ist allgemeingültig in $\mathcal{K}^{\mathcal{S}}_{\Box}$, da $\langle E_i \rangle p$ äquivalent ist zu $Z_i \wedge \neg x_i \wedge p|_{x_i=\mathbf{true}}$, woraus $Z_i \wedge \neg x_i \Rightarrow p|_{x_i=\mathbf{true}}$ folgt, was wiederum äquivalent zu $[E_i]p$ ist. Somit ist die Einschritt-Konfluenz der Zusteuerung äquivalent zur Gültigkeit der aussagenlogischen Formel (+), die sich unter Verwendung der Eigenschaft $i \neq j$ noch weiter vereinfachen lässt zur Bedingung

$$Z_i \wedge \neg x_i \wedge Z_j \wedge \neg x_j \;\Rightarrow\; Z_j|_{x_i=\mathbf{true}} \wedge Z_i|_{x_j=\mathbf{true}}, \tag{4.23}$$

die aus Symmetriegründen sogar nur für alle $1 \leq i < j \leq n$ gelten muss.

In unserer Implementation BIS (siehe Anhang A) haben wir eine leicht veränderte Formel verwendet, die auch die Baubarkeit miteinbezieht. Dabei wird obige Einschritt-Konfluenz-Eigenschaft nur für baubare Aufträge verlangt:

$$\begin{aligned}&(x_1 \Rightarrow B_1) \wedge \cdots \wedge (x_n \Rightarrow B_n) \wedge \\ &\qquad\qquad (Z_i \wedge \neg x_i \wedge Z_j \wedge \neg x_j \;\Rightarrow\; Z_j|_{x_i=\mathbf{true}} \wedge Z_i|_{x_j=\mathbf{true}}) \ .\end{aligned} \tag{4.24}$$

Die zusätzliche Einschränkung ergibt sich analog der Herleitung von 4.19. Davon ausgehend, dass die Stabilität der Zusteuerung gegeben ist, sind auch alle weiter zugesteuerten Aufträge baubar, womit man, zumindest was die Baubarkeit betrifft, der Formalisierungsvariante 4.9 nahe kommt.

Einschränkung der Auftragsmenge. Bevor wir uns der Übersetzung des letzten noch verbleibenden Tests zuwenden, wollen wir noch einige allgemeine Anmerkungen zur Einschränkung der Auftragsmenge mittels Formeln und deren Transformation nach Aussagenlogik machen. Diese werden dann auch bei dem noch zu besprechenden Test auf mögliche Codes Anwendung finden. Die bei solchen restringierten Tests typische Fragestellung ist die nach der Erfüllbarkeit obiger Formel 4.16, also nach der Erfüllbarkeit von

$$A_1 \wedge \langle Z; B \rangle A_2$$

in der regulären Standardstruktur $\mathcal{K}^{\mathcal{S}}_{\Box}$. Dabei sind A_1 und A_2 die Einschränkungsformeln vor bzw. nach der Zusteuerung und Baubarkeitsprüfung. Für den Fall $A_1 \equiv \top$ haben wir mit der durch 4.19 gegebenen Übersetzung nach Aussagenlogik

$$(Z_1 \Rightarrow x_1) \wedge \cdots \wedge (Z_n \Rightarrow x_n) \wedge \ (x_1 \Rightarrow B_1) \wedge \cdots \wedge (x_n \Rightarrow B_n) \wedge \ A_2$$

bereits eine Lösung. Enthält A_1 keine negativen Literale, so kann man unter Verwendung von Segerbergs Axiom (iii), Lemma 3.2.2(1) und Lemma 3.3.2 leicht die Gültigkeit der beiden Implikationen

$$\begin{aligned} x \wedge y &\Rightarrow [Z; B](x \wedge y) \\ x \vee y &\Rightarrow [Z; B](x \vee y) \end{aligned}$$

für $x, y \in \Phi_0$ nachweisen, durch deren induktive Erweiterung auf komplexere negationsfreie Formeln eine Verschiebung von A_1 in eine Einschränkung nach Programmausführung ermöglicht wird, so dass aus dem ursprünglichen Test in diesem speziellen Fall der einfachere Test

$$\langle Z; B\rangle(A_1 \wedge A_2)$$

wird, der sich dann mit der zuvor genannten Methode behandeln lässt.

Einschränkungen A_1 vor Programmausführung, die negierte Variablen enthalten, also Wünsche des Kunden, eine bestimmte Ausstattung nicht in seinem Fahrzeug zu haben, lassen sich so nicht behandeln. Es ist aber sowieso fraglich, ob dieser Fall nicht besser durch eine Einschränkung nach Programmausführung zu fassen ist. Ansonsten wäre es nämlich möglich, dass trotz ausdrücklichem Kundenwunsch eine Ausstattung durch die Zusteuerung dem Fahrzeug hinzugefügt wird. Bei einer als Teil von Formel A_2 modellierten Einschränkung kann dies nicht passieren.

Mögliche Codes. Nach den Vorbemerkungen des letzten Abschnitts wollen wir den Test auf mögliche Codes so verstanden wissen, dass damit Codes aufgefunden werden sollen, die in einem komplett zugesteuerten und baubaren Auftrag unter einer Nebenbedingung A, die der eventuell modifizierte Auftrag erfüllt, auftreten können. Damit ist für eine vorgegebene Einschränkung A ein Code x möglich, falls die Formel

$$\langle Z; B\rangle(A \wedge x)$$

in der regulären Standardstruktur $\mathcal{K}^{\mathcal{S}}_{\Box}$ erfüllbar ist. Wir haben damit den Test auf mögliche Codes auf einen Test der Form 4.19 zurückgeführt, den wir wie oben beschrieben behandeln können. Man erkennt hierbei auch, dass ein möglicher Code ein Spezialfall eines zulässigen (d.h. nicht unzulässigen) Codes ist.

4.1.4 Experimentelle Ergebnisse

Sämtliche Tests des letzten Abschnitts sind in unserem Baubarkeits-Informationssystem BIS (siehe Anhang A) implementiert. Wir haben in BIS einen Beweiser eingesetzt, der direkt die Erfüllbarkeit beliebiger aussagenlogischer Formeln überprüfen kann. Dies steht im Gegensatz zu vielen anderen Erfüllbarkeits-Checkern, die Eingabeformeln in konjunktiver Normalform (CNF) oder Klauseldarstellung erwarten. Der besseren Vergleichbarkeit mit anderen Beweisern und anderen Erfüllbarkeitsproblemen wegen haben wir die in diesem Abschnitt angegebenen Resultate allerdings mit einem SAT-Checker generiert, der ebenfalls auf Formeln in konjunktiver Normalform arbeitet (siehe Kapitel 6).

Da die meisten Tests Instanzen der Formel 4.17 sind, wollen wir der vereinfachten Notation wegen eine Abkürzung für deren äquivalente, zum automatischen Beweisen verwendete, aussagenlogische Übersetzung 4.19 einführen. Wir bezeichnen also die in diesen Tests auftretende Teilformel, die die Baubarkeits- und Zusteuerungsformeln enthält, mit $\mathcal{B}$, also

$$\mathcal{B} \;:=\; (Z_1 \Rightarrow x_1) \wedge \cdots \wedge (Z_n \Rightarrow x_n) \wedge \; (x_1 \Rightarrow B_1) \wedge \cdots \wedge (x_n \Rightarrow B_n) \;\; .$$

Damit lassen sich Tests dieser Art als Erfüllbarkeitstests der Form $\mathcal{B} \wedge F$ für wechselndes F, aber pro Baureihen-Ausführungsart festes $\mathcal{B}$ schreiben.

Nr.	Abk.	Test-Bezeichnung	F	$\mathcal{B} \wedge F$
1	UC	Code x_i unzulässig	x_i	unerfüllbar
2	NC	Code x_i notwendig	$\neg x_i$	unerfüllbar
3	UT	Teil t_i überflüssig	T_i	unerfüllbar
4	MT	Mehrdeutigkeit t_i/t_j	$T_i \wedge T_j$	erfüllbar
5	–	Code x_i unter A möglich	$A \wedge x$	erfüllbar
6	–	Codegruppe $\{g_1, \ldots, g_k\}$	$\neg S_a^1(g_1, \ldots, g_k)$	unerfüllbar

Tabelle 4.1: Zusammenstellung der Tests der Form $\mathcal{B} \wedge F$.

In Tabelle 4.1 haben wir alle solchen von BIS zur Verfügung gestellten Tests zusammengestellt. Dabei steht in der vierten Spalte die für den Test verwendete variable Formelkomponente F, und in der letzten Spalte ist angegeben, ob Erfüllbarkeit oder Unerfüllbarkeit mit der bezeichneten Eigenschaft übereinstimmt. In der zweiten Spalte haben wir noch eine Abkürzung angegeben, unter der wir später auf die entsprechende Test-Variante Bezug nehmen wollen.

Baureihe/AA	n	k	l	n’	k’	l’
C129/FR	1739	6268	21406	574	3859	13695
C140/FC	1691	4920	13497	485	2446	8561
C140/FV	1705	7420	32413	555	4940	23987
C140/FW	1701	5649	18445	549	3055	11837
C163/FW	1791	4364	14914	570	2194	9484
C168/FW	1804	6744	27570	653	4299	19658
C169/FV	1402	1959	2738	50	65	143
C170/FR	1765	9711	47215	554	7579	38693
C171/FR	1743	3853	9394	451	1641	3692
C202/FS	1822	7946	35266	698	4954	23399
C202/FW	1820	9947	44587	734	7290	35225
C203/FCL	1819	4794	11883	571	2216	7179
C203/FS	1838	5762	16977	618	2778	9794
C203/FW	1885	6688	20182	665	3575	11940
C208/FA	1803	6038	26496	633	3720	17185
C208/FC	1800	6295	30829	652	3975	20358
C209/FA	1785	3869	9829	516	1764	5562
C209/FC	1834	4573	11332	572	2250	6680
C210/FS	1870	6924	26715	723	4025	15103
C210/FVF	1835	3832	10354	493	1693	5824
C210/FW	1891	8495	38113	775	5491	23370
C211/FS	1602	3409	7154	247	673	2327
C211/FW	1634	5628	15501	342	2911	10630
C215/FC	1735	4417	11934	494	2290	8112
C220/FV	1782	5970	21244	597	3368	13855
C220/FW	1775	6017	21392	583	3401	13810
C230/FR	1775	4273	11768	515	2088	7275
C250/FV	1442	2262	3784	129	224	538
C250/FW	1442	2262	3784	129	224	538
C638/FKA	1753	7242	23129	528	4966	19064
C638/FKB	1750	3456	6937	506	1253	3343
C638/FVK	1727	2942	5986	452	948	2708
D1119/M20	1392	1671	2419	36	156	615
D1119/M23	1405	1963	3522	47	144	534

Tabelle 4.2: Charakteristika der Klauseldarstellung der Mercedes-Benz Produktdaten.

Für jede in DIALOG dokumentierte Baureihen-Ausführungsart (siehe Tabelle 2.1) ergibt sich also eine eigene Formel $\mathcal{B}$. In Tabelle 4.2 sind einige Charakteristika dieser Formeln zusammengestellt, die zuvor nach Klauselform konvertiert wurden. Die Tabelle führt nach der Bezeichnung der Baureihen-Ausführungsart in den nächsten drei Spalten Angaben auf, die sich auf die unmittelbar aus den DIALOG-Daten generierte Formel $\mathcal{B}$ beziehen. Dies sind der Reihe nach die Anzahl der aussagenlogischen Variablen (n), die Anzahl der Klauseln (k) und die Anzahl der Literalvorkommen (l). Die nächsten drei Spalten nehmen auf eine reduzierte Formel Bezug, die aus der ursprünglichen Formel $\mathcal{B}$ durch Anwendung von *unit-propagation*, also wiederholter Einheits-Resolution zusammen mit Einheits-Subsumtion, entstanden ist. In den Spalten steht im Einzelnen: die Anzahl der Variablen nach Reduktion (n'), wobei Variablen, die nach Reduktion nur in einer Einheitsklausel vorkommen, nicht mitgezählt wurden; die Anzahl der Klauseln (k') der reduzierten Formel, wobei Einheitsklauseln nicht mitgezählt wurden; und die Anzahl der in der reduzierten Formel (ohne Einheitsklauseln) vorkommenden Literale (l').

Die Differenzen in der Variablenanzahl bezüglich Tabelle 2.1 kommen dadurch zustande, dass wir hier nicht nur Variablen berücksichtigen, für die eine Baubarkeits- oder Zusteuerungsregel vorhanden ist, sondern auch solche, die nur in den aussagenlogischen Bedingungen anderer Regeln vorkommen, für die selbst aber nicht zwingend eine Baubarkeits- oder Zusteuerungsregel vorhanden sein muss (was z.B. bei Ländercodes der Fall ist).

Neben den auf Formel $\mathcal{B}$ beruhenden Tests gibt es noch die beiden speziellen Tests auf Stabilität (Abk.: SZ) und Reihenfolgeunabhängigkeit (Abk.: RZ) der Zusteuerung, deren aussagenlogische Übersetzung durch die Formeln 4.21 und 4.24 gegeben ist. Auch für diese Tests werden wir experimentelle Ergebnisse vorweisen.

Wir werden nun Laufzeiten und Suchraumgrößen unserer Implementation des Davis-Putnam-Verfahrens (siehe Kapitel 6) für die verschiedenen Konsistenztests der Dokumentation vorstellen. Die Laufzeiten werden wir dabei immer in Sekunden angeben, die Suchraumgrößen in Anzahl der Fallunterscheidungen, die der Beweiser vornimmt (*case-splittings* im entsprechenden Schritt des DP-Algorithmus). Aufgrund der hohen Zahl der durchgeführten Tests (insgesamt 64328) werden wir nicht für alle Testläufe individuelle, sondern in erster Linie zusammenfassende Messergebnisse vorstellen. Dabei geben wir sowohl für die Laufzeiten als auch für die Suchraumgrößen nur die durchschnittlichen bzw. maximalen Messwerte aufgeschlüsselt nach verschiedenen Testgruppen an. Alle Tests wurden auf einer Sun-Blade-1000, bestückt mit 512 MB Hauptspeicher und einer 750 MHz UltraSPARC-III-CPU, durchgeführt. Zur Laufzeitmessung haben wir die hochauflösende Uhr des Systems verwendet, die ein Auflösungsvermögen von 200ns besitzt und Realzeit misst. Die Zeitangaben beziehen sich immer auf die reine Suche des DP-Algorithmus, Initialisierungszeiten und Zeiten, um die CNF-Datei einzulesen, wurden dabei nicht mitgerechnet. Diese nicht berücksichtigten Zeiten lagen aber immer unter 0.5s.

Wir haben für unsere Messungen zehn Baureihen ausgewählt, die ein möglichst breites Spektrum der größeren und damit potentiell für das automatische Beweisen schwierigeren Baureihen umfassen. Für jede dieser Baureihen haben wir Messungen eines jeden der sechs erwähnten Konsistenztests durchgeführt. In den Messwerttabellen haben wir neben den durchschnittlichen und maximalen Suchzeiten ($\bar{t}_s$ und $t_s^{\max}$), den durchschnittlichen und maximalen Suchraumgrößen ($\bar{b}$ und $b^{\max}$) auch die Anzahl der insgesamt für diese Baureihe und Testvariante durchgeführten Beweise (n) und zwei weitere Größen, %unerf. bzw. %erf. und %UR-e., aufgeführt. Letztere Größe gibt den Anteil der Beweise an, die bereits durch Einheits-Resolution (*unit resolution*), also ohne wirkliche Suche, entschieden werden konnten. Erstere (%unerf. bzw. %erf.) bezeichnet den Anteil der unerfüllbaren bzw. erfüllbaren Instanzen. Diese Größe kann als Anhaltspunkt für die Anzahl der vom System BIS gefundenen Inkonsistenzen gewertet werden.

Wir führen jetzt die einzelnen Tests en detail auf und werden dabei weitere, den jeweiligen Test betreffende ergänzende Bemerkungen beifügen.

Notwendige und unzulässige Codes. Bei den notwendigen und unzulässigen Codes (Tabelle 4.3) wurde durch BIS bereits eine Filterung vorgenommen, die die Anzahl n der notwendigen Testinstanzen reduziert. Dabei wird das durch vorhergehende Tests erworbene Wissen mit verwendet. Ist beispielsweise Code c nicht notwendig, so kann man die vom automatischen Beweiser nebenbei generierten Beispiele (Modelle der Formel) zur Vermeidung weiterer Tests verwenden [Sin97, KK01]. Ist z.B. Code d in einem dieser Modelle mit wahr belegt, so weiß man sofort, also ohne weiteren aussagenlogischen Erfüllbarkeitstest, dass dieser Code d nicht unzulässig sein kann.

Sämtliche notwendigen und unzulässigen Codes wurden bereits durch Einheits-Resolution erkannt (siehe Spalten %unerf. und %UR-e). Für alle unerfüllbaren Formeln kam der Algorithmus daher ohne wirkliche Suche aus.

Notwendige Codes (NC):

BR/AA	n	%unerf.	%UR-e.	$\bar{b}$	$b_{\max}$	$\bar{t}_s$	$t_s^{\max}$
C168/FW	46	8.7	8.7	6.4	17	0.0026	0.0046
C170/FR	72	6.9	6.9	7.3	8	0.0028	0.0035
C202/FS	71	5.6	5.6	5.6	6	0.0021	0.0024
C202/FW	67	6.0	6.0	4.7	5	0.0024	0.0030
C208/FA	80	6.2	6.2	8.4	9	0.0020	0.0022
C208/FC	71	5.6	5.6	6.5	8	0.0021	0.0028
C210/FS	77	6.5	6.5	4.6	5	0.0017	0.0079
C210/FW	86	5.8	5.8	3.8	4	0.0022	0.0028
C220/FV	67	7.5	7.5	12.6	16	0.0024	0.0032
C638/FKA	52	0.0	0.0	8.9	10	0.0024	0.0028

Unzulässige Codes (UC):

BR/AA	n	%unerf.	%UR-e.	$\bar{b}$	$b_{\max}$	$\bar{t}_s$	$t_s^{\max}$
C168/FW	306	0.3	0.3	8.8	15	0.0026	0.0045
C170/FR	182	0.0	0.0	7.5	10	0.0027	0.0102
C202/FS	266	0.0	0.0	6.9	12	0.0024	0.0045
C202/FW	321	0.3	0.3	5.9	12	0.0028	0.0060
C208/FA	219	0.0	0.0	8.8	11	0.0020	0.0027
C208/FC	242	0.0	0.0	7.2	12	0.0018	0.0035
C210/FS	346	0.3	0.3	6.4	14	0.0020	0.0040
C210/FW	346	0.9	0.9	5.7	14	0.0026	0.0055
C220/FV	269	4.1	4.1	13.0	17	0.0024	0.0033
C638/FKA	360	2.8	2.5	7.7	16	0.0020	0.0039

Tabelle 4.3: Laufzeiten und Suchraumgrößen der Tests auf notwendige und unzulässige Codes.

Stabilität und Reihenfolgeunabhängigkeit der Zusteuerung. Bei der Prüfung sowohl der Stabilität als auch der Reihenfolgeunabhängigkeit der Zusteuerung (siehe Tabelle 4.4) haben wir wiederum die von BIS getroffene Vorauswahl der wirklich dem automatischen Beweiser zugeführten Testinstanzen übernommen. Formeln, bei denen die Komponente Z_i aus Formel 4.21 bereits unerfüllbar war (meist verursacht durch das Fehlen jeglicher Zusteuerungsbedingung für Code x_i), haben wir nicht in die Messung miteingeschlossen.

Bei der Prüfung der Reihenfolgeunabhängigkeit ergäben sich bei stupider Durchführung aller Beweise $\binom{n}{2} = \frac{1}{2} \cdot n \cdot (n-1)$ Testinstanzen bei n Codes, so dass hier eine Vorauswahl dringend geboten ist. Diese beruht hier auf einem vorhergehenden Erfüllbarkeitstest der deutlich kleineren Formel 4.23. Ist diese bereits unerfüllbar, so wird der eigentliche Test gar nicht erst durchgeführt. Es ergibt sich so eine deutliche Reduktion der Anzahl der größeren Testinstanzen für den Beweiser.

Überflüssige Teile und Mehrdeutigkeiten in der Stückliste. Beim Test auf überflüssige Teile (oder genauer: überflüssige Positionsvarianten; siehe Tabelle 4.5) ist eine Filterung schwerer durchzuführen. Unerfüllbare Selektionsformeln – auch in der Darstellungsform der langen Coderegeln – sind eher unwahrscheinlich. So kommt hier eigentlich nur eine Gruppierung gleichlautender Formeln in Frage. Dieses Verfahren zur Minimierung der Beweisanzahl wird in BIS verwendet und liegt daher auch unseren Messergebnissen zugrunde. Es ergibt sich dadurch immerhin eine Reduktion um Faktoren zwischen 2.23 (210/FS) und 4.64 (C170/FR). Trotzdem bleibt dieser Konsistenztest, gemessen an der Zahl der durchzuführenden Beweise, mit Abstand der aufwändigste.

Die überraschend hohe Anzahl der unerfüllbaren Instanzen – und damit unnötigen Positionsvarianten – bei der Baureihe C210 lässt sich wahrscheinlich zum Großteil dadurch begründen, dass das Datenmaterial von einem Zeitpunkt stammt, zu dem bereits die Umstellung auf die Nachfolgebaureihe C211 im Gange war. Es wird hier aber auch ersichtlich, dass es in einem solchen Fall kaum praktikabel ist, die von BIS gemeldeten Inkonsistenzen eine nach der anderen von Hand ohne weitere maschinelle Unterstützung daraufhin zu prüfen, ob es sich wirklich um einen Dokumentationsfehler handelt.

Beim Test auf Mehrdeutigkeiten in der Stückliste haben wir eine Reduktion nur insofern durchgeführt, als wir bereits unerfüllbare Teilformeln $T_i \wedge T_j$

Stabilität der Zusteuerung (SZ):

BR/AA	n	%erf.	%UR-e.	$\bar{b}$	$b_{\max}$	$\bar{t}_s$	$t_s^{\max}$
C168/FW	118	20.3	72.0	5.8	561	0.0016	0.1479
C170/FR	87	2.3	93.1	1.1	40	0.0004	0.0130
C202/FS	109	5.6	70.6	1.0	16	0.0004	0.0061
C202/FW	111	17.1	73.0	1.3	18	0.0009	0.0267
C208/FA	96	4.2	87.5	0.6	17	0.0002	0.0064
C208/FC	104	4.8	88.5	0.6	17	0.0002	0.0071
C210/FS	110	5.5	86.4	0.7	14	0.0003	0.0059
C210/FW	117	7.7	82.9	1.0	14	0.0006	0.0119
C220/FV	106	5.7	87.7	0.6	23	0.0003	0.0136
C638/FKA	54	13.0	83.3	0.3	3	0.0001	0.0015

Reihenfolgeunabhängigkeit der Zusteuerung (RZ):

BR/AA	n	%erf.	%UR-e.	$\bar{b}$	$b_{\max}$	$\bar{t}_s$	$t_s^{\max}$
C168/FW	80	5.0	93.8	0.2	5	0.0001	0.0010
C170/FR	32	6.2	90.6	0.8	17	0.0003	0.0067
C202/FS	46	4.3	87.0	0.3	6	0.0001	0.0028
C202/FW	57	3.5	82.5	0.5	16	0.0004	0.0092
C208/FA	64	10.9	84.4	0.6	16	0.0002	0.0060
C208/FC	70	14.3	81.4	0.7	16	0.0002	0.0064
C210/FS	40	2.5	90.0	0.8	12	0.0003	0.0047
C210/FW	59	3.4	91.5	0.7	12	0.0004	0.0063
C220/FV	43	4.7	88.4	0.3	5	0.0001	0.0009
C638/FKA	12	58.3	25.0	1.9	4	0.0008	0.0020

Tabelle 4.4: Laufzeiten und Suchraumgrößen der Tests auf Stabilität und Reihenfolgeunabhängigkeit der Zusteuerung.

Überflüssige Teile/Positions-Varianten (UT):

BR/AA	n	%unerf.	%UR-e.	$\bar{b}$	$b_{\max}$	$\bar{t}_s$	$t_s^{\max}$
C168/FW	2707	6.2	5.8	6.9	19	0.0020	0.0253
C170/FR	1976	1.2	1.1	6.3	13	0.0024	0.0072
C202/FS	5669	9.6	9.6	5.0	13	0.0017	0.0077
C202/FW	6508	8.1	8.0	4.5	13	0.0022	0.0098
C208/FA	3261	2.5	2.5	7.5	13	0.0017	0.0038
C208/FC	3536	3.0	3.0	6.2	34	0.0018	0.0152
C210/FS	9389	21.4	21.4	4.5	14	0.0013	0.0330
C210/FW	13049	29.7	29.6	3.8	17	0.0016	0.0426
C220/FV	3074	5.4	5.3	11.7	19	0.0024	0.0178
C638/FKA	1526	5.2	5.2	9.5	18	0.0025	0.0109

Mehrdeutigkeiten in der Stückliste (MT):

BR/AA	n	%erf.	%UR-e.	$\bar{b}$	$b_{\max}$	$\bar{t}_s$	$t_s^{\max}$
C168/FW	368	13.6	86.4	1.0	15	0.0003	0.0036
C170/FR	473	6.1	93.9	0.5	8	0.0002	0.0035
C202/FS	888	9.2	90.8	0.5	7	0.0002	0.0030
C202/FW	1034	13.7	86.3	0.6	8	0.0004	0.0098
C208/FA	851	10.7	89.3	0.9	10	0.0002	0.0025
C208/FC	901	7.8	92.2	0.5	8	0.0002	0.0031
C210/FS	1225	22.0	78.0	1.1	11	0.0004	0.0038
C210/FW	1464	23.2	76.8	1.0	12	0.0005	0.0043
C220/FV	501	8.4	91.6	1.1	17	0.0002	0.0033
C638/FKA	867	9.9	90.1	0.9	16	0.0003	0.0045

Tabelle 4.5: Laufzeiten und Suchraumgrößen der Tests auf überflüssige Teile und Mehrdeutigkeiten in der Stückliste.

nicht in unsere Messreihe mitaufgenommen haben; die Unerfüllbarkeit solcher Testinstanzen ließe sich aber aufgrund der geringen Variablenanzahl selbst mit einfacheren Mitteln leicht zeigen. Durch die Anwendung dieser einfachen Vorentscheidung lässt sich bereits ein Großteil der vorhandenen Instanzen bewältigen. Die noch verbleibenden sind in Tabelle 4.5 aufgeführt.

Da durch die Konstruktion der Teileselektion über kurze/lange Coderegeln ein Ausschluss sich gegenseitig ausschließender Varianten bereits sichergestellt sein soll, ist die geringere Fehlerquote bei diesem Test nicht weiter überraschend. Vielmehr zeigt es sich, dass diese Konstruktion zum Ausschluss einer mehrdeutigen Selektion nicht ausreichend ist.

Bei allen Beweisen zeigten sich ohne Ausnahme auffällig kurze Laufzeiten und auffällig kleine Suchräume. Wenn man bedenkt, dass die untersuchten Formeln bis zu 1891 verschiedene Variablen enthalten und (ohne Vorhandensein und Nutzung spezieller Eigenschaften der Formeln) im schlimmsten Fall mit einer exponentiellen Laufzeit und Suchraumgröße zu rechnen ist, muss eine maximale Laufzeit von 0.1479 s und eine maximale Suchraumgröße von 561 Fallunterscheidungen bei einer Gesamtzahl von 64328 durchgeführten Konsistenztests doch sehr überraschend erscheinen.

Zur weiteren, eingehenderen Untersuchung dieses Phänomens wählten wir die härteren der 64328 Instanzen aus. Auswahlkriterium war dabei ein Suchraum von mindestens 15 Fallunterscheidungen ($b \geq 15$) bei erfüllbaren Instanzen und mindestens zwei Fallunterscheidungen ($b \geq 2$) bei unerfüllbaren Instanzen. Nach Filterung anhand dieses Kriteriums verblieben noch 553 Instanzen. Für diese Instanzen haben wir mit vier verschiedenen Einstellungen unseres Beweisers die Messungen wiederholt. Die Einstellungsvarianten des Beweisers umfassten zwei Parameter, die sich in der Beweiser-Effizienz als wesentlich herausgestellt haben (siehe Kapitel 6 für eine ausführlichere Darstellung):

Literal-Auswahlstrategie: Dieser Parameter bestimmt, nach welchem Literal die nächste Fallunterscheidung im Davis-Putnam-Algorithmus vorgenommen wird. Wir haben hier neben der in den bisher erwähnten Experimenten verwendeten Strategie SPC auch die Strategie MO ausprobiert. Bei ersterer (SPC: *shortest positive clause*) wird ein beliebiges Literal einer kürzesten in der Probleminstanz vorkommenden positiven Klausel zur Fallunterscheidung verwendet. Bei letzterer (MO: *maximal occurrence*) wird ein Literal mit der größten Anzahl der Vorkommen in der Formel ausgewählt. Genauer wählt diese Strategie ein Literal aus, welches die Funktion $f(l) = p(l) + n(l)$ maximiert, wobei $p(l)$ die Anzahl der positiven und $n(l)$ die der negativen Vorkommen von Literal l in der Formel bezeichnet.

Lemma-Generierung: Die Lemma-Generierung (zum Teil auch als “Lernen” bezeichnet) vermeidet das wiederholte Durchsuchen von Teilen des Suchraums, die aufgrund derselben Ursache keine Lösungen enthalten können. Diese Ursache kann als weitere Einschränkungsklausel (Lemma) der Probleminstanz hinzugefügt werden. Wir haben bei aktivierter Lemma-Generierung nur diejenigen Klauseln dem Problem hinzugefügt, welche nicht mehr als zehn Literale enthielten. Klauseln bis zu einer Größe von hundert Literalen wurden zum dabei ebenfalls aktivierten nicht-chronologischen Backtracking (siehe Kapitel 6) zugelassen und temporär der Probleminstanz hinzugefügt.

Die Ergebnisse der sich ergebenden vier Einstellungsmöglichkeiten sind in den Tabellen 4.6 und 4.7 dargestellt. Wir haben hier nicht die Messergebnisse aller Probleminstanzen aufgeführt, sondern nur diejenigen maximaler Suchraumgröße bei einer der vier Einstellungen. Erfüllbare und unerfüllbare Instanzen haben wir getrennt behandelt; von ersteren haben wir die größten fünf, von letzteren die größten zwanzig ausgewählt. In den Benennungen der Tabellenspalten verwenden wir das Superskript L für die Variante mit Lemma-Generierung, N für die ohne. Die Subskripte SPC und MO stehen für die erwähnten Literal-Selektionsstrategien.

Man erkennt, dass sich bei verschiedenen Einstellungen enorme Unterschiede hinsichtlich Laufzeit und Suchraumgröße ergeben können, in unseren Experimenten bis zu einem Faktor der Größenordnung 10^6. Lemma-Generierung kann die Wahl einer schlechten Auswahlstrategie (MO) wettmachen, ist aber nicht alleine für die kurzen Laufzeiten verantwortlich, da sich bei der Auswahlstrategie SPC auch ohne Lemma-Generierung kaum schlechtere Laufzeiten ergeben haben.

Zusammenfassend lässt sich zu den Messergebnissen feststellen:

1. Es treten durchweg auffallend kurze Laufzeiten und geringe Suchraumgrößen auf.
2. Die richtige Einstellung zumindest einer der beiden Parameter Literal-Auswahlstrategie und Lemma-Generierung ist zum Erhalt kurzer Laufzeiten essentiell.

Wir werden in Kapitel 5 diese Phänomene ausführlicher analysieren.

BR/AA	Test/Id.	SAT?	$t^{\mathrm{L}}_{\mathrm{SPC}}$	$t^{\mathrm{L}}_{\mathrm{MO}}$	$t^{\mathrm{N}}_{\mathrm{SPC}}$	$t^{\mathrm{N}}_{\mathrm{MO}}$
C202/FS	SZ/122	nein	0.0060	0.0142	0.0993	15064.6003
C220/FV	SZ/121	nein	0.0053	0.0182	0.0045	7999.6767
C168/FW	SZ/128	nein	0.1495	2.3517	0.1109	61.4109
C210/FW	SZ/80	nein	0.0065	0.0136	0.0408	29.4682
C202/FW	SZ/77	nein	0.0070	0.0347	0.0397	32.0426
C210/FW	RZ/59	nein	0.0060	0.0126	0.0784	25.0997
C210/FS	RZ/40	nein	0.0049	0.0123	0.0730	19.2360
C210/FS	SZ/78	nein	0.0030	0.0099	0.0263	12.3926
C202/FW	RZ/57	nein	0.0095	0.0222	0.1061	18.7144
C208/FC	SZ/107	nein	0.0097	0.0296	0.0069	8.6428
C202/FS	SZ/74	nein	0.0043	0.0095	0.0315	7.6350
C208/FC	RZ/70	nein	0.0063	0.0166	0.0452	5.6198
C210/FW	SZ/135	nein	0.0078	0.0099	0.0951	5.5697
C210/FW	SZ/136	nein	0.0070	0.0097	0.0873	4.5972
C210/FS	SZ/129	nein	0.0057	0.0081	0.0753	3.7747
C202/FW	SZ/123	nein	0.0108	0.0237	0.1007	5.8863
C202/FW	SZ/124	nein	0.0098	0.0209	0.1086	5.7139
C210/FS	SZ/130	nein	0.0053	0.0075	0.0686	2.9526
C208/FA	RZ/64	nein	0.0067	0.0147	0.0348	1.9771
C168/FW	UT/640	nein	0.0087	0.0190	0.0053	3.5390
C208/FC	UT/3529	ja	0.0154	0.0358	0.1971	0.0398
C220/FV	UT/104	ja	0.0046	0.0150	0.0046	0.1116
C220/FV	UT/105	ja	0.0049	0.0148	0.0045	0.1116
C220/FV	UT/106	ja	0.0046	0.0147	0.0045	0.1176
C638/FKA	UT/1040	ja	0.0038	0.0249	0.0036	0.0591

Tabelle 4.6: Laufzeiten harter Instanzen: Vergleich verschiedener Varianten des Davis-Putnam-Algorithmus.

BR/AA	Test/Id.	SAT?	$b^{\mathrm{L}}_{\mathrm{SPC}}$	$b^{\mathrm{L}}_{\mathrm{MO}}$	$b^{\mathrm{N}}_{\mathrm{SPC}}$	$b^{\mathrm{N}}_{\mathrm{MO}}$
C202/FS	SZ/122	nein	16	35	245	69239176
C220/FV	SZ/121	nein	23	85	31	47887596
C168/FW	SZ/128	nein	561	8810	591	252165
C210/FW	SZ/80	nein	9	25	101	92413
C202/FW	SZ/77	nein	10	54	97	74167
C210/FW	RZ/59	nein	12	26	219	73280
C210/FS	RZ/40	nein	12	26	219	65547
C210/FS	SZ/78	nein	7	24	99	41480
C202/FW	RZ/57	nein	16	31	223	38564
C208/FC	SZ/107	nein	3	117	3	29880
C202/FS	SZ/74	nein	9	17	111	19952
C208/FC	RZ/70	nein	16	42	139	19798
C210/FW	SZ/135	nein	14	17	239	14126
C210/FW	SZ/136	nein	13	16	219	11532
C210/FS	SZ/129	nein	14	17	239	11209
C202/FW	SZ/123	nein	18	31	211	10517
C202/FW	SZ/124	nein	16	28	223	10314
C210/FS	SZ/130	nein	13	16	219	8650
C208/FA	RZ/64	nein	16	40	111	7253
C168/FW	UT/714	nein	2	18	2	6349
C208/FC	UT/3529	ja	34	81	435	85
C220/FV	UT/104	ja	15	53	16	185
C220/FV	UT/105	ja	15	53	16	185
C220/FV	UT/106	ja	15	53	16	185
C638/FKA	UT/1040	ja	15	86	15	135

Tabelle 4.7: Suchraumgrößen harter Instanzen: Vergleich verschiedener Varianten des Davis-Putnam-Algorithmus.

4.2 Dynamische Aspekte

Zwischen der Entwicklung des ersten Fahrzeugprototypen einer Serie und dem Zeitpunkt, zu dem das letzte Fahrzeug vom Band rollt, können viele Jahre vergehen. So ist es nicht weiter überraschend, dass sich in dieser Zeit sowohl das Produkt als auch dessen Produktionsbedingungen beträchtlich verändern können. All diese Änderungen müssen sich jedoch auch in der Fahrzeugdokumentation, insbesondere für die Produktion, widerspiegeln.

Wir wollen in diesem Abschnitt Kriterien entwickeln, die zur Sicherstellung der Fehlerfreiheit einer sich verändernden Dokumentation Hilfestellung leisten können. Bevor wir diese vorstellen, werden wir zur Motivation dieser Verifikationskriterien aber noch auf drei typische Änderungen eingehen, denen ein Produkt im Laufe seines Lebenszyklus unterworfen sein kann. Diese Änderungs-Szenarien enthalten sowohl Änderungen des Produkts als auch der Produktionsumgebung; sie umfassen Änderungen der Produktübersicht genauso wie Änderungen der Stückliste.

4.2.1 Änderungs-Szenarien

Teiletausch. Die Gründe, die zum Austausch von Teilen in der Produktion – und damit auch in der Produktdokumentation – führen können, sind vielfältig. Neben technischem Fortschritt oder Wechsel zwischen interner Produktion und externer Beschaffung, ist dies häufig auch lediglich ein Wechsel des Zulieferers.

Auch die Art und Weise, wie der Austausch durchgeführt wird, kann variieren. So kann die Umstellung von vorhandenem Teil zu dessen Nachfolgemodell entweder zu einem festen Zeitpunkt erfolgen, oder es wird ein Überlappungsintervall festgelegt, in dem sowohl das alte als auch das neue Teil in einem Fahrzeug verbaut werden können. Dann muss die Dokumentation natürlich eine Möglichkeit vorsehen, zwischen den beiden Teilevarianten zu unterscheiden und gezielt zu steuern. DIALOG sieht dazu die bereits erwähnten Kontrollcodes vor (siehe Abschnitt 2.3.1). Ein drittes, auch in DIALOG vorhandenes Verfahren, führt die Umstellung von Alt- zu Neuteil genau dann durch, wenn ersteres nicht mehr im Lager vorhanden ist. Diesen letzten Fall werden wir aber nicht weiter berücksichtigen.

An- und Auslauf von Ausstattungscodes. Neue Ausstattungscodes können aufgrund der kontinuierlichen Weiterentwicklung der Produkte auftreten. Der Wegfall bestehender Codes kann dadurch verursacht sein, dass diese von den Kunden nicht mehr nachgefragt und deshalb gestrichen werden oder dass sie in zusammenfassende Ausstattungspakete integriert werden. Die meisten dieser Änderungen werden von den Entwicklungs- und Marketingabteilungen ausgelöst. Im Gegensatz dazu werden zeitliche Kontrollcodes von den Produktionsabteilungen eingesetzt, verwendet und auch wieder gelöscht, hauptsächlich, um den Modelljahrwechsel handzuhaben.

In DIALOG wird ein Code c ungültig gemacht (dies wird z.B. für den Teileauslauf benötigt), indem alle (baureihenspezifischen) Baubarkeitsregeln von c entfernt (siehe z.B. Abbildung 2.8) oder über die Zeitsteuerung deaktiviert werden. Soll ein neuer Code gültig gemacht werden (Anlaufcode), so muss für diesen lediglich (mindestens) eine baureihenspezifische Baubarkeitsregel dokumentiert werden.

An- und Auslauf von Codes tritt hauptsächlich in Zusammenhang mit einem Modelljahrwechsel auf. Dieser ist ein wichtiger Bestandteil der Produktpflege und dient der Zusammenfassung aller im Laufe eines Jahres angefallenen Produktänderungen. Der Modelljahrwechsel geht demnach mit umfangreichen Änderungen der Dokumentation einher, da gewöhnlich beträchtliche Teile des Produkts während eines Jahres überarbeitet werden. Auch ein Großteil der oben erwähnten Teiletauschfälle der Stückliste wird im Zuge eines Modelljahrwechsels abgewickelt.

Die Hauptschwierigkeit bei der Dokumentation eines Modelljahrwechsels besteht darin, die durch den An- und Auslauf eines Codes induzierten Änderungen auf Teileebene abzusehen und notwendig werdende Änderungen in die Teileauswahlregeln richtig einzuarbeiten. Dies mag im Falle der direkt einer Teileauswahlregel zugeordneten Start- und Stoppcodes noch leicht ersichtlich sein, da deren Einfluss auf die Gültigkeit der Regel unmittelbar ersichtlich ist. Da Codes über die Baubarkeitsregeln untereinander in Beziehung stehen, können sich aber auch deutlich komplizierte Abhängigkeiten entwickeln, die dann potentielle Fehlerquellen sind. So kann z.B. der Auslauf eines Codes zum Ungültigwerden eines anderen (über dessen Baubarkeitsbedingung) führen, was dann seinersets die Coderegel eines Teils t unerfüllbar macht. So kann der Auslauf eines Codes x dazu führen, dass ein Teil t nicht mehr benötigt wird, auch wenn dieser Code überhaupt nicht in der Coderegel von t auftritt.

Produktions-Verlagerungen Das letzte Änderungsszenario mit dem wir uns beschäftigen wollen, hängt in erster Linie mit Modifikationen der Produktionsumgebung zusammen. Zum Beispiel werden Fertigungsstationen an Produktionsbändern von Zeit zu Zeit verändert, ausgetauscht oder gestrichen, um sich ändernden Auslastungen oder Modernisierungen anzupassen. Nicht so häufig, aber dafür mit umso größerem Änderungsaufwand verbunden, ist die Verlagerung ganzer Modellreihen von einem Produktionsband zu einem anderen oder gar in eine andere Fertigungsstätte.

Die Anforderungen an das Dokumentationspersonal sind hier ähnlich wie im Falle des An- und Auslaufs von Codes, gehen aber häufig noch darüber hinaus. Hauptproblem ist auch hier das Erkennen von neu bzw. nicht mehr benötigten Teilen (An- und Auslaufteilen), die sich durch Änderungen an der Produktübersicht oder aufgrund anderweitiger Veränderungen ergeben.

Darüberhinaus werden manche dieser Änderungen vom Dokumentationssystem nicht direkt unterstützt (z.B. Produktionsverlagerungen in DIALOG), so dass die sich ergebenden Schwierigkeiten nicht direkt mit Programmunterstützung gehandhabt werden können.

Verifikationsbedarf liegt nur in den letzten beiden dargestellten Fällen (An- und Auslauf von Codes, Produktionsverlagerungen) vor.

4.2.2 Formalisierung und Testmöglichkeiten

Für die Formalisierung der dynamischen Konsistenzkriterien ist es erforderlich, die zeitlichen Einschränkungen der Regeln mitzuberücksichtigen. Grundprinzip unserer Formalisierung ist die logische Erfassung der Dokumentation zu einem oder mehreren festgelegten Stichtagen mit nachfolgendem Vergleich der Testergebnisse.

In einem ersten Schritt werden wir also die Formalisierung der Produktdokumentation aus Abschnitt 3.3 um zeitliche Gültigkeitsintervalle, sowie Start- und Stoppcodes erweitern. Für eine Regel R sei $I(R) = (t_\alpha(R), t_\omega(R))$ das zeitliche Gültigkeitsintervall und $\mathrm{CC}_\alpha(R)$ sowie $\mathrm{CC}_\omega(R)$ deren Start- und Stoppcodes. Zu einem festgelegten Zeitpunkt t^* ergibt sich dann die zeitabhängige Definition der Formeln $B_i(t^*)$, $Z_i(t^*)$ und $T_i(t^*)$ anhand der angepassten Formelbausteine $\mathrm{PCRegel}(c)$, $\mathrm{BRegel}(c, g, p, v)$, $\mathrm{ZRegel}(c, g, p, v)$ und $\mathrm{LCRegel}(t_i)$, die nun ebenfalls zeitabhängig werden. Dazu wird jede dieser Regeln R durch ihr zeitabhängiges Äquivalent $R(t)$ ersetzt. Dieses ist definiert durch

$$R(t) := \begin{cases} R \wedge \mathrm{CC}_\alpha(R) \wedge \neg\mathrm{CC}_\omega(R) & \text{falls } t < t_\alpha(R), \\ R \wedge \neg\mathrm{CC}_\omega(R) & \text{falls } t_\alpha(R) \leq t < t_\omega(R), \\ \bot & \text{falls } t \geq t_\omega(R). \end{cases}$$

Dadurch ergeben sich dann direkt die zeitabhängigen abstrakten Formeln $B_i(t)$, $Z_i(t)$ und $T_i(t)$ des nun zeitabhängigen PDL-Prüfprogramms. Die zeitabhängigen Konsistenzbedingungen ergeben sich analog aus den zeitunabhängigen von Abschnitt 4.1.2, selbiges gilt für deren Übersetzung nach Aussagenlogik. So ergibt sich beispielsweise für die in Abschnitt 4.1.4 definierte Formel $\mathcal{B}$, die die Baubarkeits- und Zusteuerungsbedingungen zusammenfasst, das zeitabhängige Äquivalent

$$\begin{aligned} \mathcal{B}(t) \;=\; & (Z_1(t) \Rightarrow x_1) \wedge \cdots \wedge (Z_n(t) \Rightarrow x_n) \wedge \\ & \qquad (x_1 \Rightarrow B_1(t)) \wedge \cdots \wedge (x_n \Rightarrow B_n(t)) \;. \end{aligned} \tag{4.25}$$

Damit haben wir die Grundlagen geschaffen, die zu einer Formalisierung verschiedener dynamischer Konsistenzkriterien erforderlich sind. Neben den bereits vorgestellten statischen Tests, die auch für die Erweiterung um zeitliche Intervalle gültig bleiben, und also auch für die Produktion von Interesse sind, ergeben sich im dynamischen Fall weitere mögliche Konsistenztests. Typische Fragestellungen und deren Formalisierung als aussagenlogisches Erfüllbarkeitsproblem sind dabei:

Induzierte Änderungen auf Teileebene: Welchen Effekt haben Änderungen der Produktübersicht auf die Teileebene? Dieser Effekt besteht zum einen aus zusätzlich benötigten Teilen, zum anderen aus überflüssig werdenden. Wir werden im nächsten Abschnitt ausführlich auf die Formalisierung und Behandlung dieser Fragestellung eingehen.

Zusammenfassung von Produktänderungen: Welche Aufträge werden im Laufe eines Zeitabschnitts ungültig oder neu gültig? Für die Formalisierung dieser Fragestellung nehmen wir zwei Zeitpunkte t_0 und t_1 mit $t_0 < t_1$ an. Dann sind die Modelle der Formeln $\mathcal{B}(t_1) \wedge \neg\mathcal{B}(t_0)$ bzw. $\mathcal{B}(t_0) \wedge \neg\mathcal{B}(t_1)$ die neu baubar werdenden bzw. nicht länger baubaren Aufträge.

Zeitintervalle ohne baubare Aufträge: Gibt es einen Zeitraum, während dessen es – bezogen auf die Dokumentation – keine baubaren Produkte (meist mit einer bestimmten Zusatzeigenschaft) gibt? Nehmen wir an, dass sich die einschränkende Zusatzeigenschaft als eine Formeleinschränkung F an die Aufträge angeben lässt, so ergeben sich die Zeiten T_F, zu denen keine Aufträge mit Eigenschaft F baubar sind durch

$$T_F \;=\; \{t \mid \mathcal{B}(t) \wedge F \text{ ist nicht erfüllbar}\}.$$

Die Berechnung dieser Menge kann bewerkstelligt werden, indem zuerst alle relevanten Start- und Stoppzeiten

$$T_R \;=\; \{t_\alpha(R), t_\omega(R) \mid R \text{ ist Regel der Produktübersicht}\}$$

aus der Dokumentation extrahiert werden. Ordnung dieser Menge, so dass $T_R = \{t_0, \ldots, t_k\}$ mit $t_i < t_{i+1}$, erlaubt dann die Durchführung von k Erfüllbarkeitstests der Form $\mathcal{B}(t) \wedge F$ für alle Zeitpunkte $t_i^* = \frac{1}{2}(t_i + t_{i+1})$ und $0 \leq i < k$. Das Ergebnis der Stichprobe zum Zeitpunkt t_i^* ist dann für das gesamte Intervall $[t_i, t_{i+1})$ gültig.

4.2.3 Zwei Methoden zur Erkennung induzierter Änderungen auf Teileebene

Um das Hauptproblem der dynamischen Konsistenzprüfung, die Feststellung der Auswirkungen von Änderungen der Produktübersicht auf den Teilebedarf, zu behandeln, schlagen wir zwei Methoden vor, die wir nun ausführlich vorstellen möchten (siehe auch [SK01]). Beiden Methoden gemein ist die Analyse von Änderungen durch den Vergleich zweier oder mehrerer unterschiedlicher Produktdokumentations-Varianten zu unterschiedlichen Stichtagen.

Die $\pm\delta$-Methode. Mit der $\pm\delta$-Methode können Teile (Positionsvarianten) bestimmt werden, die durch eine zeitlich relativ exakt lokalisierte Änderung der Produktdokumentation entweder überflüssig oder zusätzlich benötigt werden. Alle relevanten Änderungen müssen dabei in einem kurzen Zeitintervall $(t_c - \delta, t_c + \delta)$ um einen festgelegten Zeitpunkt t_c herum erfolgen. Wir verwenden die Bezeichnung LCRegel(p, t) für die zeitabhängige lange Coderegel von Teil p zum Zeitpunkt t, und außerdem zur Verkürzung der Schreibweise die Formel $F(p, t) = \mathcal{B}(t) \wedge \text{LCRegel}(p, t)$ für ein Teil $p \in \mathcal{T}$ und einen Zeitpunkt t. Die $\pm\delta$-Methode besteht dann aus drei Schritten:

Schritt 1: Bestimme die Menge P_1 der Teile, die direkt vor t_c benötigt werden:

$$P_1 = \{p \in \mathcal{T} \mid F(p, t_c - \delta) \text{ ist erfüllbar.}\}$$

Schritt 2: Bestimme die Menge P_2 der Teile, die direkt nach t_c benötigt werden:

$$P_2 = \{p \in \mathcal{T} \mid F(p, t_c + \delta) \text{ ist erfüllbar.}\}$$

Schritt 3: Berechne die Differenzmengen $U = P_1 \setminus P_2$ und $Z = P_2 \setminus P_1$.

Die sich ergebenden Mengen U und Z geben die durch die betrachteten Änderungen überflüssig werdenden bzw. zusätzlich benötigten Teile an. Der Parameter δ muss derart gewählt werden, dass nur die relevanten Änderungen im Zeitintervall $t_c \pm \delta$ liegen. Dies ist – zumindest theoretisch – ein einschränkender Faktor bei der Verwendung der $\pm\delta$-Methode.

Die 3-Punkt-Methode. Im Unterschied zur $\pm\delta$-Methode erlaubt die 3-Punkt-Methode sowohl die Behandlung von in DIALOG dokumentierten als auch von (noch) nicht in DIALOG dokumentierten oder nicht dokumentierbaren Änderungen. Dies erlaubt unter anderem die Simulation von möglichen zukünftigen Änderungen ohne direkt die betriebswichtigen DIALOG-Daten zu verändern. Die Spezifikation von DIALOG-externen Änderungen wird durch einen einfachen Zusatzmechanismus erreicht. Durch diesen werden die folgenden Änderungen formulierbar:

- An- und Auslauf von Ausstattungs- oder Kontrollcodes,
- Ungültigwerden beliebiger Codekombinationen, wie zur Simulation von Produktionsverlagerungen notwendig.

Konsistenztests können dann auf den so veränderten Daten ablaufen, die entsprechenden Änderungen noch vor deren eigentlicher Dokumentation berücksichtigend. Der Zusatzmechanismus besteht darin, Modifikationen an der aussagenlogischen Produktübersichtsformel $\mathcal{B}(t)$ vorzunehmen. Dazu definieren wir eine modifizierte, durch eine Indexmenge $I \subseteq \{1, \ldots, n\}$ und eine Einschränkungsformel E parametrisierte Baubarkeitsformel

$$\mathcal{B}^*(I, E, t) \ := \ E \wedge \bigwedge_{i \in \{1,\ldots,n\} \setminus I} \big((Z_i(t) \Rightarrow x_i) \wedge (x_i \Rightarrow B_i(t))\big) \ .$$

Mit dieser Baubarkeitsformel lassen sich bestehende Zusteuerungs- und Baubarkeitsbedingungen für einen Code x_i ausblenden, indem man i zur Indexmenge I hinzufügt. Neu hinzukommende Bedingungen lassen sich dann als Teil der Einschränkungsformel E spezifizieren. Soll also z.B die

Änderung der Baubarkeitsformeln von Code x_{i^*} von B_{i^*} nach B_N unter Beibehalt der Zusteuerungsformel von x_{i^*} simuliert werden, so fügt man i^* zu I hinzu und erweitert E konjunktiv um $(Z_i(t) \Rightarrow x_i) \wedge (x_i \Rightarrow B_N)$. Das Gültigwerden eines bisher gesperrten Codes (d.h. eines Codes mit $B_i \equiv \bot$) kann analog vorgenommen werden; das Ungültigwerden eines Codes x_i wird durch den Zusatz von $\neg x_i$ zu E erreicht. Für das Invalidieren ganzer Codekombinationen werden diese ebenfalls negiert der Formel E hinzugefügt.

Die 3-Punkt-Methode baut auf dieser Möglichkeit der DIALOG-externen Datenmodifizierung auf. Gegeben zwei Zeitpunkte t_0 und t_1, zwischen denen die zu simulierende Änderung durchgeführt werden soll, und eine modifizierte Baubarkeitsformel $\mathcal{B}^*(I, E, t_1)$ für den Zeitpunkt t_1, die die beabsichtigten Änderungen zusammenfasst, kann die aus vier Schritten bestehende 3-Punkt-Methode zur Berechnung verschiedener Teilemengen (zusätzlich benötigt, nicht mehr benötigt) angewandt werden:

Schritt 1: Bestimme die Menge P_{t_0} der Teile, die zum Zeitpunkt t_0 benötigt werden:

$$P_{t_0} = \{p \in \mathcal{T} \mid \big(\mathcal{B}(t_0) \wedge \text{LCRegel}(p, t_0)\big) \text{ ist erfüllbar.}\}$$

Schritt 2: Bestimme die Menge P_{t_1} der Teile, die ohne die simulierte Änderung zum Zeitpunkt t_1 benötigt werden:

$$P_{t_1} = \{p \in \mathcal{T} \mid \big(\mathcal{B}(t_1) \wedge \text{LCRegel}(p, t_1)\big) \text{ ist erfüllbar.}\}$$

Schritt 3: Bestimme die Menge $P^*_{t_1}$ der Teile, die mit simulierter Änderung zum Zeitpunkt t_1 benötigt werden:

$$P^*_{t_1} = \{p \in \mathcal{T} \mid \big(\mathcal{B}^*(I, E, t_1) \wedge \text{LCRegel}(p, t_1)\big) \text{ ist erfüllbar.}\}$$

Schritt 4: Berechne die Differenzmengen

$$\begin{array}{ll} Z_{10} = P_{t_1} \setminus P_{t_0} & U_{10} = P_{t_0} \setminus P_{t_1}, \\ Z_{*0} = P^*_{t_1} \setminus P_{t_0} & U_{*0} = P_{t_0} \setminus P^*_{t_1}, \\ Z_{*1} = P^*_{t_1} \setminus P_{t_1} & U_{*1} = P_{t_1} \setminus P^*_{t_1} \quad . \end{array}$$

Bei den Ergebnismengen bezeichnet Z_{10} beispielsweise die Menge der zusätzlich benötigten Teile zum Zeitpunkt t_1 im Vergleich zu t_0, wobei die zu

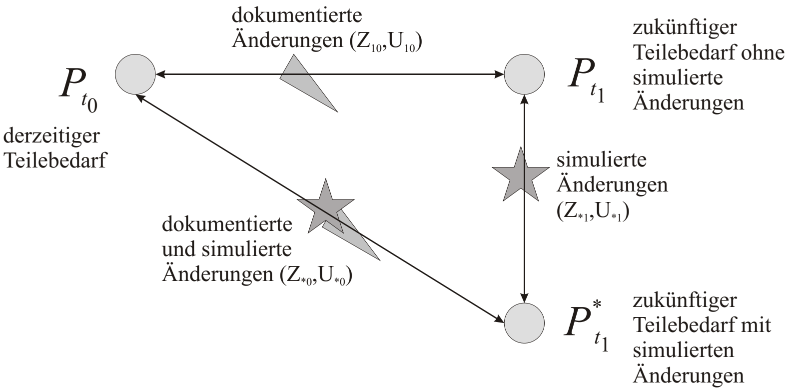


Abbildung 4.1: Die 3-Punkt-Methode.

simulierenden Änderungen nicht mitberücksichtigt werden. Die Beziehungen zwischen den drei Teilemengen und den berechneten Differenzmengen sind in Abbildung 4.1 grafisch schematisiert.

Um nun die Auswirkungen einer beabsichtigten Dokumentationsänderung auf den Teilebedarf abzusehen, können verschiedene Differenzmengen betrachtet werden. Die Mengen Z_{*0}/U_{*0} zeigen die vollständige Teileänderung zwischen t_0 und t_1 an, verursacht sowohl durch bereits dokumentierte Änderungen an der Produktübersicht und den Teilen selbst, als auch durch die zu simulierende Änderung. Unter Betrachtung der Differenzmengen Z_{*1}/U_{*1} erhält man lediglich die durch die simulierte Änderung entstehenden Teilebedarfsdifferenzen. Außerdem erhält man, ähnlich der $\pm\delta$-Methode, den Einfluss der bereits dokumentierten Änderungen im Zeitintervall (t_0, t_1) auf den Teilebedarf durch die Mengen Z_{10}/U_{10}.

4.2.4 Experimentelle Ergebnisse

Wir haben mit unserem BIS-System Experimente mit DIALOG-Daten aus der Produktion, die auch die erforderlichen zeitlichen Zusatzdaten enthalten, durchgeführt. Laufzeiten und Suchraumgrößen der Beweiserläufe sind vergleichbar zu den bereits angegebenen Untersuchungen der Teileliste aus Abschnitt 4.1.4, so dass wir auf eine Aufstellung dieser Größen hier verzichten wollen.

Die exemplarischen Ergebnisse der kompletten Untersuchung einer Fahrzeugbaureihe anhand der 3-Punkt-Methode sind in Anhang B aufgeführt.

Kompilierung und Komplexität

5

Zwei Eigenschaften haben sich im letzten Kapitel als charakteristisch für die Verifikation der Mercedes-Produktdaten herausgestellt: Die insgesamt große Anzahl an durchzuführenden Erfüllbarkeitstests und die erstaunlich geringe Laufzeit jedes einzelnen dieser Tests. Erstere Eigenschaft erschwert eine effiziente Verarbeitung, letztere erleichtert sie. Wir wollen diese beiden Charakteristika nun näher untersuchen, um aus den sich ergebenden Erkenntnissen speziell auf unsere Anwendung angepasste Algorithmen entwickeln zu können.

Zur Feststellung der Komplexität werden wir verschiedene in der Literatur untersuchte handhabbare (*tractable*) Teilklassen des SAT-Problems betrachten und einen neuen Ansatz basierend auf *aktiven Literalen* vorstellen. Dieser ist eng mit der Idee einer Kompilierung der Daten verknüpft, welche bei der Vielzahl der durchzuführenden Tests Vorteile durch eine verkürzte amortisierte Laufzeit verspricht.

5.1 Grundlegende Begriffe

Wir werden in diesem Kapitel häufig Formeln in Klauseldarstellung verwenden. Daher wollen wir zuerst einige dazu notwendige Begriffe klären.

Gegeben sei eine endliche Menge $\Phi_0 = \{x_1, \ldots x_n\}$ von aussagenlogischen Variablen. Eine Variable x_i oder deren Negat $\neg x_i$ bezeichnen wir als *Literal*, und weiter mit $L := \Phi_0 \cup \neg\Phi_0$ die Menge aller Literale, wobei $\neg\Phi_0 := \{\neg v \mid v \in \Phi_0\}$. Nicht-negierte Literale $l \in \Phi_0$ heißen auch *positive* Literale, negierte Literale $l' \in \neg\Phi_0$ auch *negative*. Für die Negation $\neg l$ eines Literals schreiben wir auch $\bar{l}$. Dabei identifizieren wir für ein negatives Literal $l = \neg l'$ den Ausdruck $\bar{l}$ mit l'. Das Literal $\bar{l}$ heißt auch zu l *komplementär*. Eine (endliche) Menge von Literalen heißt *Klausel*. Wir werden Klauseln anstatt mit Mengenklammern häufig auch mit runden Klammern schreiben. Eine Klausel mit k Literalen wird k-Klausel genannt, im besonderen Fall von $k = 1$ auch *Einheits- oder Unit-Klausel*; die 0-Klausel wird auch als *leere Klausel* (in Zeichen: $\square$) bezeichnet. Eine Klausel, die nur positive (bzw. nur negative) Literale enthält, heißt *positive* (bzw. *negative*) *Klausel*. *Horn-Klauseln* sind Klauseln mit höchstens einem positiven Literal. Für eine Menge S von Klausen bezeichen wir die darin vorkommenden Variablen mit $\mathrm{Var}(S)$; analog bezeichne $\mathrm{Var}(c)$ und $\mathrm{Var}(l)$ die Menge der in einer Klausel c vorkommenden bzw. die einem Literal l zugeordnete Variable. Eine Klausel c ist eine (echte) Teilklausel der Klausel c', falls $c \subseteq c'$ ($c \subset c'$). Falls c eine Teilklausel von c' ist, so sagt man auch c *subsumiert* c'. Für eine Klauselmenge S bezeichnen wir mit $S|_l := \{c \setminus \{\bar{l}\} \mid c \in S, l \notin c\}$ die vereinfachte Klauselmenge, die dem Setzen von l auf $\mathbf{true}$ in S entspricht.

Die logische Interpretation einer Klausel ist die Disjunktion ihrer Literale. Eine Menge von Klauseln wird logisch als Konjunktion der enthaltenen Klauseln verstanden, so dass eine Klauselmenge direkt einer Formel in CNF entspricht. Eine Klauselmenge $S = \{c_1, \ldots, c_k\}$ kann somit auch als (aussagenlogische) Theorie $\mathcal{T}$ (in CNF) verstanden werden. Eine Funktion $\beta : \Phi_0 \mapsto \{\mathbf{true}, \mathbf{false}\}$ heißt *Variablenbelegung*. Jede Variablenbelegung kann eindeutig zu einer Bewertungsfunktion beliebiger Formeln homomorph erweitert werden (mit der üblichen Interpretation der Negation, Disjunktion und Konjunktion). Eine Variablenbelegung ist *Modell* einer Formel F in CNF, falls F unter β zu $\mathbf{true}$ evaluiert. Eine Formel, die kein Modell besitzt, heißt *unerfüllbar* oder *inkonsistent*. Eine Klausel c ist eine *Konsequenz* (oder *Folgerung* aus) einer Theorie $\mathcal{T}$, geschrieben als $\mathcal{T} \models c$, falls alle Modelle von $\mathcal{T}$ auch Modelle von c sind.

Für zwei Klauseln c und d, die ein komplementäres Literalpaar $(l, \bar{l})$ enthalten, also $c = c' \dot{\cup} \{l\}$ und $d = d' \dot{\cup} \{\bar{l}\}$ nennen wir die Klausel $e = (c' \setminus \{l\}) \cup (d' \setminus \{\bar{l}\})$ eine *Resolvente* von c und d, und sagen auch, dass e aus c und d durch *Resolution* (über l oder $\bar{l}$) entstanden ist. Man verwendet oft auch die grafische (Deduktionsregel-)Notation

$$\frac{c \quad d}{e}\ \mathrm{Res}$$

um anzuzeigen, dass e aus c und d durch Resolution entstanden ist. Eine Resolvente heißt *trivial*, falls sie ein komplementäres Literalpaar enthält (und somit eine Tautologie ist). Für zwei Klauselmengen S und T bezeichnen wir mit $\mathrm{Res}(S, T)$ die Menge aller nicht-trivialen Resolventen, die durch Resolution zweier Klauseln $c \in S$ und $d \in T$ gebildet werden können.

5.2 Komplexität

Das aussagenlogische Erfüllbarkeitsproblem (SAT), mit dem wir bei all unseren Konsistenztests zu tun haben, war das erste Problem, dessen NP-Vollständigkeit nachgewiesen werden konnte [Coo71]. Aus der NP-Vollständigkeit folgt, dass der Test auf Erfüllbarkeit einer Formel F im schlimmsten Fall nur in exponentieller Zeit (in der Größe der Formel) durchzuführen ist (sofern nicht P=NP). Auch 3SAT, das Erfüllbarkeitsproblem für Formeln in CNF mit höchstens drei Literalen pro Klausel, ist NP-vollständig.

Bestimmte Teilklassen des SAT-Problems weisen jedoch eine bessere *worst-case*-Komplexität auf. Unter den bekanntesten sind die Einschränkungen auf Horn-Klauseln oder 2-Klauseln (Krom-Klauseln), die beide in linearer Zeit lösbar sind. Weitere in polynomieller Zeit entscheidbare Teilklassen sind:

Renamable Horn [HW74]: Eine Formel dieser Problemklasse ist durch Umbenennen von Variablen in eine Horn-Formel transformierbar. Dabei werden bei einer Umbenennung (bezüglich einer Variablenmenge $V \subseteq \mathrm{Var}(F)$) alle Literale $l \in L(V) := V \cup \neg V$ in F mit ihren Negaten vertauscht und umgekehrt.

Extended Horn [CH91]: Diese Teilklassse basiert auf einer Verallgemeinerung des Horn-Klausel-Begriffs. Sie ist über eine zusätzliche Graphstruktur A definiert, die als Flussgraph interpretiert wird. A muss dabei eine Arboreszenz (mit Wurzel W) sein, in der jede Kante mit einer unterschiedlichen Variablen des Problems beschriftet ist. Eine Klausel C ist dann eine erweiterte (extended) Horn-Klausel bezüglich A, falls die positiven Literale von C einen (möglicherweise leeren) Pfad in A bilden und die negativen Literale weiteren Eigenschaften genügen (siehe z.B. [FG03]). Existiert eine Struktur A, so dass jede Klausel einer Menge S eine erweiterte Horn-Klausel ist, so gehört S zu der Problemklasse *extended Horn*. Es gibt für diese Problemklasse auch eine einfache Charakterisierung mittels Integer-Linear-Programmen, wobei die Lösungen immer durch Rundung der reellen Lösungen der (komplexitätstheoretisch einfacheren) Relaxation des Problems erhalten werden können. Es ist bisher weder ein polynomieller Algorithmus bekannt, der zu einer gegebenen Formel F eine passende Struktur A findet, noch ein polynomielles Prüfverfahren, das feststellt, ob F zur Klasse *extended Horn* gehört oder nicht.

q-Horn [BCH90]: Die Klasse q-Horn ist als Vereinigung über q-Horn-Klauselmengen bezüglich sogenannter “Musterklauseln” (pattern) P definiert. Für eine gegebene Musterklausel P ist eine Klauselmenge S eine q-Horn-Klauselmenge zu P, geschrieben $S \in Q(P)$, falls für alle Klauseln $C \in S$ entweder $|C \setminus P| \leq 1$ oder $|C \setminus P| = 2$ zusammen mit $C \cap \neg P = \emptyset$ gilt. Die Klasse q-Horn ist dann definiert als die Vereinigung der $Q(P)$ über alle Klauseln P. Zum Beispiel sind für $P = \Box$ die Elemente von $Q(P)$ alle Klauselmengen, die nur 1- oder 2-Klauseln enthalten. Für $P = \neg\Phi_0$ erhält man für $Q(P)$ die Menge alle Hornklauselmengen. Sowohl Prüfung der Zugehörigkeit zu q-Horn als auch das Erfüllbarkeitsproblem für Formeln in q-Horn ist in linearer Zeit lösbar [Tru98].

Balancierte Formeln [CC92]: Diese sind dadurch charakterisiert, dass in der Formulierung als Integer-Linear-Porgramm alle Untermatrizen mit genau zwei von Null verschiedenen Einträgen pro Zeile und Spalte eine durch vier teilbare Summe ihrer Einträge besitzen. Für diesen Fall ist bekannt, dass die Koordinaten der Eckpunkte des den Lösungsraum beschreibenden Polytops entweder ganzzahlig sind oder der Lösungsraum leer ist. In der Klauseldarstellung ergibt sich ebenfalls eine einfache Charakterisierung dieser Problemklasse: Eine erfüllbare Klauselmenge S gehört zur Menge der balancierten Formeln, falls jede Klausel mindestens zwei Literale besitzt, und für jede Variable $x_i \in \mathrm{Var}(S)$ auch die erweiterten Klauselmengen $S \cup \{(x_i)\}$ und $S \cup \{(\neg x_i)\}$ erfüllbar sind. Für unerfüllbare Klauselmengen besteht keine Einschränkung. Gerade in der Konfiguration ist diese Eigenschaft für die aussagenlogische Codierung der Produktdaten häufig gegeben (vgl. notwendige/unzulässige Codes).

Die Klassen *renamable Horn* und *(renamable) extended Horn* sowie die balancierten Formeln können alle mittels einfacher Unit-Propagation entschieden werden [dV00]. Kullmanns Konzept der linearen Autarkien [Kul00] und der zugehörige Begriff der linearen Erfüllbarkeit umfasst die Klassen der q-Horn-Klauseln – damit auch der Horn- und Krom-Klauseln – sowie Franco und Van Gelders “matched clause sets” [FG03], und liefert so ebenfalls ein handhabbares (*tractable*) Entscheidungsverfahren für diese Problemklassen. Pretolani entwickelt aufbauend auf den beiden Klassen Horn und Krom unter Verwendung der Unit-Propagation eine Hierarchie polynomiell lösbarer Teilklassen [Pre96]. Man kann diese auch als die von einem modifizierten Davis-Putnam-Algorithmus (siehe Kapitel 6) mit beschränkter Suchtiefe entscheidbaren Probleme charakterisieren.

Eine genauere Beschreibung und eine weiterführende Übersicht verschiedener handhabbarer Klassen ist den Artikeln von del Val [dV00] und Franco und Van Gelder [FG03] zu entnehmen. Kleine Büning und Lettmann geben eine schöne Übersicht über eine Vielzahl (meist NP-vollständiger) aussagenlogischer Probleme und deren Komplexität [KBL94].

BR/AA	n	k	%$\mathrm{H}_{\max}$	%$\mathrm{P}_{\min}$
C168/FW	653	4299	76.60	0.0000
C170/FR	554	7579	69.81	0.0001
C202/FS	698	4954	73.90	0.0000
C202/FW	734	7290	81.06	0.0000
C208/FA	633	3720	80.32	0.0000
C208/FC	652	3975	79.52	0.0000
C210/FS	723	4025	77.86	0.0002
C210/FW	775	5491	79.80	0.0000
C220/FV	597	3368	69.69	0.0000
C638/FKA	528	4966	95.91	0.0000

Tabelle 5.1: Syntaktisch-kombinatorische Größen für einige Mercedes Baureihen.

Einfache Experimente zur Untersuchung einiger statischer syntaktisch-kombinatorischer Größen der Mercedes-Fahrzeugbaureihen haben die in Tabelle 5.1 dargestellten Ergebnisse gebracht. Dort sind für ausgewählte Baureihen/Ausführungsarten neben der bereits in Tabelle 4.2 aufgeführten Anzahl der Variablen (n) und Klauseln (k) der (durch Unit-Propagation reduzierten) Baubarkeits- und Zusteuerungsformel $\mathcal{B}$ auch zwei weitere Größen, %$\mathrm{H}_{\max}$ und %$\mathrm{P}_{\min}$, angegeben. Erstere bezeichnet die maximale relative Anzahl der Hornklauseln, die sich durch Umbenennung (s.o.) von Literalen mittels eines einfachen heuristischen Maximierungsalgorithmus ergeben. Diese Zahlen bewegen sich (bis auf eine Ausnahme) im Bereich von circa 70 bis 80 Prozent, was mit anderen Probleminstanzen vergleichbar ist und somit keine Besonderheit der Fahrzeugkonfigurationsdaten darstellt. Anders verhält es sich bei der zweiten Größe, %$\mathrm{P}_{\min}$, welche die minimale Anzahl von positiven Klauseln angibt, die durch Umbenennung erzielbar ist. In allen Fällen bleibt höchstens eine positive Klausel übrig, oft sogar gar keine. In letzterem Fall ist damit die Erfüllbarkeit der Basisformel $\mathcal{B}$ direkt nachgewiesen. Für Erfüllbarkeitstests der Form $\mathcal{B} \wedge F$ lässt sich dies aber nur bedingt verwerten, da durch das Hinzufügen neuer positiver Klauseln und nachfolgender Unit-Propagation weitere positive Klauseln aus der Reduktion von $\mathcal{B}$ entstehen können. Trotzdem wäre diese Eigenschaft wohl eine eingehendere Untersuchung wert.

5.3 Strukturelle Analyse

Neben den syntaktisch-kombinatorischen, handhabbaren Teilklassen des letzten Abschnitts besteht eine alternative Methode um potentiell einfache Probleme zu identifizieren darin, die interne Struktur der Probleminstanz zu untersuchen. Während die oft zu theoretischen Untersuchungen herangezogenen schweren Zufallsprobleme (3SAT mit einem festen Klausel/Variablen-Verhältnis k nahe dem Phasenübergangspunkt von ungefähr $k = 4.25$ [GW94]) kaum eine erkennbare Struktur aufweisen, sind viele Codierungen von Anwendungsproblemen (aus der Planung, Konfiguration, Hardware-Verifikation etc.) mit einer internen Struktur behaftet, die sich in deutlich einfacheren Erfüllbarkeitsproblemen niederschlagen kann.

Zur Identifikation und Nutzung dieser internen Struktur sind in der Literatur verschiedene Verfahren vorgeschlagen worden (siehe z.B. [Dan83, DP89, PVG00]). Die meisten beruhen auf einer Analyse der Abhängigkeiten zwischen den Problemvariablen. Anhand dieser Abhängigkeiten wird ein Graph aufgebaut, dessen Knoten die Variablen des Problems sind und dessen Kanten Variablen verbinden, die zusammen in mindestens einer Klausel auftreten. Dieser Graph wird von Dantsin "Variablengraph" (*graph of variables*), von Rish und Dechter "Interaktionsgraph" (*interaction graph*) genannt [RD00]. In anderen Artikeln wird eine Hypergraph-Struktur mit analoger Konstruktion verwendet [GLP93]. Park und Van Gelder argumentieren für die Verwendung des dualen Graphen, in dem die Knoten dann Klauseln entsprechen, und Kanten Klauseln mit gemeinsamen Variablen (oder komplementären Literalen) verbinden [PVG00]. Wir werden der Vorgehensweise von Rish und Dechter folgen und daher auch deren Begriff des Interaktionsgraphen verwenden.

Zur Analyse der Struktur der Fahrzeug-Produktdaten haben wir die aussagenlogischen Formeln $\mathcal{B}$ aus Abschnitt 4.1.4, die sich aus den kombinierten Baubarkeits- und Zusteuerungsformeln einer Baureihe ergeben, einer solchen Graphen-basierten Untersuchung unterzogen. Der Interaktionsgraph der Struktur der Gesamtformel ist jedoch recht unübersichtlich, so dass wir uns zur weiteren Untersuchung im Wesentlichen auf Restriktionen der Gesamtformel $\mathcal{B}$ beschränken wollen. Abbildung 5.1 sind Größenangaben zu den sich durch verschiedene Restriktionen ergebenden Formeln für die Modellklasse C202/FW zu entnehmen. Der Wurzelknoten der Baumstruktur stellt die Gesamtformel $\mathcal{B}$ dar, deren Variablen- und Klauselanzahl neben dem Knoten vermerkt ist. Die vier Kindknoten des Wurzelknotens repräsentieren eine Grobklassifizierung der Fahrzeugvarianten in vier Teilklassen.

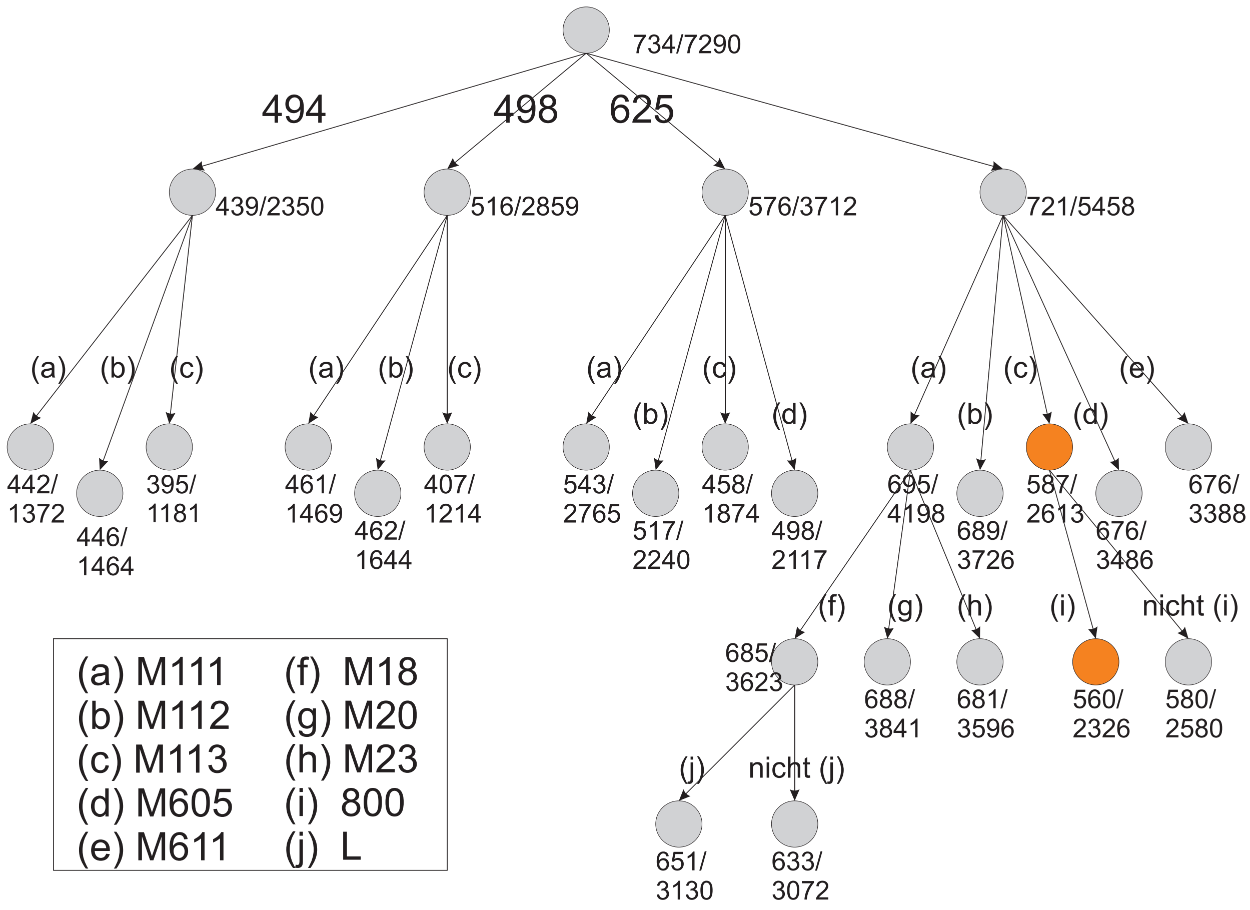


Abbildung 5.1: Problemreduktion bei Restriktionen der Modellklasse C202/FW.

Deren erste, mit dem Code 494 beschriftet, kennzeichnet USA-Fahrzeuge, die im Vergleich zu Fahrzeugen anderer Länder mit deutlich abweichenden Restriktionen belegt sind. Die nächsten beiden Unterklassen betreffen Fahrzeuge, die nach Japan (498) und Australien (625) geliefert werden. Der letzte Zweig, mit $\emptyset$ markiert, enthält alle verbleibenden Fahrzeuge, also die Varianten, in denen weder Code 494, noch 498, noch 625 vorkommt, für die also $\neg x_{494} \wedge \neg x_{498} \wedge \neg x_{625}$ gilt. Da die Codes 494, 498 und 625 sich gegenseitig ausschließen, sind mit diesen vier Varianten alle möglichen Fälle abgedeckt. Es ist durchaus bemerkenswert, dass die Unit-Propagation zu einer überraschend deutlichen Verkleinerung bei den Baubarkeitsformeln der Restriktionen führt. Für die Teilklasse 494 der USA-Fahrzeuge z.B. bleiben nach Unit-Propagation nur 439 der ursprünglich 734 Variablen und 2350 der ursprünglich 7290 Klauseln übrig. Ähnlich deutliche Reduktionen ergeben sich in den anderen Fällen, am wenigsten deutlich im letzten Fall der in gewissem Sinne allgemeinsten Teilklasse $\emptyset$. Eine weitere Aufspaltung hinsichtlich der gewählten Motorisierung in (je nach vorhergehender länderspezifischer Restriktion) drei bis fünf Varianten ergibt eine weitere Reduktion der Baubarkeitsformel, bis hinunter zu 395 Variablen und 1181 Klauseln für die USA-Fahrzeuge (494) mit V8-Ottomotor (M113).

Alleine diese zweistufige Aufteilung in spezialisiertere Varianten bringt also eine bis zu 46%-ige Reduktion in der Anzahl der Variablen und 84%-ige Reduktion in der Anzahl der Klauseln. Von den bei acht Variablen (494, 498, 625, M111, M112, M113, M605, M611) möglichen 256 Varianten sind auf dieser zweiten Ebene lediglich 15 vertreten, alle anderen sind aufgrund einfacher Ausschlüsse nicht möglich, so dass sich dadurch eine weitere Problemvereinfachung ergibt. Es sei noch angemerkt, dass die acht gewählten Variablen zu den in der Baubarkeitsformel $\mathcal{B}$ am häufigsten auftretenden gehören: Code 498 kommt z.B. in 18.2%, Code M113 in 12.4% aller Klauseln vor.

Zur Veranschaulichung der Struktur der eingeschränkten Baubarkeitsformeln haben wir die Interaktionsgraphen für die beiden in Abbildung 5.1 orange markierten Fälle in den Abbildungen 5.2, 5.3 und 5.4 dargestellt. In Abbildung 5.2 ist die Einschränkung auf M113, bei Nichtvorhandensein der Codes 494, 498 und 625, dargestellt. Der Übersichtlichkeit halber sind Ländercodes (mit L endend) nicht berücksichtigt; Ausstattungscodes (mit A endend) sind als gelb markierte, für Bedienungsanleitungen und Hinweisschilder stehende Codes (mit B endend) als orange markierte Knoten dargestellt. Eine Kante zwischen zwei Knoten bedeutet, dass es eine Klausel gibt, in der die Codes der beiden Knoten gemeinsam vorkommen. Ände-

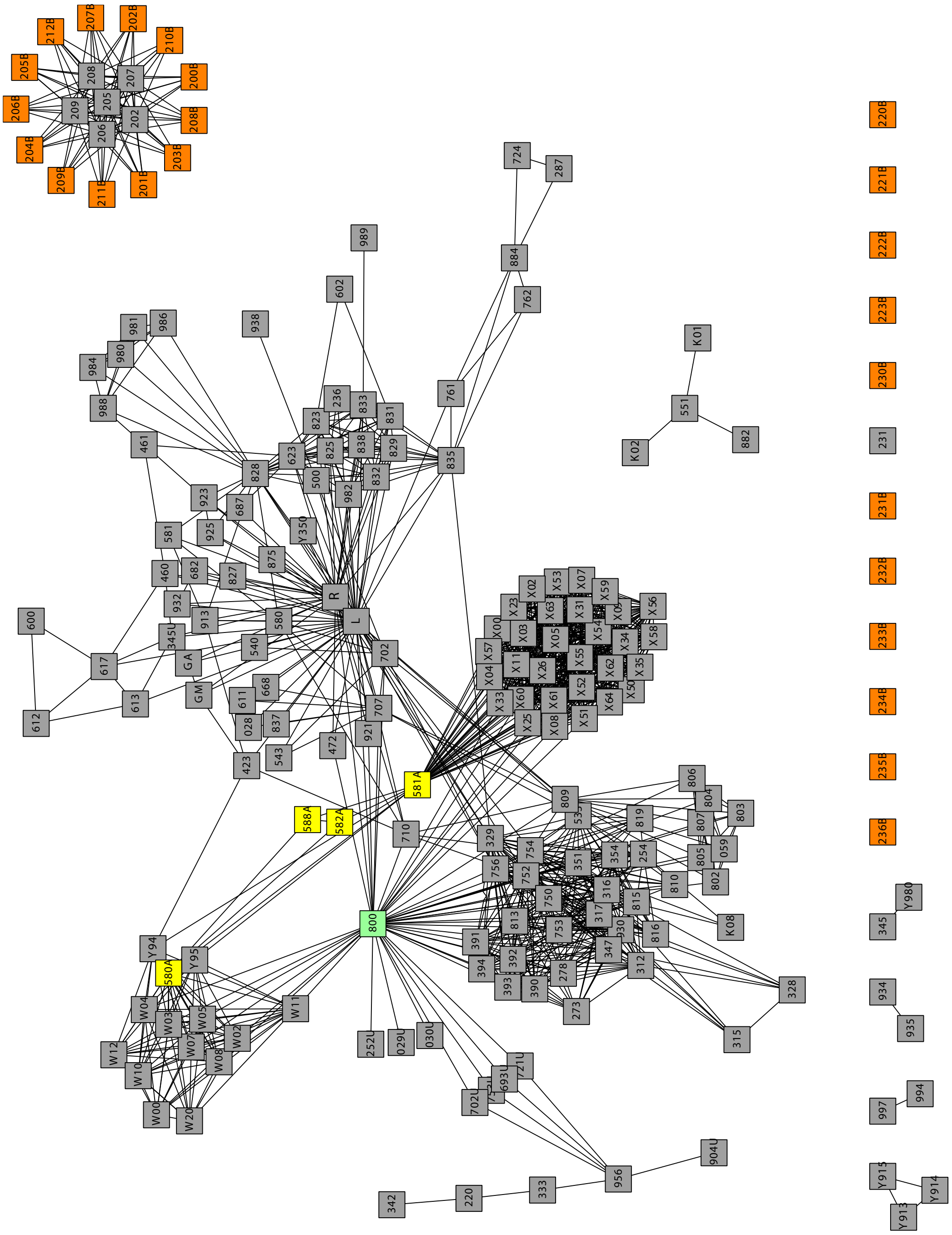


Abbildung 5.2: Konfigurations-Abhängigkeiten der Modellklasse C202/FW (I): Einschränkung auf M113, ohne Ländercodes.

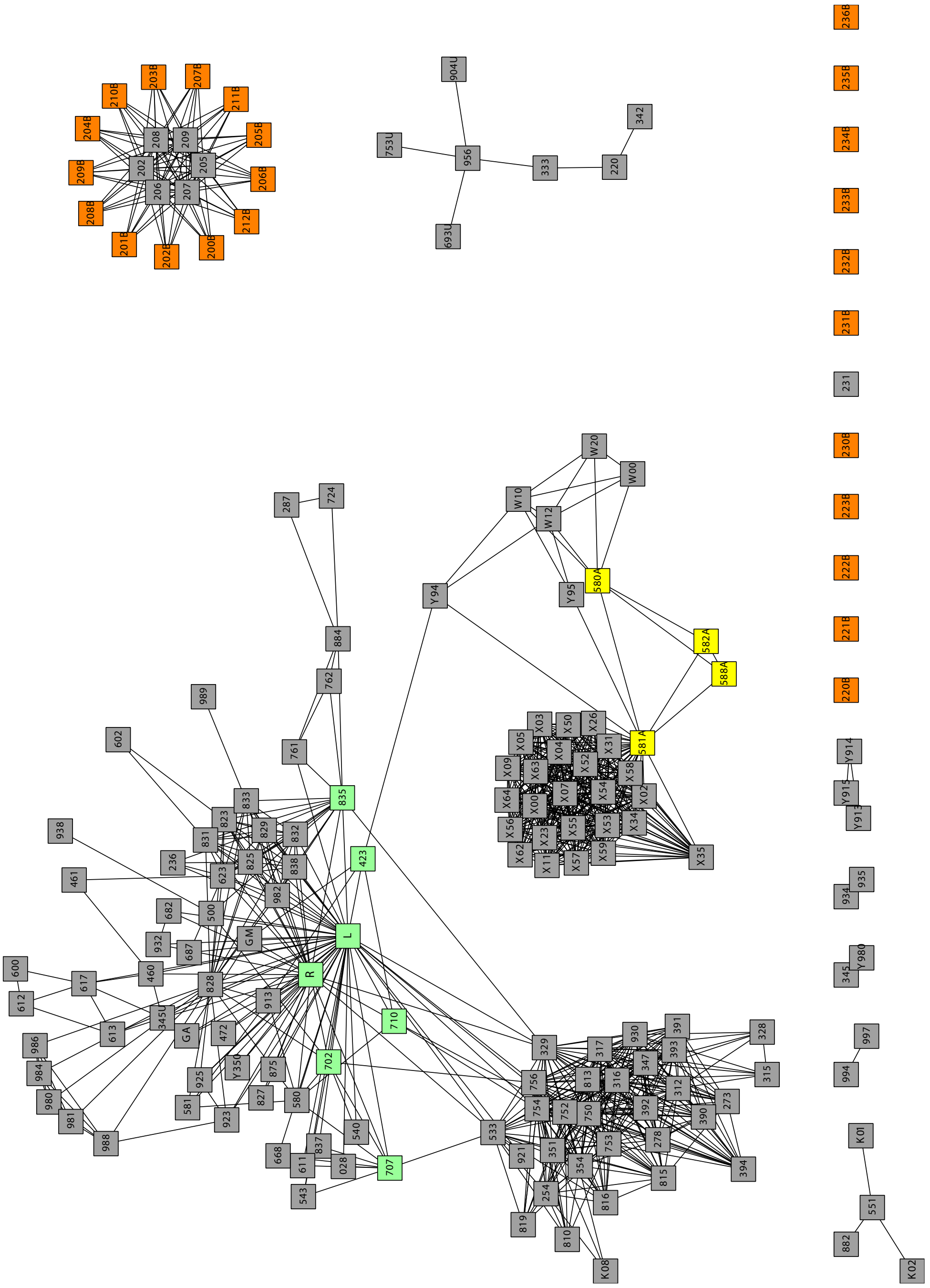


Abbildung 5.3: Konfigurations-Abhängigkeiten der Modellklasse C202/FW (II): Einschränung auf M113 und 800, ohne Ländercodes.

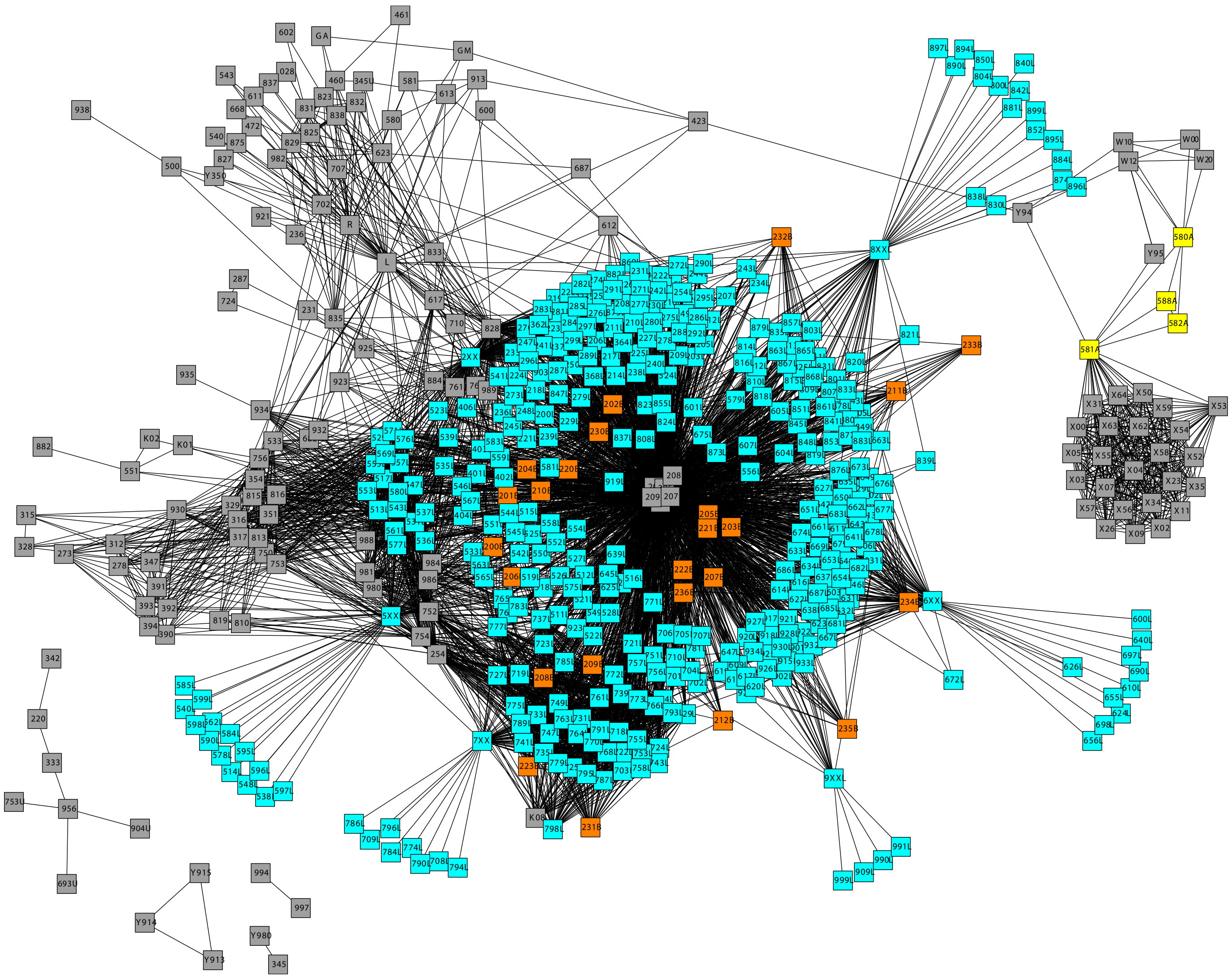


Abbildung 5.4: Konfigurations-Abhängigkeiten der Modellklasse C202/FW (III): Einschränkung auf M113 und 800, inklusive Ländercodes.

rung eines durch einen Knoten repräsentierten Code in einer Konfiguration kann also Auswirkungen auf alle benachbarten Knoten haben und demnach eine Umkonfiguration dieser erfordern.

Man erkennt, dass die Kanten kein regelmäßiges, gleichverteiltes Muster aufweisen, sondern dass Verdichtungen und mehr oder weniger stark zusammenhängende Codegruppen existieren. So haben beispielsweise Änderungen (Setzen oder Löschen) der Codes L, R, 800 oder 581A Auswirkungen auf eine Vielzahl anderer Codes. Durch gezieltes Setzen von Variablen, die im Randbereich zwischen zwei solchen stärker zusammenhängenden Codegruppierungen liegen, kann der Graph in unabhängige Komponenten zerlegt werden. Diese Komponenten können dann getrennt voneinander konfiguriert werden, die Erfüllbarkeitstests können ebenfalls unabhängig voneinander auf beide Komponenten angewandt werden. Durch Setzen des zentralen Knotens 800 (auf `true`) entsteht die in Abbildung 5.3 dargestellte Struktur. In dieser kann durch weiteres Setzen des Codes 423 eine Abspaltung der die X-, Y- und W-Codes enthaltenden Komponente erreicht werden. Zusätzliche Belegung aller verbleibenden grün markierten Variablen führt dann zu einer weiteren Aufspaltung des Interaktionsgraphen in (mindestens) zwei Komponenten.

Bei der soeben vorgestellten Untersuchung handelt es sich um eine Vereinfachung insofern, als die Ländercodes nicht berücksichtigt wurden. Fügt man im zuletzt beschriebenen, in Abbildung 5.3 dargestellten Fall, die Ländercodes wieder ein, so ergibt sich der in Abbildung 5.4 wiedergegebene Interaktionsgraph. In diesem sind die zugefügten Ländercodes als blau markierte Knoten dargestellt. Man sieht hier, dass die in einzelne Komponenten zerfallende Struktur durch die Überlagerung mit den Ländercodes beträchtlich weniger deutlich zum Tragen kommt. Erschwerend kommt noch hinzu, dass der nur informell festgeschriebene Sachverhalt, dass in einem gültigen Auftrag genau ein Ländercode vorkommen muss, in DIALOG nicht dokumentiert ist. Es existieren also keine Klauseln, die sicherstellen, dass mit dem Setzen eines Ländercodes die anderen Ländercodes durch Unit-Propagation zum Verschwinden gebracht werden.

Wir haben auch Experimente zur Bestimmung des von Dantsin [Dan83] eingeführten Teilungsgrads (*degree of the split*) vorgenommen. Zu dessen Bestimmung muss eine Menge von Begrenzungsvariablen (*boundary variables, boundary set*) berechnet werden, die (bei passender Festlegung der 'Blöcke') in unserem Beispiel aus Abbildung 5.3 mit den grün markierten Variablen übereinstimmt. Die Ergebnisse unserer experimentellen Untersuchungen waren allerdings nicht vielversprechend.

Ähnlich verhielt es sich mit dem von Rish und Dechter verwendeten Maß der *bounded (induced) tree width*. Auch hier brachten die experimentellen Ergebnisse keine überzeugende Erklärung der Handhabbarkeit unserer Konfigurationsprobleme. Wir werden auf die von Rish und Dechter verwendeten Methoden weiter unten im Zusammenhang mit Kompilierung noch näher eingehen.

Eine Partitionierung, wie in Park und Van Gelder besprochen [PVG00], könnte sich bei den Konfigurationsproblemen als erfolgversprechend erweisen, sofern diese auch auf die rekursiv entstehenden Teilprobleme angwandt wird. Untersuchungen dazu liegen allerdings (noch) nicht vor.

5.4 Kompilierung

Die meisten der zur Verifikation der Produktdokumentation durchzuführenden Erfüllbarkeitstests sind von der Form $\mathcal{B} \wedge F$. Diese können auch als logisches Folgerungsproblem (*entailment problem*) formuliert werden und stellen sich dann in der Form $\mathcal{B} \models \neg F$ dar. In diesem Fall wird $\mathcal{B}$ als eine Theorie aufgefasst, wobei man sich für die aus dieser Theorie ableitbaren Konsequenzen interessiert. Die Unerfüllbarkeit von $\mathcal{B} \wedge F$ ist dabei äquivalent zur Eigenschaft, dass $\neg F$ eine Konsequenz der Theorie $\mathcal{B}$ ist. Man sieht dies leicht ein, indem man feststellt, dass erstere Aussage äquivalent zur Tautologieeigenschaft von $\mathcal{B} \Rightarrow \neg F$ ist und dann das Deduktionstheorem (siehe z.B. [Gal86]) anwendet. Bei dem aussagenlogischen Folgerungsproblem beschränkt man sich meist auf Folgerungen F, die sich als eine Klausel darstellen lassen. Um für eine beliebige Formel F (mit $\neg F = \{c_1, \ldots, c_k\}$ in CNF) zu entscheiden, ob sie aus einer Theorie folgt, wird das Folgerungsproblem dann für jede Klausel der Negation einzeln bearbeitet.

Die Überprüfung der Konsistenzkriterien der Baubarkeitsdokumentation erfordert eine Vielzahl an Folgerungstests für ein und dieselbe Theorie $\mathcal{B}$. Es kann daher hilfreich sein, die Theorie $\mathcal{B}$ in einem Vorverarbeitungsschritt so abzuändern, dass eine äquivalente Theorie $\mathcal{B}'$ entsteht, die aber bessere Komplexitätseigenschaften aufweist. Die (Wissens-)Kompilierung (*knowledge compilation*) beschreitet genau diesen Weg. Eine Zusammenfassung der Vielzahl zur Kompilierung vorgeschlagenen Verfahren kann der Übersicht von Cadoli und Donini [CD97] oder dem Handbuchkapitel von Marquis [Mar00] entnommen werden.

Die zur Wissenskompilierung verwendeten Methoden können grob in zwei

```
ALGORITHM THEORY-APPROX
INPUT:    $\mathcal{T}_{\text{lb}}, \mathcal{T}, \mathcal{T}_{\text{ub}}, c$ mit $\mathcal{T}_{\text{lb}} \models \mathcal{T} \models \mathcal{T}_{\text{ub}}$
OUTPUT:   true falls $\mathcal{T} \models c$, false sonst
BEGIN
    IF $\mathcal{T}_{\text{ub}} \models c$ THEN return true
    ELSE IF $\mathcal{T}_{\text{lb}} \not\models c$ THEN return false
    ELSE return $\mathcal{T} \models c$
END
```

Abbildung 5.5: Algorithmus zur Theorie-Approximation.

Kategorien eingeteilt werden: Näherungsverfahren und exakte Kompilierungen. Näherungsverfahren (*approximate compilation*) verwenden meist Theorie-Approximationen, wobei eine Theorie $\mathcal{T}$ durch eine komplexitätstheoretisch handhabbarere Näherungstheorie $\mathcal{T}'$ ersetzt wird. Selman und Kautz [SK94] verwenden beispielsweise zwei angenäherte Horn-Theorien, von denen eine die Zieltheorie von oben ($\mathcal{T} \models \mathcal{T}_{\text{ub}}$), die andere von unten annähert ($\mathcal{T}_{\text{lb}} \models \mathcal{T}$).[1] Um nun mittels diesen Näherungstheorien ein Folgerungsproblem für eine Klausel c zu entscheiden, wird der in Abbildung 5.5 dargestellt Algorithmus verwendet. Da das Folgerungsproblem für Horn-Theorien, also auch für $\mathcal{T}_{\text{ub}}$ und $\mathcal{T}_{\text{lb}}$, in linearer Zeit entschieden werden kann, kann man darauf hoffen, dass Algorithmus `THEORY-APPROX` viele Eingaben effizient entscheidet. Je besser die Näherungstheorien, desto mehr Fälle werden handhabbar. Selman und Kautz stellen verschiedene Algorithmen zu deren Berechnung vor [SK94].

Exakte Verfahren versuchen, im Gegensatz zu Approximationen, eine zu $\mathcal{T}$ äquivalente Theorie $\mathcal{T}'$ zu finden, für die das Folgerungsproblem handhabbar, also in polynomieller Zeit entscheidbar ist. Die vorherrschende Methode um solche Kompilierungen zu berechnen besteht in der Berechnung aller Primimplikate der Theorie $\mathcal{T}$. Ein *Implikat* einer Theorie $\mathcal{T}$ ist eine nicht-triviale Klausel c (d.h. ohne komplementäre Literale) für die $\mathcal{T} \models c$ gilt. Darüberhinaus ist c ein *Primimplikat* von $\mathcal{T}$, falls keine echte Teilklausel von c ebenfalls Implikat von $\mathcal{T}$ ist. Die Berechnung der Menge $\text{PI}(\mathcal{T})$ aller Primimplikate einer als Klauselmenge gegebenen Theorie $\mathcal{T}$ ergibt eine neue Theorie $\mathcal{T}' = \text{PI}(\mathcal{T})$, die zu $\mathcal{T}$ äquivalent ist und darüberhinaus die folgende wichtige Eigenschaft besitzt: Eine Klausel c ist Konsequenz der Theorie $\mathcal{T}$, also $\mathcal{T} \models c$, genau dann, wenn es eine Klau-

---

[1] Die Schranken sind dabei in Bezug auf Modelle zu verstehen: die untere Schranke hat weniger, die obere mehr Modelle als die Vergleichstheorie.

sel $p \in \mathrm{PI}(\mathcal{T})$ gibt, die eine Teilklausel von c ist. Verwendet man also $\mathcal{T}' = \mathrm{PI}(\mathcal{T})$ als kompilierte Version der Theorie $\mathcal{T}$, so kann das Folgerungsproblem $\mathcal{T} \models a$ für eine Klausel a in linearer Zeit in der Größe von $\mathcal{T}'$ und der Anfrage a entschieden werden. Leider ist die Anzahl der Primimplikate im schlimmsten Fall exponentiell in der Größe der Theorie $\mathcal{T}$. Daher wurden verschiedene Verfahren entwickelt, um kompaktere Kompilate zu erzeugen. Zu den vorgeschlagenen Verfeinerungen der Primimplikat-Berechnung gehören *no-merge*-Resolventen [dV94], Theorie-Primimpliakte [Mar95] und *tractable cover*-Kompilierungen [BÉM⁺97]. Eine Variante des erstgenannten Verfahrens kann in [Sin02] gefunden werden. Experimente mit den Mercedes-Benz Fahrzeugdaten ergaben bereits für die Baubarkeitsformel $\mathcal{B}$ der (bezogen auf Variablen und Klauseln, siehe Tabelle 4.2) relativ kleinen Baureihe C210/FVF eine Primimplikatmenge mit 496050800 Klauseln [Sin02]. Auf weitere Experimente in diese Richtung wurde deshalb verzichtet.

Auch das ursprünglich von Davis und Putnam entwickelte Beweisverfahren [DP60], in dem der Theorie resolvierte Klauseln hinzugefügt werden, kann als eine gewisse Form der Kompilierung verstanden werden. Auch Rish und Dechter propagieren, unter der Bezeichung *gerichtete Resolution* (*directional resolution*), diesen Ansatz [RD00].

Dabei geht man von einer gegebenen strikten, totalen Ordnung $\prec$ auf der Variablenmenge Φ_0 aus. (Nicht-tautologische) Klauseln schreiben wir in diesem Zusammenhang immer als geordnete Klauseln in aufsteigender Literal-Reihenfolge, so dass wir für die Notation einer Klausel $c = (l_1, \ldots, l_k)$ fordern, dass $\mathrm{Var}(l_i) \prec \mathrm{Var}(l_{i+1})$ für $1 \leq i < k$. Mit $\mathrm{maxlit}(c)$ bezeichnen wir das größte Literal einer Klausel, für unsere Klausel c ist also $\mathrm{maxlit}(c) = l_k$. Die Bildung von Resolventen wird dann eingeschränkt auf solche, bei denen über in den Ausgangsklauseln maximale Literale resolviert wird. Diese spezielle, eingeschränkte Form der Resolution wird auch geordnete Resolution genannt (siehe z.B. [KH69, Joy69]). Wir schreiben $\mathrm{ORes}(S,T)$ für die Menge der aus den Klauselmengen S und T mittels geordneter Resolution herleitbaren (nicht-tautologischen) Klauseln. In der grafischen Herleitungsregel-Notation ergibt sich somit für zwei geordnete Klauseln $c = (l_1, \ldots, l_k, l)$ und $d = (m_1, \ldots, m_{k'}, \bar{l})$ durch geordnete Resolution

$$\frac{(l_1, \ldots, l_k, l) \quad (m_1, \ldots, m_{k'}, \bar{l})}{(n_1, \ldots, n_{k''})}\ \mathrm{ORes}$$

die Klausel $e = (n_1, \ldots, n_{k''})$, wobei $\{n_1, \ldots, n_{k''}\} = \{l_1, \ldots, l_k\} \cup \{m_1, \ldots, m_{k'}\}$. Insbesondere gilt $n_{k''} = \max\{l_k, m_{k'}\}$ bezüglich der Ordnung $\prec$. Wir erlauben die Anwendung der ORes-Regel nur, wenn dadurch keine triviale (tautologische) Klausel entsteht, wodurch das maximale Literal der resolvierten Klausel auch eindeutig bestimmt ist. Außerdem verwenden wir die Notation $S \vdash_{\mathrm{ORes}} c$, um anzuzeigen, dass sich die Klausel c durch endlich viele (evtl. auch gar keine) Anwendungen der ORes-Regel aus der Klauselmenge S ableiten lässt. Den aus den Ableitungsschritten sich dabei ergebenden Baum nennen wir auch Herleitung oder Beweis von c (aus S).

Durch iterierte Anwendung der geordneten Resolution auf eine durch eine Klauselmenge S gegebene Theorie $\mathcal{T}$ entstehen der Reihe nach die Mengen R_i für $i \geq 0$ durch

$$R_0 = S \quad \text{und} \quad R_{i+1} = R_i \cup \mathrm{ORes}(R_i, R_i) \ .$$

Es gilt auf jeden Fall $R_n = R_{n+1}$ für $n = |\Phi_0|$, so dass sich R_n als der Abschluss der Menge S unter geordneter Resolution (ORes) ergibt. Diesen Abschluss kann man auch, eventuell unter Weglassen subsumierter Klauseln[2], als eine kompilierte, zu $\mathcal{T}$ äquivalente Theorie $\mathcal{T}'$ verstehen. Wegen der Widerlegungsvollständigkeit der geordneten Resolution (siehe z.B. [FLTZ93]) enthält die kompilierte Theorie die leere Klausel genau dann, wenn die Ursprungstheorie $\mathcal{T}$ inkonsistent ist.

Eine weitere wichtige Eigenschaft einer unter ORes abgeschlossenen Menge, die gerade in Bezug auf die Konfiguration eine nicht zu unterschätzende Rolle spielt, ist die Möglichkeit, sukzessive durch Setzen der Variablen in der durch die Ordnung $\prec$ vorgegebenen Reihenfolge zu einem Modell der Formel zu gelangen, ohne dass dabei "falsche" Wege beschritten werden und somit Backtracking erforderlich ist. Der Algorithmus dazu sieht wie folgt aus: Ausgehend von einer (erfüllbaren) Klauselmenge S, deren bezüglich $\prec$ kleinste Variable x ist, berechnen wir die beiden Mengen $S' := S|_x$ und $S'' := S|_{\bar{x}}$. Diese Berechnungen sind in linearer Zeit in der Größe von S möglich. Falls $\Box \in S'$, so wird der Algorithmus mit S'' neu gestartet, falls $\Box \in S''$, so mit S'. Ist beides nicht der Fall, so kann der Benutzer zwischen dem Setzen von x auf $\mathbf{true}$ oder $\mathbf{false}$ wählen. Abhängig von der Wahl wird danach das Verfahren rekursiv mit entweder S', falls x auf $\mathbf{true}$ gesetzt wurde, oder andernfalls mit S'' fortgesetzt. Das ganze Verfahren bricht ab, wenn S keine Klauseln mehr enthält. Die dabei gesetzten Variablen geben dann das Modell an. Der Algorithmus findet dabei immer ein Modell, da nie gleichzeitig $\Box \in S'$ und $\Box \in S''$ auftreten kann:[3]

---

[2] Wir nennen eine Klauselmenge, die keine subsumierten Klauseln enthält, auch *abgeschlossen unter Subsumtion* (Subs).

[3] Außerdem überprüft man leicht, dass $S|_x$ und $S|_{\bar{x}}$ unter ORes abgeschlossen sind, falls x minimal und S unter ORes abgeschlossen ist.

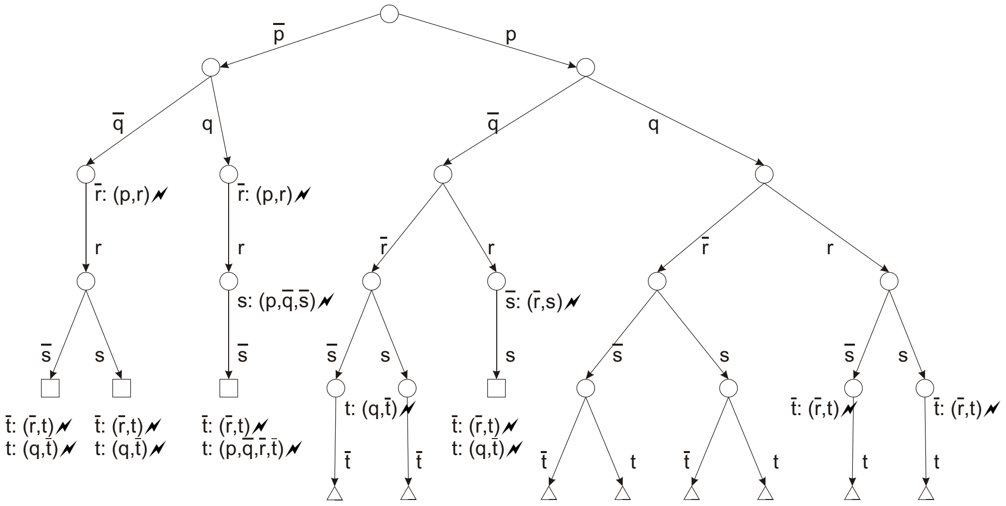


Abbildung 5.6: Beispiel zur interaktiven Konfiguration, Entscheidungsbaum.

Lemma 5.4.1 *Sei S eine unter* ORes *abgeschlossene Klauselmenge mit $\Box \notin S$. Dann gilt für alle $x \in \mathrm{Var}(S)$: Die leere Klausel $\Box$ ist nicht sowohl in $S|_x$ als auch in $S|_{\bar{x}}$ enthalten.*

Beweis: Angenommen $\Box \in S|_x$ und $\Box \in S|_{\bar{x}}$ für ein $x \in \mathrm{Var}(S)$. Da $\Box \notin S$, muss aufgrund der Definition von $S|_l$ sowohl $(x) \in S$ als auch $(\bar{x}) \in S$ sein. Da S unter ORes abgeschlossen ist, gilt dann aber auch zwangsläufig $\Box \in S$, ein Widerspruch. $\Box$

Die unterschiedlichen Situationen bei der interaktiven Konfiguration sind nochmals beispielhaft in den Abbildungen 5.6 und 5.7 veranschaulicht. Wir betrachten dazu die Klauselmenge

$$S = \{\ (p,r),(q,\bar{t}),(q,\bar{r},s),(\bar{r},t),(p,\bar{q},\bar{s}),(p,q,t),(p,\bar{q},\bar{r},\bar{t})\ \}$$

und deren Abschluss unter ORes und Subs bezüglich der Ordnung $p \prec q \prec r \prec s \prec t$:

$$S' = \{\ (p),(q,\bar{r}),(q,\bar{t}),(\bar{r},t)\ \}\ .$$

Abbildung 5.6 zeigt den Entscheidungsbaum, der sich bei der interaktiven Konfiguration in der durch $\prec$ vorgegebenen Ordnung für die Klauselmenge S ergibt. Da sowohl $\Box \notin S|_p$ als auch $\Box \notin S|_{\bar{p}}$, kann der Benutzer zwischen diesen beiden Alternativen wählen. Entscheidet er sich jedoch für

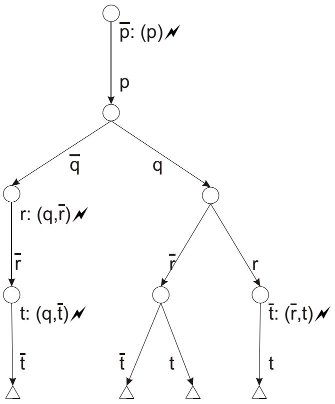


Abbildung 5.7: Beispiel zur interaktiven Konfiguration, Entscheidungsbaum nach Abschluss unter ORes.

$p = \mathbf{false}$, so kann dies durch keine weiteren Festlegungen der verbleibenden Variablen zu einer gültigen Konfiguration ergänzt werden. Alle Zweige enden mit einem Widerspruch (angezeigt durch □-Knoten). So ist man z.B. nach weiterer Auswahl von $q = \mathbf{true}$ auf die Belegungen $r = \mathbf{true}$ und $s = \mathbf{false}$ festgelegt, da sich ansonsten ein Widerspruch ergibt, angezeigt durch die dann zur leeren Klausel reduzierten Klauseln (p, r) und $(p, \bar{q}, \bar{s})$ der ursprünglichen Klauselmenge S, die neben den zugehörigen Kanten angegeben sind. Dann ergibt sich aber sowohl beim Setzen von $t = \mathbf{true}$ als auch beim Setzen von $t = \mathbf{false}$ ein Widerspruch, der den Benutzer dazu zwingt, seine ursprüngliche Festlegung $p = \mathbf{false}$ (oder $q = \mathbf{true}$) zu ändern. Letztendlich kommt der Benutzer aber (nach Backtracking) an einem erfüllenden Knoten – entsprechend einer gültigen Konfigurtion – an, sofern ein solcher existiert; in unserem Beispiel könnte sich nach Auswahl von $p = q = r = \mathbf{true}$ und $s = \mathbf{false}$ mit der erzwungenen Belegung $t = \mathbf{true}$ eine gültige Konfiguration, angezeigt durch einen $\triangle$-Knoten, ergeben.

Bei Verwendung der unter ORes abgeschlossenen Klauselmenge S' hingegen können solche falschen Variablenbelegungen, die zur Rücknahme bereits getroffener Entscheidungen zwingen, nicht auftreten. Der zugehörige Entscheidungsbaum in Abbildung 5.7 verdeutlicht dies (die beliebig belegbare Variable s ist hier weggelassen). Jede dem Benutzer gegebene Entscheidungsmöglichkeit führt zu mindestens einer Lösung, falsche Belegungen werden durch den einfachen, oben angegebenen Algorithmus bereits ausgeschlossen.

Wenden wir uns nun wieder den Mercedes-Benz-Fahrzeugbaureihen und

BR/AA	n	k	k_{Subs}	k_{ORes}	t_{ORes}
C168/FW	653	4299	3550	4468	499.79
C170/FR	554	7579	6917	8496	2365.76
C202/FS	698	4954	4258	5186	626.51
C202/FW	734	7290	6440	9394	4092.02
C208/FA	633	3720	3393	3885	293.47
C208/FC	652	3975	3622	5422	615.12
C210/FS	723	4025	3712	4707	484.78
C210/FW	775	5491	5014	6703	1032.57
C220/FV	597	3368	2865	3131	186.80
C638/FKA	528	4966	1562	1393	17.61

Tabelle 5.2: Kompilierung durch Abschluss unter geordneter Resolution und Subsumtion.

deren Konsistenzprüfung zu. Die Ergebnisse einer Kompilierung der Baubarkeitsformeln $\mathcal{B}$ durch geordnete Resolution (ORes) sind in Tabelle 5.2 dargestellt. Die ersten drei Spalten geben die Baureihen- und Ausführungsart-Bezeichung, die Anzahl der Variablen (n) und die Anzahl der Klauseln (k) der Baubarkeitsformel $\mathcal{B}$ wieder. Dabei wurden Unit-Klauseln durch einen vorverarbeitenden Unit-Propagation-Schritt bereits aus der Ausgangsformel entfernt, so dass in diesen Zahlen Einheits-Klauseln nicht mitgerechnet sind. Durch Streichen subsumierter Klauseln, also durch Bildung des Abschlusses unter Subs, ergibt sich eine verkleinerte Klauselmenge, deren Größe (Anzahl der Klauseln) in der nächsten Spalte (k_{Subs}) angegeben ist. Berechnung des Abschlusses unter Subsumtion und geordneter Resolution liefert dann eine Klauselmenge mit k_{ORes} Klauseln. Die Gesamtzeit (in Sekunden) zur Berechnung der Kompilierung mittels eines einfachen Prototyps in C++ ist in der letzten Spalte (t_{ORes}) wiedergegeben. Die Messungen wurden auf denselben Rechnern und unter denselben Bedingungen wie in Abschnitt 4.1.4 durchgeführt. Die zur Berechnung der geordneten Resolution verwendete Variablenordnung basiert dabei auf der Funktion $f(v) = (p(v) + 1) \cdot (n(v) + 1)$, wobei $p(v)$ die Anzahl der positiven, $n(v)$ die der negativen Vorkommen von Variable v in der betrachteten Klauselmenge S bezeichnet. Dann gilt $x \prec y$ für $x, y \in \text{Var}(S)$, falls $f(x) > f(y)$. Variablen mit gleichem Funktionswert werden beliebig geordnet. Durch diese Ordnung werden einerseits Variablen, die häufig im Problem vorkommen, bei der interaktiven Konfiguration zuerst betrachtet, andererseits aber auch die Menge der möglichen neuen geordneten Resolventen potentiell minimiert.

Abschließend lässt sich feststellen, dass die Kompilierung mittels geordneter Resolution ausgesprochen überzeugende Ergebnisse für die Mercedes-Fahrzeugdatensätze liefert. Zur interaktiven Konfiguration der Fahrzeuge in der durch die soeben beschriebene Ordnung festgelegten Reihenfolge ist dieses Verfahren ausreichend (kein Backtracking!). Soll allerdings eine andere als die vorgegebene Ordnung als Zuweisungsreihenfolge an die Variablen verwendet werden, so bedarf die Methode einer Erweiterung. Ebenso ist die Kompilierung mittels geordneter Resolution für die Behandlung des Folgerungsproblems nicht ausreichend. Im nächsten Abschnitt werden wir ein Verfahren vorstellen, mit dem sich beide Schwächen überwinden lassen.

5.5 Aktive Literale und Klauseln

Wir wollen nun das Folgerungsproblem für eine unter geordneter Resolution abgeschlossenen Klauselmenge betrachten. Für eine Anfrageklausel a und eine gegebene Klauselmenge S ist dies die Frage nach $S \models a$. Falls $a = (l_1, \ldots, l_k)$ genau die k bezüglich $\prec$ minimalen Variablen von S umfasst, so ist das Folgerungsproblem mit einem dem im letzten Abschnitt angegebenen Algorithmus ähnlichen Verfahren leicht zu lösen: Man generiert sukzessive die Mengen $S_1 = S|_{\bar{l}_1}, \ldots, S_k = S_{k-1}|_{\bar{l}_k}$. Dann ist $\Box \in S_k$ genau dann, wenn $S \models a$. Sind die in a enthaltenen Literale l_i nicht die k kleinsten der Ordnung, so müssten im Entscheidungsbaum alle möglichen Belegungen kleinerer Variablen als der in a maximalen überprüft werden, was nur in den wenigsten Fällen praktikabel ist.

Eine Alternative besteht darin, die Äquivalenz des Folgerungsproblems $S \models (l_1, \ldots, l_k)$ mit dem Erfüllbarkeitsproblem $S \cup \{(\bar{l}_1), \ldots, (\bar{l}_k)\}$ auszunutzen, und auf letztere Klauselmenge S' eine Erfüllbarkeitsprüfung anzuwenden. Dafür bietet sich dann wieder geordnete Resolution an, da S bereits unter ORes abgeschlossen ist. Man kann daher hoffen, dass durch das Hinzufügen der k neuen Unit-Klauseln nur geringe Modifikationen an S durchzuführen sind, um den Abschluss von S' zu erhalten. Diese Ideen wollen wir nun genauer ausarbeiten.

Für eine Klausel $c = (l_1, \ldots, l_k)$ bezeichnen wir im folgenden mit $\bar{c} = \{(\bar{l}_1), \ldots, (\bar{l}_k)\}$ die Menge der Einheitsklauseln, die deren Negation entspricht. In Anlehnung an die Arbeit von Basin und Ganzinger [BG01] bezeichnen wir eine unter ORes bezüglich $\prec$ und Subs abgeschlossene Klauselmenge auch als $\prec$-saturiert (*saturated up to redundancy under ordered* *resolution with respect to* $\prec$*).*

Definition 5.5.1 (aktive Literale) *Gegeben eine* $\prec$*-saturierte Klauselmenge* S *und eine Klausel* $a = (m_1, \ldots, m_j)$*. Die Menge der* ***aktiven Literale*** $A(S,a)$ ***von*** S, a *ist dann rekursiv definiert durch*

$$\begin{aligned} A_0(S,a) &= \bigcup \bar{a} = \{\bar{m}_1, \ldots, \bar{m}_j\} \ , \\ A_{i+1}(S,a) &= \ A_i(S,a) \ \cup \ \{p_k \mid (p_1, \ldots, p_k, q_1, \ldots, q_l) \in S, \ l \geq 1 \textit{ und} \\ &\qquad\qquad\qquad\qquad P_i(q_j) \textit{ für alle } j \textit{ mit } 1 \leq j \leq l\} \ , \\ A(S,a) &= \bigcup_{i \geq 0} A_i(S,a) \ . \end{aligned}$$

Dabei verwenden wir das Prädikat $P_i(q)$ *als Abkürzung für die Bedingung*

$$q \in A_i(S,a) \wedge (\exists c \in S)(\mathrm{maxlit}(c) = \bar{q}) \ \vee \ \bar{q} \in A_i(S,a) \ .$$

Die erweiterte Menge der aktiven Literale, $A^*(S,a)$*, enthält zusätzlich das Pseudoliteral* $\Box$*, falls es ein* $c \in S$ *gibt, so dass* $P(q)$ *für alle* $q \in c$ *gilt, wobei* $P(q)$ *aus* $P_i(q)$ *durch Ersetzen von* $A_i(S,a)$ *durch* $A(S,a)$ *entsteht.*

Der Zusammenhang zwischen den aktiven Literalen $A(S,a)$ und dem Folgerungsproblem $S \models a$ wird in nachfolgendem Lemma und Satz klargemacht.

Lemma 5.5.2 *Sei* S *eine* $\prec$*-saturierte Klauselmenge und* a *eine Klausel. Dann ist das* $\prec$*-maximale Literal einer jeden aus* $S \cup \bar{a}$ *per geordneter Resolution herleitbaren, nicht von* S *subsumierten Klausel ein aktives Literal von* S, a*.*

Beweis: Betrachte zu gegebenem S und a die für $i \geq 0$ durch

$$R_0 = \bar{a}, \quad R_{i+1} = R_i \cup \mathrm{ORes}(R_i \cup S, R_i)$$

induktiv definierten Resolventenmengen R_i. Wir beweisen zuerst per simultaner Induktion über i die beiden Eigenschaften

(a) $A_i(S,a) \supseteq \mathrm{maxlit}(R_i)$ und

(b) falls $(\ldots, p, q) \in R_i$ und $P_i(q)$, so ist $p \in A_{i+1}(S,a)$.

Dabei bezeichnet $P_i(q)$ die in Def. 5.5.1 angegebene Bedingung. Für die Formulierung der Behauptung (a) haben wir die Definition von maxlit auf Klauselmengen erweitert durch $\mathrm{maxlit}(M) = \bigcup_{c \in M} \mathrm{maxlit}(c)$.

Für $i = 0$ ist $A_0(S, a) = \bigcup \bar{a} = \mathrm{maxlit}(R_0)$, so dass Behauptung (a) in diesem Fall gilt. Da R_0 nur Unit-Klauseln enthält, gilt auch Teil (b).

Nehmen wir nun also an, die Behauptungen (a) und (b) seien bereits für i bewiesen. Wir beweisen nun zuerst (b) für $i + 1$. Sei $c = (\dots, p, q) \in R_{i+1}$ und gelte $P_{i+1}(q)$. Betrachte die ORes-Herleitung H von c aus $S \cup \bar{a}$. Falls es in dieser Herleitung eine Klausel $c' = (\dots, p, q, r_1, \dots, r_k) \in S$ als Vorgänger von c gibt, so muss zum Erhalt von c über $r_1, \dots, r_k$ in H resolviert worden sein. Jeder dieser Resolutionsschritte (wir betrachten den für r_j) involviert als Elternklauseln entweder zwei Klauseln aus R_i oder eine Klausel aus R_i und eine aus S. In ersterem Fall ist nach Induktionsvoraussetzung (a) (neben r_j auch) $\bar{r}_j \in A_i(S, a)$, damit auch $\bar{r}_j \in A_{i+1}(S, a)$ und somit gilt $P_{i+1}(r_j)$. In letzterem Fall ist nach Induktionsvoraussetzung (a) $r_j^* \in A_i(S, a)$ (somit auch in $A_{i+1}(S, a)$), wobei $r_j^* = r_j$ oder $r_j^* = \bar{r}_j$, und wegen $(\dots, \bar{r}_j^*) \in S$ gilt auch hier $P_{i+1}(r_j)$. Insgesamt erhalten wir die Gültigkeit von $P_{i+1}(r_j)$ für alle j mit $1 \leq j \leq k$. Nach der rekursiven Def. der $A_i(S, a)$ ist daher $p \in A_{i+2}(S, a)$. Falls es in der Herleitung von c keine Vorgängerklausel $(\dots, p, q, \dots) \in S$ gibt, so muss es zwei Klauseln $d' = (\dots, p, s_1, \dots, s_l) \in S$ und $e' = (\dots, q, t_1, \dots, t_m) \in S$ geben, aus denen die benachbarten Literale p und q in c entstanden sind, sowie eine Zwischenklausel $c' = (\dots, p, q, r_1, \dots, r_k)$, in der p und q erstmals gemeinsam auftraten:

$$\begin{gathered}
\cfrac{\begin{matrix} d' = (\dots, p, s_1, \dots, s_l) & \qquad & e' = (\dots, q, t_1, \dots, t_m) \\ \vdots & & \vdots \\ (\dots, p, \dots, s_1, \dots, s_{l'}, \dots) & & (\dots, q, \dots, t_1, \dots, t_{m'}, \dots) \end{matrix}}{c' = (\dots, p, q, r_1, \dots, r_k)} \ \mathrm{ORes} \\
\vdots \\
c = (\dots, p, q)
\end{gathered}$$

Zum Erhalt von c muss also (u.a.) über $s_1, \dots, s_l$ und $t_1, \dots, t_m$ resolviert worden sein. Analog zu obiger Argumentation erhalten wir $P_{i+1}(s_j)$ für alle $1 \leq j \leq l$, womit nach Def. der $A_i(S, a)$ auch $p \in A_{i+2}(S, a)$. (Der Fall $l = 0$ kann wegen der Herleitung von c ausgeschlossen werden.) Damit ist Behauptung (b) bewiesen.

Uns nun dem Induktionsschritt von Behauptung (a) zuwendend, betrachten wir eine Klausel $c \in R_{i+1}$. Diese kann auf drei Arten entstanden sein:

(i) durch Resolution zweier Klauseln aus R_i; (ii) durch Resolution einer Klausel aus R_i mit einer Klausel aus S; oder (iii) die Klausel c war schon in R_i enthalten. Wir betrachten nun diese drei Fälle:

(i) Hierbei ist Klausel c entstanden aus zwei Klauseln $d, e \in R_i$ durch Anwendung der Regel

$$\frac{d = (\dots, p, q) \in R_i \quad e = (\dots, r, \bar{q}) \in R_i}{c = (\dots, m) \in R_{i+1}}\ \mathrm{ORes}$$

wobei $m = \max\{p, r\}$.[4] Nach Induktionsvoraussetzung (a) ist $q, \bar{q} \in A_i(S, a)$, also nach (b) $p, r \in A_{i+1}(S, a)$ und daher auch $m = \mathrm{maxlit}(c) \in A_{i+1}(S, a)$.

(ii) Hier ist c durch einen Resoltionsschritt der Form

$$\frac{d = (\dots, p, q) \in R_i \quad e = (\dots, r, \bar{q}) \in S}{c = (\dots, m) \in R_{i+1}}\ \mathrm{ORes}$$

entstanden, wobei wiederum $m = \max\{p, r\}$. Nach Induktionsvoraussetzung (a) ist $q \in A_i(S, a)$, also nach Def. der $A_i(S, a)$ auch $r \in A_{i+1}(S, a)$. Außerdem gilt wegen Behauptung (b) $p \in A_{i+1}(S, a)$, womit in jedem Fall $m \in A_{i+1}(S, a)$ gilt.

(iii) In diesem Fall ist $c \in R_i$, es gilt nach Induktionsvoraussetzung (a) also $\mathrm{maxlit}(c) \in A_i(S, a)$. Aufgrund der Definition von $A_i(S, a)$ ist auch $\mathrm{maxlit}(c) \in A_{i+1}(S, a)$.

Damit ist der Beweis der Hilfsbehauptungen (a) und (b) abgeschlossen, wobei (a) die Behauptung des Lemmas liefert, sofern $\bigcup_{i \geq 0} R_i$ alle möglichen, aus $S \cup \bar{a}$ durch geordnete Resolution herleitbaren, nicht von S subsumierten Resolventen enthält. Dies ist aber leicht einzusehen (z.B. per Induktion über die Herleitungslänge), unter Beachtung der Tatsache, dass alle geordneten Resolventen zwischen Klauseln in S entweder bereits in S vorhanden sind (Abschluss unter ORes) oder von S subsumiert werden (Abschluss unter Subs). $\square$

Satz 5.5.3 *Sei S eine $\prec$-saturierte Klauselmenge und a eine Klausel mit $S \models a$. Dann gibt es einen* ORes*-Beweis $S \cup \bar{a} \vdash_{\mathrm{ORes}} \Box$, so dass alle Beweisschritte nach aktiven Literalen von S, a resolvieren.*

---

[4] Wir formulieren hier explizit nur den Fall, dass es sich bei d und e um keine Unit-Klauseln handelt; die Unit-Klausel-Fälle sind analog zu beweisen.

Beweis: Angenommen $S \models a$. Aus der Widerlegungsvollständigkeit der geordneten Resolution folgt, dass es eine Herleitung von $S \cup \bar{a} \vdash_{\mathrm{ORes}} \Box$ gibt. Sei H eine kürzeste solche Herleitung. (Falls $\Box \in S$, so ist H die leere Herleitung, für die die Behauptung des Satzes trivialerweise gilt.) Dann gibt es in H keine Resolventen der Form

$$\frac{d \in S \quad e \in S}{c}\ \mathrm{ORes}\ ,$$

da S unter ORes abgeschlossen ist, und somit bereits $c \in S$. Also hat jedes Blatt aus dem Herleitungsbaum H die Gestalt

$$\frac{d \in S \cup \bar{a} \quad e \in \bar{a}}{c}\ \mathrm{ORes}\ ,$$

wobei $e = (l) \in \bar{a}$ eine Unit-Klausel ist, für deren Literal $l \in A(S, a)$ gilt. Somit gilt für die Blätter von H die Behauptung. Für die inneren Knoten der Herleitung H findet Lemma 5.5.2 Anwendung. Der Fall, dass eine innere Klausel von S subsumiert wird, kann nicht auftreten, da der Beweis ansonsten verkürzt werden könnte. Somit gilt für alle Resolutionsschritte im Herleitungsbaum H die Behauptung. $\Box$

Korollar 5.5.4 *Sei S eine $\prec$-saturierte Klauselmenge mit $\Box \notin S$ und a eine (nicht-tautologische) Klausel mit $\Box \notin A^*(S, a)$. Dann gilt $S \not\models a$.*

Beweis: Sei $\Box \notin S$ und $\Box \notin A^*(S, a)$. Angenommen es gelte $S \models a$. Nach Satz 5.5.3 gibt es dann eine Herleitung $S \cup \bar{a} \vdash_{\mathrm{ORes}} \Box$, in der alle Beweisschritte nach aktiven Literalen von S, a resolvieren. Sei H eine solche Herleitung. Sei ferner $b = (l_1, \ldots, l_k) \in S$ eine Klausel, die als Blatt des Herleitungsbaumes H auftritt. Da a nicht tautologisch ist und $\Box \notin S$, existiert immer eine solche. Über alle Literale l_i aus b muss zur Herleitung der leeren Klausel resolviert worden sein, wobei nach Voraussetzung für alle i ($1 \leq i \leq k$) zumindest eines der Literale l_i und $\bar{l}_i$ ein aktives Literal ist. Bertrachten wir nun den Beweisschritt, der über l_i resolviert, genauer:

$$\frac{c = (\ldots, l_i) \quad d = (\ldots, \bar{l}_i)}{e}\ \mathrm{Res}$$

Falls $d \notin S$, so ist nach Lemma 5.5.2 $\bar{l}_i$ ein aktives Literal, und daher gilt $P(l_i)$. Andernfalls ist $d \in S$ und $c \notin S$ eine aus $S \cup \bar{a}$ per ORes herleitbare, nicht von S subsumierte Klausel, für die Lemma 5.5.2 wiederum Anwendung findet, womit $l_i \in A(S, a)$. Da außerdem $d \in S$ mit $\mathrm{maxlit}(d) = \bar{l}_i$

ist, gilt auch hier $P(l_i)$, insgesamt also $P(l_i)$ für alle $l_i \in b$. Damit ist aber $\Box \in A^*(S,a)$, so dass die Annahme $S \models a$ falsch gewesen sein muss. $\Box$

Wir haben damit ein in polynomieller Zeit berechenbares Kriterium, um das Folgerungsproblem in manchen Fällen entscheiden zu können: Ist die leere Klausel $\Box$ nicht in der erweiterten Menge der aktiven Literale $A^*(S,a)$, so kann $S \models a$ nicht gelten. Ist $\Box \in A^*(S,a)$, so kann man keine direkte Aussage treffen, ob a aus S folgt oder nicht.

In diesem Fall versuchen wir die Komplexität des Entscheidungsverfahrens abzuschätzen, indem wir eine obere Schranke für die Zahl der zum ORes-Abschluss von $S \cup \bar{a}$ neu zu generierenden Klauseln angeben. Ähnliche Kriterien wurden auch von Rish und Dechter unter der Bezeichung (induzierte) Diversität (*(induced) diversity*) eingeführt [RD00]. Rish und Dechter definieren die (induzierte) Diversität als einen zu einer (unter ORes abgeschlossenen) Theorie S gehörenden Parameter. Die in dem zitierten Artikel dargestellten Ergebnisse sind allerdings nur auf in der Praxis kaum anzutreffende Situationen anwendbar.

Wir erweitern nun zu einem gegebenen Folgerungsproblem $S \models a$ den Begriff der Aktivität von Literalen, $A(S,a)$, auf die zugehörigen Klauseln der Theorie S.

Definition 5.5.5 (aktive/passive Klauseln) *Sei S eine $\prec$-saturierte Klauselmenge und a eine Klausel. Eine Klausel $c = (p_1, \ldots, p_k)$ heißt **aktive Klausel von** S, a, falls entweder $c \in \bar{a}$ oder falls es ein $c' = (p_1, \ldots, p_k, q_1, \ldots, q_l) \in S$ gibt mit $l \geq 1$ und $P(q_i)$ für alle $1 \leq i \leq l$. Die Menge der **aktiven Klauseln eines Literals** l **(bezüglich** S, a**)** ist definiert als*

$$\mathrm{AK}_l \ := \ \{c \mid c \textit{ ist aktive Klausel von } S, a \textit{ mit } \mathrm{maxlit}(c) = l\} \ .$$

*Die Menge der **passiven Klauseln von** l **(bezüglich** S**)** als*

$$\mathrm{PK}_l \ := \ \{c \in S \mid \mathrm{maxlit}(c) = l\} \ .$$

Für gegebenes S und a bezeichen wir des weiteren mit $\mathbf{A}(l) := |\mathrm{AK}_l|$ und $\mathbf{P}(l) := |\mathrm{PK}_l|$ die Mächtigkeiten der aktiven und passiven Klauselmengen zu einem Literal l.

Zu einer gegebenen Klauselmenge S sei $\mathrm{Var}(S) = (x_1, \ldots, x_n)$ geordnet bezüglich der Ordnung $\prec$, d.h. $x_i \prec x_{i+1}$ für $1 \leq i < n$. Wir berechnen nun einen Vektor $(N_1, \ldots, N_n)$, anhand dessen eine obere Schranke der zusätzlich generierten Klauseln bei Abschluss von $S \cup \bar{a}$ unter ORes angegeben werden kann. Dabei soll jedes N_i die Anzahl der neu generierten, durch Resolution über x_i entstandenen Klauseln abschätzen. Wir betrachten dazu zu gegebenem i vier Fälle: (i) sowohl x_i als auch $\neg x_i$ kommen in $A(S, a)$ vor; (ii) nur x_i, nicht aber $\neg x_i$, kommt in $A(S, a)$ vor; (iii) nur $\neg x_i$, nicht aber x_i kommt in $A(S, a)$ vor; (iv) weder x_i noch $\neg x_i$ kommen in $A(S, a)$ vor. Dann gilt

$$N_i = \begin{cases} (\mathbf{A}(x_i) + \mathbf{P}(x_i) + N^+_{>i}) \cdot & \\ \qquad (\mathbf{A}(\neg x_i) + \mathbf{P}(\neg x_i) + N^-_{>i}) & \text{falls (i)} \\ (\mathbf{A}(x_i) + N^+_{>i}) \cdot (\mathbf{P}(\neg x_i) + N^-_{>i}) & \text{falls (ii)} \\ (\mathbf{P}(x_i) + N^+_{>i}) \cdot (\mathbf{A}(\neg x_i) + N^-_{>i}) & \text{falls (iii)} \\ 0 & \text{sonst} \end{cases} \tag{5.1}$$

wobei $N^+_{>i}$ und $N^-_{>i}$ Abschätzungen (obere Schranken) der Anzahl neuer Klauseln mit positivem bzw. negativem größtem Literal angeben, die durch Resolution über ein Literal l mit $\mathrm{Var}(l) \succ x_i$ entstehen können. Zur Berechnung dieser Abschätzungen stellen wir uns die neu entstandenen Klauseln (durch N_i gezählt) eingeteilt in zwei Klassen vor, solche mit positivem größtem Literal (N_i^+) und solche mit negativem größtem Literal (N_i^-). Über diese Aufteilung ist nur bekannt, dass $N_i = N_i^+ + N_i^-$ (und natürlich $N_i^+, N_i^- \geq 0$). Das Verhältnis von positiven zu negativen Klauseln hingegen ist unbekannt. Damit ergibt sich für die Summen $N^+_{>i}$ und $N^-_{>i}$ dann $N^+_{>i} = \sum_{j=i+1}^{n} N_j^+$ und $N^-_{>i} = \sum_{j=i+1}^{n} N_j^-$.

Betrachtung des *worst case* erfordert die Maximierung der N_i, wobei das Verhältnis zwischen $N^+_{>i}$ und $N^-_{>i}$ variabel ist. Dies lässt sich als parametrisiertes Optimierungsproblem ausdrücken, bei dem die Zielgröße $z = (a + x) \cdot (b + y)$ unter den Nebenedingungen $x + y = c$, $x > 0$, und $y > 0$ maximiert werden soll. In unserem Fall ergibt sich als z die zu maximierende Größe N_i, für x und y hat man sich die variablen Anteile $N^+_{>i}$ und $N^-_{>i}$ vorzustellen. Die Parameter a und b ergeben sich aus den Gleichungen 5.1 als die in den Summanden des Produkts vorkommenden restlichen Glieder, c ist die Summe $N_{>i} = N^+_{>i} + N^-_{>i}$.

Als Lösung des Optimierungsproblems ergibt sich der Ausdruck

$$z_{\max}(a, b, c) = \begin{cases} \lceil \frac{1}{4}(a + b + c)^2 \rceil & \text{falls } |a - b| \leq c, \\ a \cdot (b + c) & \text{falls } a - b > c, \\ (a + b) \cdot c & \text{falls } b - a > c, \end{cases}$$

womit sich für die N_i unter Verwendung der Abkürzung $N_{>i} := N^+_{>i} + N^-_{>i}$

ergibt:

$$N_i = \begin{cases} z_{\max}(\mathbf{A}(x_i) + \mathbf{P}(x_i), \mathbf{A}(\neg x_i) + \mathbf{P}(\neg x_i), N_{>i}) & \text{falls (i)} \\ z_{\max}(\mathbf{A}(x_i), \mathbf{P}(\neg x_i), N_{>i}) & \text{falls (ii)} \\ z_{\max}(\mathbf{P}(x_i), \mathbf{A}(\neg x_i), N_{>i}) & \text{falls (iii)} \\ 0 & \text{sonst} \end{cases}$$

Man kann diese Abschätzungen unter zusätzlicher Verwendung der Information, welche Literale in Einheits-Klauseln vorkommen, noch weiter verbessern: Sei $U(S, a)$ die Menge der sich durch Unit-Propagation aus der Klauselmenge $S \cup \bar{a}$ ergebenden Literale . Diese lässt sich in linearer Zeit in S berechnen. Wir nehmen an, dass $U(S, a)$ keine komplementären Literale enthält, denn ansonsten ist $S \cup \bar{a}$ direkt als unerfüllbar erkannt. Ist also $x_i \in U(S, a)$ oder $\neg x_i \in U(S, a)$, so können durch Resolution über x_i zwar bis zu $\mathbf{A}(\neg x_i) + \mathbf{P}(\neg x_i) + N_{>i}$ neue Klauseln entstehen, diese subsumieren aber jeweils mindestens eine bereits bestehende Klausel, so dass sich bei gleichzeitigem Subsumtionstest in den Fällen $x_i \in U(S, a)$ mit $N_i = 0$ arbeiten lässt

Als obere Schranke für die Anzahl der durch Abschluss von $S \cup \bar{a}$ unter ORes und Subs neu entstandenen Klauseln N ergibt sich somit $N = \sum_{i=1}^{n} N_i$.

BR/AA	Prob.	A	V	U	☐
C202/FW	UC-G1	208	126	1	ja
C202/FW	UC-G2	135	80	68	ja
C202/FW	UC-G3	31	20	11	ja
C202/FW	NC-G1	208	126	1	ja
C202/FW	NC-G2	135	80	1	ja
C202/FW	NC-G3	31	20	1	ja
C202/FW	UC-M1	196	116	17	ja
C202/FW	UC-M2	207	125	73	ja
C202/FW	UC-M3	167	95	16	ja
C202/FW	NC-M1	196	116	2	ja
C202/FW	NC-M2	207	125	1	ja
C202/FW	NC-M3	167	95	2	ja

Tabelle 5.3: Ergebnisse einiger Experimente mit aktiven Literalen.

Für die Fahrzeugdaten der Mercedes-Baureihe C202/FW haben wir mit ersten Experimenten zur Untersuchung aktiver Literale begonnen. Dazu haben wir Folgerungtests der Form $\mathcal{B} \Rightarrow l$ für verschiedene Literale l durchgeführt, entsprechend den Konsistenztests auf unzulässige und notwendige Codes. Wir haben dabei l so ausgewählt, dass sowohl l als auch $\bar{l}$ in mindestens einer Klausel von $\mathcal{B}$ als bezüglich $\prec$ größtes Literal auftrat. Ansonsten wäre zumindest bei einem der beiden Konsistenztests (notwendige bzw. unzulässige Codes) zu $x = \mathrm{Var}(l)$ die Menge der aktiven Literale trivial, sie bestände nur aus dem Literal selbst. Außerdem haben wir solche Literale gewählt, deren Variable x in der Ordnung $\prec$ nicht zu den kleinsten gehört: Drei Experimente haben wir mit bezüglich $\prec$ großen Variablen (G1 = 404L, G2 = 145A, G3 = 540) und drei mit bezüglich $\prec$ mittleren Variablen durchgeführt (M1 = 761, M2 = 329, M3 = 774). Die Ergebnisse unserer Experimente sind in Tabelle 5.3 dargestellt. Dabei steht in der Problem-Spalte, wie bereits weiter oben, UC für den Test auf unzulässige, NC für den Test auf notwendige Codes. In Spalte A ist die Anzahl der aktiven Literale $l' \in A(\mathcal{B}, (l))$ für das Folgerungsproblem $\mathcal{B} \models l$, in der Spalte V die Anzahl der Variablen aktiver Literale und in Spalte U die Anzahl der aktiven Literale, die aus $\mathcal{B} \cup \neg l$ durch Unit-Propagation entstehen. Die letzte Spalte gibt schließlich an, ob sich die leere Klausel $\Box$ in der erweiterten Literalmenge $A^*(\mathcal{B}, (l))$ befand. Ist dies nicht der Fall, so ist, wie oben bereits erwähnt, das Folgerungsproblem bereits gelöst.

Es ist zu erkennen, dass die Menge der aktiven Literale stets nur einen relativ geringen Bruchteil aller möglichen 1468 Literale ausmacht, so dass zumindest bei der Konsistenzprüfung der Fahrzeugdaten eine Problemvereinfachung ermöglicht wird. In Tabelle 5.3 sind darüberhinaus bereits schwierigere Instanzen des Folgerungsproblems für Unit-Klauseln wiedergegeben. Von den insgesamt 1468 Folgerungstests der Form $\mathcal{B} \models \pm x$ können 625, also 42.6%, bereits ohne Suche, nur durch Anwendung von Korollar 5.5.4 gelöst werden.

6

Beweiser und Parallelisierbarkeit

In diesem Kapitel wollen wir den parallelen automatischen Beweiser (SAT-Checker) PaSAT [SBK01] vorstellen, der in Zusammenarbeit mit W. Blochinger im Rahmen dieser Dissertation entstanden ist. Eine sequentielle Version von PaSAT wurde auch zur Durchführung der in Abschnitt 4.1.4 vorgestellten Experimente verwendet.

PaSAT ist eine Implementation des Davis-Putnam-(oder genauer Davis-Putnam-Logemann-Loveland-)Algorithmus [DP60, DLL62] mit verschiedenen Literalauswahlstrategien, nicht-chronologischem Backtracking [BJS97] und Lemmagenerierung [MSS96]. Im Gegensatz zu anderen parallelen Implementierungen des DPLL-Verfahrens [BS96, ZBH96, JCU01] werden in PaSAT die generierten Lemmata auch zwischen verschiedenen, an derselben Probleminstanz arbeitenden Prozessen ausgetauscht.

6.1 Sequentieller Algorithmus

Der grundlegende sequentielle, in PaSAT implementierte Algorithmus ist in Abbildung 6.1 wiedergegeben. Dieser wird initial mit der Klauselmenge S der Probleminstanz und einem (Such-)Level von 0 aufgerufen. Zeilen (3)-(7) dieses Algorithmus sind neben Unit-Propagation (bestehend aus Unit-Subsumption und Unit-Resolution) zur Problemvereinfachung auch dafür zuständig, die Ursache für eine neu entstandene Unit-Klausel festzuhalten. Als Ursache wird dabei eine (minimale) partielle Variablenbelegung $V = \{x_1 \mapsto b_1, \dots, x_k \mapsto b_k\}$ angesehen, die zusammen mit einer Klausel $(l_1, \dots, l_{k+1})$ der ursprünglichen Klauselmenge S dafür sorgt, dass ein Literal eine bestimmte Belegung annehmen muss. Dies ist der Fall, wenn k der $k + 1$ Literale der Klausel unter V mit false belegt werden, und somit das letzte verbleibende Literal für eine erfüllende Belegung auf true gesetzt werden muss. Zeilen (8)-(12) dienen der Lemmagenerierung und dem nicht-chronologischen Backtracking. Wurde eine nicht erfüllende

```
(1)   boolean DPLL(ClauseSet $S$, Level $d$ )
(2)   {
(3)      while ( $S$ contains a unit clause $(L)$ ) {
(4)         register cause of $L$ becoming unit;          // conflict management
(5)         delete from $S$ clauses containing $L$;       // unit-subsumption
(6)         delete $\overline{L}$ from all clauses in $S$;           // unit-resolution
(7)      }
(8)      if ( $\emptyset \in S$ ) {                                 // empty clause?
(9)         generate conflict induced clause $C_C$;       // lemma generation
(10)        compute backjump-level; add $C_C$ to $S$;
(11)        return false;
(12)     }
(13)     if ( $S = \emptyset$ ) return true;                       // no clauses?
(14)     choose a literal $L_{d+1}$ occurring in $S$;          // case-splitting
(15)     if ( DPLL($S \cup \{(L_{d+1})\}$), $d+1$ ) return true;     // first branch
(16)     else if ( backjump-level < $d$) ) return false;
(17)     else if ( DPLL($S \cup \{(\overline{L}_{d+1})\}$), $d+1$) return true;  // second branch
(18)     else return false;
(19)  }
```

Abbildung 6.1: Sequentieller Davis-Putnam-Logemann-Loveland-Algorithmus mit Lemmagenerierung.

partielle Variablenbelegung, ein *Konflikt*, gefunden (es lässt sich dann die leere Klausel unter dieser Belegung ableiten), so wird eine neue Klausel (*Konfliktklausel*, *Lemma*) der Probleminstanz hinzugefügt, die das Auftreten weiterer Konflikte derselben Ursache verhindert. Dies wird durch Generierung einer minimalen, zur Herleitung der leeren Klausel ausreichenden, partiellen Variablenbelegung erreicht. Jede Erweiterung dieser minimalen Belegung wird dann durch das Hinzufügen des Lemmas für die weitere Suche ausgeschlossen. Die Untersuchung der Ursache eines Konflikts ermöglicht auch, auf Teile des Suchraums ohne Lösungen zu schließen. Davon kann durch die Berechnung eines “backjump-levels” Gebrauch gemacht werden (siehe auch [MSS96, BJS97]). In Zeile (14) wird nach nun abgeschlossener Problemvereinfachung ein Literal zur Fallunterscheidung ausgewählt. Die Auswahl dieses Literals kann entscheidenden Einfluss auf die Suchraumgröße und damit die Laufzeit des Algorithmus haben. In der Literatur wurden daher verschiedene Heuristiken zu dessen Auswahl vorgestellt [Fre95, HV95]. Wir haben in PaSAT fünf Strategien implementiert, die wir weiter unten ausführlich beschreiben werden. In den Zeilen (15)-(18) wird dann die eigentliche Fallunterscheidung vorgenommen und der DPLL-Algorithmus rekursiv aufgerufen, wobei der Klauselmenge S das ausgewählte Literal als Unit-Klausel nacheinander in beiden Polaritäten hinzugefügt wird. Der zweite rekursive Aufruf kann unterbleiben, wenn die Berechnung des Backjump-Levels dies ermöglicht. Ist die Probleminstanz S erfüllbar, so liefert Algorithmus DPLL `true` zurück, ansonsten `false`. Der

Algorithmus kann leicht dahingehend erweitert werden, in ersterem Fall auch eine oder mehrere erfüllende Belegungen (*Modelle*) zurückzuliefern.

Durch die rekursiven Aufrufe des DPLL-Algorithmus entsteht ein Aufrufbaum, dessen Knoten den Prozedurinstanzen entsprechen. Die Kanten können zusätzlich mit den zur Fallunterscheidung ausgewählten Literalen beschriftet werden. Diesen Aufrufbaum bezeichnen wir auch als *Suchbaum* des DPLL-Algorithmus.

Bei der Lemmagenerierung ist zu beachten, dass jedes Blatt des Suchbaums ein neues Lemma liefert. Da der Suchbaum exponentiell viele Blätter (in der Anzahl der Variablen) enthalten kann, können im schlimmsten Fall auch exponentiell viele Konfliktklauseln der Probleminstanz hinzugefügt werden. Um dieses exponentielle Wachstum zu verhindern versucht man daher, "gute" Konfliktklauseln mittels einer Heuristik auszuwählen. Eine einfache, aber oft implementierte Heuristik, fügt Lemmata nur hinzu, sofern sie eine bestimmte festgelegte Größe l (Anzahl der Literale) nicht überschreiten. Auch wir haben diese einfache Strategie in PaSAT implementiert und für unsere Experimente eine maximale Konfliktklausellänge von $l_1 = 10$ eingestellt. Darüberhinaus verwenden wir auch temporäre Konfliktklauseln zum nicht-chronologischen Backtracking, die so lange behalten werden, wie sie unter der aktuellen partiellen Variablenbelegung Unit-Klauseln sind. In unseren Experimenten haben wir diese temporären Konfliktklauseln bis zu einer maximalen Größe von $l_2 = 100$ berücksichtigt (siehe [MSS96] für Details zu dieser Methode).

Die Konfliktklausel eines sich ergebenden Konflikts ist nicht eindeutig bestimmt, so dass auch hier Variationsmöglichkeiten bestehen. Um ein eindeutiges Ergebnis zu erhalten, kann man entweder fordern, dass das Lemma nur Literale enthält, die zur Fallunterscheidung (DPLL Zeile (14)) ausgewählt wurden. Eine andere Möglichkeit wurde von Zhang beschrieben [Zha00]: Wir haben seine Methode übernommen, allerdings mit zusätzlicher Berücksichtigung von UIPs (*unique implication points*), wie sie von Marques Silva und Sakallah beschrieben wurde [MSS96, ZMMM01].

In PaSAT stehen die folgenden Literalauswahlstrategien zur Verfügung:

MO (maximal occurrences): Bei dieser Strategie wird ein Literal mit der größten Anzahl an Vorkommen in der aktuellen Klauselmenge ausgewählt. Dabei werden positive und negative Literale gleich gewichtet und die Summe aller Vorkommen betrachtet. Die Grundidee dieser Strategie zielt darauf ab, eine maximale Problemreduktion in der nachfolgenden Unit-Propagation-Phase zu erreichen.

MBO (maximal binary occurrences): Diese Strategie ähnelt der vorhergehenden, berücksichtigt aber nur Literalvorkommen in binären Klauseln (Klauseln mit zwei Literalen). Dadurch sollen Literale in diesen Klauseln bevorzugt belegt werden, wodurch sich weitere Unit-Klauseln ergeben. Sind keine binären Klauseln vorhanden, so wird Strategie MO verwendet.

SPC (shortest positive clause): Bei dieser Auswahlstrategie werden nur Literale berücksichtigt, die in positiven Klauseln (alle Literale positiv) vorkommen. Aus diesen wird ein beliebiges Literal in einer kürzesten Klausel ausgewählt. Sind keine positiven Klauseln mehr vorhanden, so ist das Problem trivial erfüllbar durch Belegung aller verbleibenden Variablen mit false.

SMB (shortest positive clause / max. binary occurrences): Diese Strategie versucht die Ideen der beiden vorhergehenden zu verbinden. Dazu werden in einem ersten Schritt alle Literale in kürzesten positiven Klauseln gesammelt, und von diesen dann in einem zweiten Schritt dasjenige ausgewählt, das unter sämtlichen binären Klauseln der Probleminstanz die höchste Anzahl an Vorkommen besitzt. Dabei werden die Anzahl $p(l)$ der positive Vorkommen und die Anzahl $n(l)$ der negativen Vorkommen eines Literals l getrennt betrachtet und durch die Bewertungsfunktion $f(l) = (p(l)+1) \cdot (n(l)+1)$ gewichtet. Ausgewählt wird dann das Literal, welches die Funktion f maximiert. Durch die multiplikative Gewichtung werden Literale mit ausgewogenem Verhältnis von positiven zu negativen Vorkommen bevorzugt.

SHM (shortest non-Horn clause / maximal binary occurrences): Auch diese Strategie läuft in zwei Schritten ab: Zuerst wird die Menge aller positiven Literale bestimmt, die in einer Testmenge bestehend aus den kürzesten nicht-Hornklauseln vorkommen. Aus diesen wird dann wiederum dasjenige mit der größten Anzahl an binären Vorkommen in der gesamten Klauselmenge ausgewählt. Dazu wird dieselbe Gewichtungsfunktion f wie bei der Strategie SMB verwendet. Die Verwendung von nicht-Hornklauseln rührt daher, dass die Erfüllbarkeit einer reinen Hornklauselmenge bereits durch Unit-Propagation entschieden werden kann.

Vergleiche der Performance von PaSAT mit SATO [ZBH96] sowie Laufzeiten verschiedener Benchmarks aus den Bereichen Kryptoanalyse und Bounded Model Checking (BMC) können Tabelle 6.1 (am Ende des Kapitels) oder [SBK01] entnommen werden.

6.2 Aspekte der Parallelisierung

Die Überprüfung der Konsistenzkriterien aus Abschnitt 4.1.3 erfordert die Durchführung einer enormen Zahl von aussagenlogischen Erfüllbarkeitstests. Für die Prüfung auf überflüssige Teile der Stückliste sind z.B. alleine für die Baureihe C210/FW mehr als 13.000 SAT-Tests durchzuführen. Eine (fast schon triviale) Parallelisierung kann also dadurch erreicht werden, dass man die einzelnen Tests nicht sequentiell verarbeitet, sondern auf verschiedene Prozessoren bzw. Rechner verteilt und dann parallel ausführt. Zur Minimierung der durchschnittlichen Wartezeit pro Testergebnis kann man dann zusätzlich Tests mit einer kurzen Laufzeit bevorzugt behandeln. Dazu werden Erfüllbarkeitstests, die nach einer bestimmten Zeit kein Ergebnis geliefert haben, abgebrochen und erst nach Durchführung aller anderen Tests wieder weiter bearbeitet. Diese einfache Form der Parallelisierung profitiert davon, dass keine Datenabhängigkeiten zwischen den einzelnen parallel auszuführenden Instanzen bestehen. Eine Implementation dieses Verfahrens (in einer Studienarbeit) lieferte gute Ergebnisse.

Bei härteren Erfüllbarkeitsproblemen kann es aber durchaus hilfreich sein, auch einzelne Instanzen weiter zu zerlegen und Teile des Suchraums parallel auf erfüllende Belegungen hin zu untersuchen. In Verbindung mit der Lemmagenerierung und dem dadurch möglich werdenden Austausch neu gewonnener struktureller Information über die Probleminstanz zwischen einzelnen Prozessen ergibt sich so eine Beschleunigungsmöglichkeit des Algorithmus, die im sequentiellen Fall nur schwer umzusetzen wäre.

PaSAT erlaubt die parallele und verteilte Bearbeitung des Erfüllbarkeitsproblems durch eine Suchraumaufteilung, die auf den von Zhang eingeführten “guiding paths” beruht [ZBH96]. Wir wollen diese hier nur kurz vorstellen, und ansonsten auf die Literatur verweisen [BSK03]. Ein Guiding Path ist ein Pfad im Suchbaum, der durch die Literalbeschriftung der Kanten dieses Baums eindeutig bestimmt ist. Jeder Guiding Path startet an der Wurzel des Suchbaums und hat darüberhinaus zu jeder Kante noch Angaben zum aktuellen Zustand der Suche gespeichert. Diese besteht aus der Information, ob das Literal dieser Kante den ersten oder zweiten rekursiven Aufruf des DPLL-Algorithmus repräsentiert, ob also in dem übergeordneten Knoten noch Backtracking erforderlich ist oder nicht. Ersteres wird durch den Zusatz B zur Kante erkenntlich gemacht, letzteres durch den Zusatz von N. Ein Guiding Path P besteht also aus einer Sequenz von Paaren $P = ((L_1, b_1), \ldots, (L_k, b_k))$, wobei L_i die zur Fallunterscheidung ausgewählten Literale (oder deren Negate) sind und $b_i \in \{\mathsf{B}, \mathsf{N}\}$. In Abbildung 6.2 ist

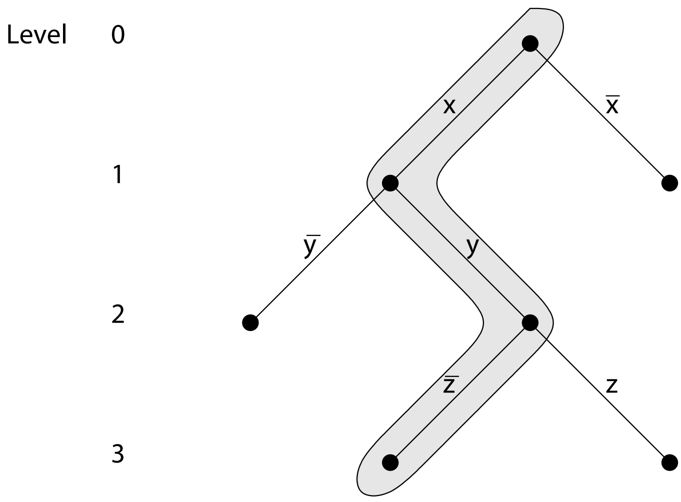


Abbildung 6.2: Guiding Path $((x, \mathsf{B}), (y, \mathsf{N}), (\overline{z}, \mathsf{B}))$, bei Traversierung des Suchraums in Präorder von links nach rechts.

ein solcher Guiding Path dargestellt.

Die Parallelisierung beruht nun daruf, noch zu bearbeitende Teile des Suchraums dynamisch abzuspalten und auf andere Prozesse zu verteilen. Befindet sich die Suche in einem Zustand, der durch einen Guiding Path P repräsentiert wird, und enthält dieser ein Literal mit Zusatz B, so kann dieser noch zu durchsuchende Teilbaum an einen anderen Prozess abgegeben werden. Damit verbunden ist auch eine Aufspaltung des Guiding Paths in zwei Pfade für die jeweiligen Prozesse. So kann der Guiding Path

$$P = ((L_1, b_1), \ldots, (L_{i-1}, b_{i-1}), (L_i, \mathsf{B}), (L_{i+1}, b_{i+1}), \ldots, (L_k, b_k))$$

aufgeteilt werden in zwei neue Pfade P_1 and P_2, wobei

$$P_1 = ((L_1, b_1), \ldots, (L_{i-1}, b_{i-1}), (\overline{L_i}, \mathsf{N}))$$

und

$$P_2 = ((L_1, b_1), \ldots, (L_{i-1}, b_{i-1}), (L_i, \mathsf{N}), (L_{i+1}, b_{i+1}, \ldots, (L_k, b_k)) \enspace .$$

Dabei hat es sich als vorteilhaft erwisen, zur Aufspaltung ein Literal kleinstmöglichen Levels zu wählen, da dadurch ein voraussichtlich größtmöglicher Teil des Suchraums abgespalten wird. Diese Form der Suchraumaufteilung erfordert nur minimale Änderungen am sequentiellen Algorithmus (wie z.B. Start des DPLL-Algorithmus an einer beliebigen, durch einen Guiding Path festgelegten Stelle des Suchraums). Durch die Verwendung der DOTS (Distributed Object-Oriented Threads System) Middleware [BKLW99], die das Fork/Join-Paradigma zur Parallelisierung verwendet, ergibt sich so eine natürliche Integration der parallelen Aspekte in den sequentiellen DPLL-Algorithmus.

Neben der Suchraumaufteilung ist der Austausch von Lemmata ein wesentlicher Bestandteil der parallelen Konzepte von PaSAT. Dabei werden die von einem Prozess (lokal) generierten Lemmata auch anderen Prozessen zur Verfügung gestellt. Um einen zu großen Overhead durch die zur Kommunikation zwischen den Prozessen benötigte Zeit zu vermeiden, werden nicht alle Lemmata ausgetauscht, sondern diese beim Absender und Empfänger gefiltert. Der Absender stellt nur solche Lemmata den anderen Prozessen zur Verfügung, die eine bestimmte Größe nicht überschreiten. Auf Empfängerseite wird darüberhinaus eine Filterung anhand des initialen Guiding Paths des Suchprozesses vorgenommen. Wird eine Klausel bereits vom initialen Guiding Path subsumiert, braucht sie nicht eingefügt zu werden.

Probleminstanz	t_{SATO}	t_{Seq}	t_{Par}	t_{PaSAT}	$s_{\mathrm{Par/Seq}}$	$s_{\mathrm{PaSAT/Seq}}$
DES-R3-B1-K1.1	949.93	725.5	20.69	19.23	35.07	37.73
DES-R3-B1-K1.2	631.07	1566.35	112.14	51.14	13.97	30.63
DES-R3-B2-K1.1	1.3	3.84	2.13	1.86	1.80	2.06
DES-R3-B2-K1.2	287.04	2.96	3.07	2.58	0.96	1.15
DES-R3-B3-K1.1	42.75	31.29	4.3	4.22	7.28	7.41
DES-R3-B3-K1.2	194.31	1480.47	297.26	64.01	4.98	23.13
DES-R3-B4-K1.1	492.36	2.75	2.79	2.42	0.99	1.14
DES-R3-B4-K1.2	495.48	10.15	10.04	9.48	1.01	1.07
longmult4	13.44	1.93	0.54	0.64	3.57	3.02
longmult5	26.29	7.46	2.31	2.31	3.23	3.23
longmult6	42.42	20.68	5.73	4.85	3.61	4.26
longmult7	160.99	73.78	22.24	15.49	3.32	4.76
longmult8	390.25	176.91	51.54	42.76	3.43	4.14
longmult9	577.60	291.00	79.51	77.78	3.66	3.74
longmult10	1586.71	414.49	113.92	138.71	3.64	2.99
longmult11	1625.74	541.14	145.81	179.28	3.71	3.02
longmult12	728.09	646.43	163.66	162.78	3.95	3.97
longmult13	2198.44	809.67	202.92	227.93	3.99	3.55
longmult14	1294.64	929.21	242.13	248.59	3.84	3.74
longmult15	863.51	895.94	235.29	298.54	3.81	3.00

Tabelle 6.1: Laufzeitmessungen von PaSAT.

Tabelle 6.1 zeigt experimentelle Ergebnisse, die wir mit PaSAT erhalten haben. Die Probleminstanzen stammen aus den Bereichen Kryptoanalyse und

Bounded Model-Checking. Wir haben die Experimente auf einer Sun Ultra E450 mit vier UltraSparcII Processoren (à 400 MHz) und einem Gigabyte Hauptspeicher unter Solaris 7 durchgeführt. Obwohl DOTS auch die Verteilung auf mehrere Maschinen erlaubt, haben wir von dieser Fähigkeit keinen Gebrauch gemacht und uns auf den lokalen Austausch von Klauseln über *shared memory* beschränkt.

Die Probleminstanzen der Kryptoanalyse stammen von F. Massacci[1] und codieren den DES-Verschlüsselungsalgorithmus mit drei Verschlüsselungsrunden. Bei diesen Problemen ist der Codierungsschlüssel zu finden, wobei eine bestimmte Anzahl von korrespondierenden Klartext/Schlüsseltext-Blöcken gegeben ist. Die Anzahl dieser Blöcke variiert zwischen 1 und 4 (wie vom B-Teil des Namens der Probleminstanz angegeben). Alle Probleminstanzen sind erfüllbar, haben aber nur wenige Modelle, meist nur ein einziges. Die SAT-Instanzen des Bounded Model-Checking stammen aus der Codierung der Äquivalent zweier unterschiedlicher Hardware-Designs eines 16-Bit-Multiplizierers[2]. Die 16 Dateien korrespondieren mit den 16 Ausgabebits der Multiplizierer. Alle Instanzen sind unerfüllbar.

In Tabelle 6.1 sind die Laufzeiten in Sekunden der sequentiellen Version (t_{Seq}), der parallelen Version ohne (t_{Par}) sowie mit (t_{PaSAT}) Lemmaaustausch angegeben. Zu Vergleichszwecken haben wir auch noch die mit SATO (in der Version 3.2 mit Standardeinstellungen) erhaltenen Laufzeiten mit angegeben (t_{SATO}). Die beiden letzten Spalten zeigen die Speed-ups der parallelen Versionen relativ zur sequentiellen Version an, einmal für die Version ohne Austausch von Konfliktklauseln ($s_{\text{Par/Seq}}$), einmal für die Version mit Austausch ($s_{\text{PaSAT/Seq}}$). Für die Probleme der Kryptoanalyse haben wir die Literalauswahlstrategie SPC, für die BMC-Probleme die Strategie MBO in PaSAT gewählt.

Da die Instanzen der Kryptoanalyse alle erfüllbar sind, ist es nicht verwunderlich, dass bei der parallelen Version superlineare Speed-ups im Vergleich zur sequentiellen Version zu beobachten sind: Die Wahrscheinlichkeit, dass ein Prozess “nahe” an einer Lösung der Suche beginnt, steigt im parallelen Fall linear mit der Anzahl der eingesetzten Prozessoren. Durch den zusätzlichen Einsatz des Lemmaaustauschs ergeben sich weitere, zum Teil beträchtliche Beschleunigungen, z.B. bei der Instanz DES-R3-B3-K1.2.

Bei den BMC-Instanzen erreichten wir in der parallelen Version ohne Lemmaaustausch Speed-ups bis zu 3.99. Im Durchschnitt der acht laufzeitin-

---

[1] http://www.dis.uniroma1.it/~massacci/cryptoSAT

[2] http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html

tensivsten Instanzen erhielten wir eine Effizienz der parallelen Version von über 93%. Bei Aktivierung des Lemmaaustauschs beobachteten wir eine breitere Streuung der Laufzeiten, teilweise sogar superlineare Speed-ups. Diese lassen sich durch zusätzliche Beschneidungen des Suchraums durch den Lemmaaustausch erklären. Diese werden dadurch ermöglicht, dass den einzelnen Prozessen nun Information zur Verfügung steht, die in der sequentiellen Version nur während eines geringeren Bruchteils des Suchprozesses verfügbar ist.

Eine ausführliche Beschreibung von PaSAT, die auch eine detaillierte Untersuchung verschiedener Einstellungsparameter auf das Laufzeitverhalten beinhaltet, kann in [BSK03] gefunden werden.

7

Zusammenfassung

In dieser Arbeit wurde die Anwendbarkeit formaler Methoden zur Konsistenzprüfung von Produktdokumentations- und Konfigurations-Systemen untersucht. Neben der ausführlichen Darstellung der Grundlagen – einschließlich einer Analyse der Anforderungen an ein Dokumentationssystem und an den darin verwendeten logischen Formalismus – war insbesondere die Evaluierung unserer Methoden anhand des Dokumentationsystems DIALOG von DaimlerChrysler wesentlicher Bestandteil.

Daher sehen wir als Hauptbeiträge dieser Arbeit die folgenden an:

- **Verifikation eines Regelsystems als Programmverifikation.** In dieser Sichtweise wird Verifikation nicht als eine ausschließlich statische Prüfung der Konsistenz der Regeln verstanden, sondern erfordert die Miteinbezehung der Mechanismen, welche die Regeln auswerten und ausführen. Motiviert ist dies durch die Feststellung, dass eine alleinige Betrachtung der Regeln zur Verifikation nicht ausreicht, da die Regelsemantik zum Teil erst durch die Verarbeitungsalgorithmen festgelegt wird. Nur durch die Untersuchung des Gesamtsystems bestehend aus Regeln und Verarbeitungssystem kann somit eine umfassende Verifikation sichergestellt werden.[1] Die Verifikationsaufgabe wird dadurch zu einer Programmverifkationsaufgabe, was sich auch in der Verwendung einer dynamischen Logik zur Formalisierung des Gesamtsystems widerspiegelt.

- **Formalisierung eines realen Konfigurationssystems in PDL.** Durch die Wahl von PDL ergab sich eine natürliche Formalisierung des regelbasierten Auftragsverabreitungsalgorithmus.[2] Zur rein logischen Darstellung des Verfahrens war die Abstraktion von Besonderheiten des

---

[1] Ähnliche Gedanken sind bei Waldinger und Stickel zu finden, die zur Verifikation von Expertensystemen die Entwicklung einer "system theory" empfehlen [WS92].

[2] Verwendung von PDL zur Verifikation regelbasierter Systeme hat sich auch anderweitig bewährt [SLSK02].

Systems oder deren Einarbeitung in die logische Codierung erforderlich. Die Verwendung einer modifizierten Kripke-Semantik und die Transformation von PDL nach Aussagenlogik erachten wir ebenfalls als einen wichtigen Beitrag.

- **Entwicklung geeigneter Konsistenztests.** Die Entwicklung geeigneter Konsistenztests, wie sie nun im Baubarkeits-Informations-System BIS realisiert sind, erforderte Kompromisse zwischen erforderlichem Produktwissen und Systematisierbarkeit. Sowohl statische als auch dynamische Aspekte (“Evolution der Dokumentation”) mussten berücksichtigt werden. Eine möglichst hohe Spezialisierung der Konsistenztests hat sich zur Akzeptanz des BIS-Systems als wesentlich herausgestellt.

- **Experimentelle Evaluierung der vorgeschlagenen Methoden.** Die direkte Umsetzung der Prüfungsmethoden in BIS erlaubte die Evaluierung unserer Verfahren anhand einer umfangreichen realen Dokumentation (DIALOG). Handhabbare Berechnungszeiten und praktische Verwendbarkeit ließen sich so sicherstellen.

- **Kompilierung und Komplexitätsanalyse mittels geordneter Resolution und aktiven Literalen.** Durch die Kompilierung mittels geordneter Resolution konnte einerseits (prinzipiell) die (bereits überraschend geringe) Rechenzeit einer großen Serie von Tests reduziert werden. Andererseits erlaubt die Kompilierung auch eine eingehendere Komplexitätsanalyse unter Verwendung des neu eingeführten Begriffs der aktiven Literale. Diese könnten für das Folgerungsproblem strukturierter Probleminstanzen neue handhabbare Teilklassen und neue Erklärungsansätze liefern.

A

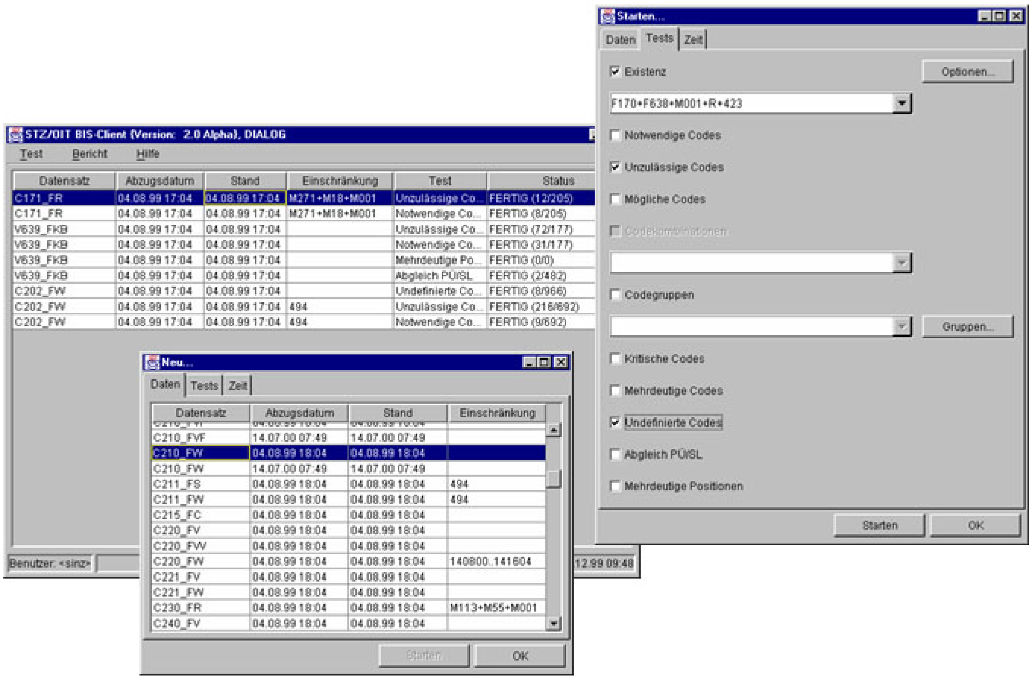
BIS: Baubarkeits-Informations-System

Das Baubarkeits-Informations-System BIS ist ein System zur Konsistenzprüfung der Produktdokumentationssysteme von DaimlerChrysler. Es ist in zwei Varianten verfügbar, die Daten aus unterschiedlichen Produktdokumentationssystemen (PDMS) beziehen: Zum einen gibt es eine Anbindung an das hier beschriebene PDM System DIALOG, das zur Dokumentation der Mercedes Fahrzeug- und Aggregate-Baureihen verwendet wird. Die zweite, in dieser Arbeit nicht näher beschriebene Variante greift auf Daten aus dem PDM System EDS/BCS zu, das eine ähnliche Dokumentationsmethode wie DIALOG verwendet und in DaimlerChryslers LKW-Bereich der Dokumentation der Atego- und Actros-Baureihen dient. BIS wurde im Rahmen dieser Arbeit in Zusammenarbeit mit A. Kaiser entwickelt.

BIS ist ein modulares Softwaresystem, das aus den drei Hauptkomponenten *Client*, *Server* und *Beweiser* besteht. Die Komponenten kommunizieren über CORBA/IDL-Schnittstellen und -Middleware und können in variabler Anzahl vorhanden sein. So können beispielsweise mehrere Beweiserkomponenten verwendet werden. Server und Beweiser sind in C++ geschrieben, der Client ist in Java codiert. Die grafische Benutzeroberfläche des Clients ist in Abbildung A.1 dargestellt.

Über den Client können Baureihen ausgewählt, Konsistenztests gestartet und die Ergebnisse betrachtet werden. Sämtliche in Abschnitt 4.1.1 beschriebenen Tests können mittels BIS durchgeführt werden. Darüberhinaus gibt es einen allgemeinen Erfüllbarkeitstest für Formeln der Form $\mathcal{B} \wedge F$, wobei F frei wählbar und $\mathcal{B}$ die in Abschnitt 4.1.4 definierte kombinierte Baubarkeits- und Zusteuerungsformel ist. Man kann dadurch also die Existenz von Aufträgen mit bestimmten Nebenbedingungen (durch F ausgedrückt) nach Zusteuerung und erfolgreicher Baubarkeitsprüfung, aber vor der Teilebedarfsermittlung überprüfen.

BIS greift auf die Produktdaten über eine Datei-Schnittstelle zu. Dazu müssen die DIALOG-Datenbanktabellen mit den relevanten Informationen in



Datensatz	Abzugsdatum	Stand	Einschränkung	Test	Status
C171_FR	04.08.99 17:04	04.08.99 17:04	M271+M18+M001	Unzulässige Co...	FERTIG (12/205)
C171_FR	04.08.99 17:04	04.08.99 17:04	M271+M18+M001	Notwendige Co...	FERTIG (8/205)
V639_FKB	04.08.99 17:04	04.08.99 17:04		Unzulässige Co...	FERTIG (72/177)
V639_FKB	04.08.99 17:04	04.08.99 17:04		Notwendige Co...	FERTIG (31/177)
V639_FKB	04.08.99 17:04	04.08.99 17:04		Mehrdeutige Po...	FERTIG (0/0)
V639_FKB	04.08.99 17:04	04.08.99 17:04		Abgleich PÜ/SL	FERTIG (2/482)
C202_FW	04.08.99 17:04	04.08.99 17:04		Undefinierte Co...	FERTIG (8/966)
C202_FW	04.08.99 17:04	04.08.99 17:04	494	Unzulässige Co...	FERTIG (216/692)
C202_FW	04.08.99 17:04	04.08.99 17:04	494	Notwendige Co...	FERTIG (9/692)

Benutzer: <sinz>

Datensatz	Abzugsdatum	Stand	Einschränkung
C210_FVF	14.07.00 07:49	14.07.00 07:49	
C210_FW	04.08.99 18:04	04.08.99 18:04	
C210_FW	14.07.00 07:49	14.07.00 07:49	
C211_FS	04.08.99 18:04	04.08.99 18:04	494
C211_FW	04.08.99 18:04	04.08.99 18:04	494
C215_FC	04.08.99 18:04	04.08.99 18:04	
C220_FV	04.08.99 18:04	04.08.99 18:04	
C220_FVV	04.08.99 18:04	04.08.99 18:04	
C220_FW	04.08.99 18:04	04.08.99 18:04	140800..141604
C221_FV	04.08.99 18:04	04.08.99 18:04	
C221_FW	04.08.99 18:04	04.08.99 18:04	
C230_FR	04.08.99 18:04	04.08.99 18:04	M113+M55+M001
C240_FV	04.08.99 18:04	04.08.99 18:04	

☑ Existenz
F170+F638+M001+R+423
☐ Notwendige Codes
☑ Unzulässige Codes
☐ Mögliche Codes
☐ Codekombinationen
☐ Codegruppen
☐ Kritische Codes
☐ Mehrdeutige Codes
☑ Undefinierte Codes
☐ Abgleich PÜ/SL
☐ Mehrdeutige Positionen

Abbildung A.1: Grafische Benutzeroberfläche des Baubarkeits-Informations-Systems.

Textdateien abgespeichert werden, auf die BIS dann zugreifen kann. Aus diesen erzeugt BIS die Baubarkeitsformel $\mathcal{B}$, die von den meisten Konsistenztests benötigt wird. Zur Durchführung eines Konsistenztests generiert BIS typischerweise eine Sequenz von aussagenlogischen Formeln, die dann der Reihe nach von dem Beweiser auf Erfüllbarkeit hin untersucht werden. Die verwendete Beweiserkomponente verarbeitet beliebige aussagenlogische Formeln und ist nicht auf eine vorherige Konvertierung in konjunktive Normalform (CNF) angewiesen. Der von A. Kaiser entwickelte Beweiser ist in einem technischen Report beschrieben [Kai01]. Die von diesem Beweiser verarbeitete Syntax umfasst auch den in Abschnitt 3.1.5 beschriebenen Selektionsoperator, der in BIS aber lediglich für die Untersuchung hinsichtlich Codegruppen eingesetzt wird. Eine weitere Besonderheit der Beweisereinheit ist die integrierte Erklärungskomponente, durch die die Fehlersuche in dem umfangreichen Regelsystem deutlich vereinfacht und somit auch weniger erfahrenen Benutzern ermöglicht wird.

Die gleichzeitige gemeinsame Nutzung eines BIS-Servers durch mehrere Anwender ist ebenfalls möglich, wobei Testergebnisse dann prinzipiell auch von anderen Benutzern miteingesehen werden könnten. Diese Möglichkeit ist in BIS bisher allerdings noch nicht integriert.

Eine Erweiterung von BIS für die in den Werken verwendete Produktionsversion von DIALOG (DIALOG/P) ist ebenfalls vorhanden. Diese berücksichtigt die zusätzlichen zeitlichen Einschränkungen der Regeln. Dadurch wird es möglich, alle in BIS vorhandenen Tests auf einen (auch in der Zukunft oder Vergangenheit liegenden) Stichtag zu beziehen, und die Tests relativ zu diesem durchzuführen. Ergänzende Perl-Skripte erlauben die Verwendung der $\pm\delta$- und der 3-Punkt-Methode zum Auffinden der Auswirkungen von Änderungen der Produktübersicht auf Teileebene (Anlaufteile, Auslaufteile, Alleinteile).

Das BIS-System ist ausführlich in dem Artikel [SKK03] beschrieben.

Testergebnisse

B

Wir stellen hier auszugsweise die Ergebnisse einiger BIS-Konsistenztests vor. Für die Entwicklungsdokumentation und deren statische Analyse haben wir Ergebnisse der sechs Konsistenztests aus Abschnitt 4.1.1 für die Baureihen C202/FW und C210/FS ausgewählt. Für die dynamische Analyse der Produktionsdokumentation stellen wir Testergebnisse der 3-Punkt-Methode, angewandt auf die Baureihe C210/FS, vor.

Die Konsistenztests beruhen auf Datenmaterial vom 4. August 1999 (statische Analyse) und vom 22. Mai 2001 aus Werk 50 (dynamische Analyse).

B.1 Statische Analyse

B.1.1 Baureihe C202/FW

Notwendige Codes (NC): F202, HA, VL, VR (4 von 694)

Alle vier notwendigen Codes werden baureihengebunden zugesteuert und sind beabsichtige notwendige Codes.

Unzulässige Codes (UC): M010 (1 von 694)

Eine Analyse mit Hilfe der in BIS eingebauten Erklärungskomponente ergibt, dass M010 nur baubar ist, falls M111 und 801 im Auftrag vorhanden sind. Code 801 ist aber nicht baubar (keine Baubarkeitsbedingung vorhanden). Der Grund hierfür kann sein, dass Code M010 erst mit einem Modelljahrwechsel freigegeben werden soll, jedoch bereits (ohne weitere zeitliche Einschränkungen) dokumentiert wurde.

Stabilität der Zusteuerung (SZ): 200B, 203B, 207B, 208B, 212B, 231B, 232B, 234B, 235B, 236B, 2XXL, 423, 652, 657, 6XXL, 772, 7XXL, 956, 9XXL (19 von 125)

Die Zusteuerung eines jeden dieser Codes kann einen baubaren Auftrag in einen nicht mehr baubaren abändern. Im Fall des Codes 956 zum Beispiel wird der Auftrag bestehend aus den Codes (231, 249, 254, 265, 345, 404, 441, 460, 494, 543, 580A, 581A, 800, 820, 873, 904U, 934, 989, K08, L, M113, M43) durch Hinzufügen des Codes 956 nicht mehr baubar, da 904U nur baubar ist, falls 956 nicht im Auftrag vorkommt.

Reihenfolgeunabhängigkeit der Zusteuerung (RZ): ⟨423, GM⟩, ⟨918, 925⟩ (2 von 7750)

Zu jedem dieser Codepaare gibt es einen Auftrag, bei dem beide Codes einzeln, aber nicht gemeinsam zugesteuert werden können. Daher stellt sich hier die Frage, welcher Code wirklich dem Auftrag hinzugefügt wird. Im Falle der Mehrdeutigkeit ⟨423, GM⟩ ergibt sich folgendes: Bei Vorhandensein des Codes M605 in einem Auftrag wird GM baureihengebunden zugesteuert, 423 wird codegebunden zugesteuert bei 625+M605, so dass aufgrund der Prioritätsregelung Code 423 zugesteuert wird. An einem Beispielauftrag (200A, 201A, 209, 222, 345, 404, 551, 570, 580, 584, 625, 666, 673, 724, 752, 762, 800, 873, 932, 955, 989, K08, M002, M25, M605, R) zeigt sich darüberhinaus, dass dieser bei Zusteuerung von GM nicht mehr baubar ist, bei Zusteuerung von 423 aber weiterhin. Es handelt sich also hier um keinen Dokumentationsfehler.

Überflüssige Teile/Positions-Varianten (UT):

> 0104411167A0014, 3300520266A0065, 3300520267A0075,
> 3300520270A0065, 3300520271A0075, 3300520296A0065,
> 3300520297A0075, 3309420200A0065, 3309420201A0075,
> 3309420204A0065, 3309420205A0075, 3500020244A0041,
> 3500020253A0041, 3500420222A0041, 3500420231A0041,
> A0009899203, A0010102000, A0010102200, A0010106600,
> A0015408697, A0019908312, A0020101500, A0020101600,
> ...
> A6042050106, A6111500099, A6112340239,
> H000029837182, H000029837482, H000029838082,
> HWA2028810101, HWA2028810201, N000433006402,

N000931008182, N007982004256, N304017005005,
N912010003000, N915003008201, N915017010204

(322 nicht benötigte Teile bei 18508 Positionsvarianten)

Teil N915017010204 kommt beispielsweise nur an einer einzigen Positionsvariante vor und besitzt dort die (kurze) Coderegel M604+423, Code M604 ist aber für diese Baureihe nicht freigegeben. Für das Bauteil 3500020253A0041 ist die Coderegel M111+M23+423+-M001 spezifiziert, wobei M23 die Baubarkeitsbedingung M111+(494/704/706/707/715/719) +-M001 besitzt. Keiner der Codes 494...719 ist aber in Kombination mit M23, M111 und ohne M001 baubar. Dieser Konflikt entsteht erst im Zusammenspiel von mehr als 30 Regeln und ist daher ohne genaue Produktkenntnis oder spezielle Programmunterstützung nur schwer zu überschauen.

Mehrdeutigkeiten in der Stückliste (MT):

Modul	Position	Pos.-Varianten	Grund f. Mehrdeutigkeit
020808	0160	⟨0002,0003⟩	Coderegel mit neg. Code
020808	0161	⟨0002,0003⟩	Coderegel mit neg. Code
020808	0168	⟨0005,0003⟩	Coderegel mit neg. Code
⋮	⋮	⋮	
040484	0355	⟨0001,0002⟩	identische Coderegeln
040804	0025	⟨0003,0006⟩	CR disj. in anderer enth.
⋮	⋮	⋮	
903212	0200	⟨0001,0002⟩	identische Coderegeln

(142 von 37258)

Die häufigsten Ursachen von Mehrdeutigkeiten sind identische Coderegeln oder durch die spezielle Behandlung von negierten Codes verursachte Überschneidungen.

B.1.2 Baureihe C210/FS

Notwendige Codes (NC): 584, F210, HA, VL, VR (5 von 588)

Alle diese Codes werden baureihengebunden zugesteuert, finden bis auf 584 hauptsächlich zu internen Dokumentationszwecken Verwendung und sind absichtlich in jedem Auftrag vorhanden. Entweder Code 584 oder 583 muss in jedem Auftrag vorhanden sein, Code 583 ist aber nicht freigegeben.

Unzulässige Codes (UC): Z90 (1 von 588)

Code Z90 darf nur zusammen mit 992 verbaut werden, dieser Code ist aber nicht freigegeben.

Stabilität der Zusteuerung (SZ): 2XXL, 6XXL, 7XXL, 8XXL, 954, 9XXL (6 von 132)

Exemplarisch wollen wir die Zusteuerung von 954 genauer betrachten. Dem Auftrag (293, 345, 352, 414, 480, 498, 500, 521A, 524, 551, 600, 617, 623U, 675L, 722, 761, 801, 810, 819, 820, 860, 873, 877, 882, 989, L, M113, M55, X50) wird z.B. wegen des Enthaltenseins von M113 und M55 der Code 954 (baureihengebunden) zugesteuert. Danach ist aber die Baubarkeitsbedingung -954/954+691 von Code 345 nicht mehr erfüllt, da Code 691 nicht im Auftrag vorkommt. Code 691 hat in diesem Fall zwar eine erfüllte Zusteuerbedingung (954+345/494+M55), diese wird aber nach der Zusteuerung von Code 954 nicht mehr aktiv, da sie codegebunden ist.

Reihenfolgeunabhängigkeit der Zusteuerung (RZ): ⟨265, 954⟩ (1 von 8646)

Dem Beispielauftrag (293, 345, 352, 414, 480, 498, 500, 521A, 524, 551, 600, 617, 675L, 722, 761, 801, 810, 819, 820, 860, 873, 877, 882, 989, L, M113, M55, X50) kann sowohl 265 als auch 954 zugesteuert werden. Gemeinsame Zusteuerung beider Codes ist nur bei anfänglicher Zusteuerung von 265 und nachfolgender Zusteuerung von 954, nicht aber in der anderen Reihenfolge möglich, da Code 265 die Zusteuerungsbedingung 494/498+-954 besitzt. Da Code 265 codegebunden, also priorisiert, zugesteuert wird, tritt der erste Fall ein, es werden also beide Codes zugesteuert.

Überflüssige Teile/Positions-Varianten (UT):

> 0100311122A0014, 0100311137A0014, 0100311918A0014,
> 0100360510A0014, 0100360511A0014, 0100360518A0014,
> 0100360607A0014, 0100360608A0014, 3500021048A0043,

3500021050A0043, 3500021070A0043, 3500021080A0043,
3500121025A0043, A0000940848, A0001531279,
...
N912002008001, N912006005003, N914007010058,
N914007010071, N914025004108, N914127004310,
N914130006201, N915003013202, N915010008202,
N916002013100, N916002015100, N916016015202

(1072 nicht benötigte Teile bei 23324 Positionsvarianten)

Teil 0100311122A0014 kommt beispielsweise an 45 Positionsvarianten in Modul 490004 vor. Die Coderegel einer jeder dieser Varianten enthält aber Code 808 oder 809. Keiner von beiden ist freigegeben und somit ist auch keiner baubar.

Mehrdeutigkeiten in der Stückliste (MT):

Modul	Position	Pos.-Varianten	Grund f. Mehrdeutigkeit
020004	0600	⟨0003,0005⟩	Coderegel mit neg. Code
020004	0600	⟨0004,0006⟩	Coderegel mit neg. Code
020008	0010	⟨0001,0002⟩	doppelte Default-Regel
⋮	⋮	⋮	
240412	0410	⟨0010,0130⟩	überlappende CR
240412	0505	⟨0040,0060⟩	CR disj. in anderer enth.
⋮	⋮	⋮	
901212	7040	⟨0001,0002⟩	ident. Coderegeln

(269 von 43585)

B.2 Dynamische Analyse

Für die dynamische Analyse haben wir die 3-Punkt-Methode zur Simulation eines Modelljahrwechsels verwendet. Dazu simulierten wir den Auslauf von Code 801 zum 30.06.2001 unter gleichzeitiger Festsetzung von Code 802 als Seriencode. Dies entspricht der Vorgehensweise bei DaimlerChrysler, vor dem Termin des Modelljahrwechsels (30.06.) ein Zeitintervall einzuschieben, in dem sowohl Produktvarianten des alten als auch des neuen

Modelljahres gültig sind. Nach dem 30.06. sind nur noch die Varianten des neuen Modelljahres gültig, die Dokumentation der Übergangsphase wird dann Schritt für Schritt entfernt.

Wir verwendeten die 3-Punkt-Methode aus Abschnitt 4.2.3 für die Zeitpunkte $t_0 = 30.06.2001$ und $t_1 = 01.07.2001$, zusammen mit der modifizierten Produktübersicht $\mathcal{B}^*(I, E, t_1)$ mit $I = \{801, 802\}$ und $E = \neg x_{801} \wedge x_{802} \wedge B_{802}(t_1)$. Wir verglichen dann den Teilebedarf vor und nach der simulierten Änderung (d.h. P_{t_0} und $P^*_{t_1}$).

Die Änderungen sind aufgeschlüsselt nach Codes und Teilen, die nach der Änderung wegfallen bzw. neu hinzukommen. In unserem Beispiel ergab sich folgendes:

Wegfallende Codes (Auslaufcodes):

> 023U, 024U, 029U, 034U, 036U, 143U, 182U, 293, 345, 349, 414, 460, 494, 551, 644, 677, 680, 681, 691, 735, 750A, 751A, 752A, 756A, 760U, 800, 801, 816U, 882, 891U, 960U, K01, K02, P32, P33, P34, W31, X00, X02, X03, X04, X05, X07, X09, X22, X23, X26, X31, X34, X35, X38, X39, X41, X44, X45, X46, X50, X52, X53, X54, X55, X56, X57, X58, X59, X62, X63, X64, X68, X69, X70, X71, X75, Y94, Y95, Y96, Z50, Z51

(78 von 583)

Wegfallende Teile (Auslaufteile):

> 3500021087A0043, 3500121014A0043, A0004316707, A0004761759, A0005904612, A0009900729, A0009902310, A0009905678, A0009907454, A0009977974, A0019895051, A0059978690, A0069974390, A0069977548, A0101542902,
> ...
> PB210911927P, PB210912920P, PB210912922P, PB210912923P, PB611500025P, PB612500002P

(504 von 9014)

Häufigster Grund eines Teilewegfalls ist, dass keine seiner Coderegeln mehr von einem gültigen Auftrag ausgelöst wird (471 der 504 Fälle). So ist z.B.

Teil PB210912923P an zwei Positionsvarianten dokumentiert, deren Coderegeln beide den nicht mehr gültigen Code 750A enthalten. In den verbleibenden 36 Fällen (z.B. auch bei Teil A0101542902) sind die Teile aufgrund der zeitlichen Gültigkeitsintervalle der Regeln nicht mehr in der Dokumentation sichtbar.

Neu hinzukommende Codes (Anlaufcodes): keine

Neu hinzukommende Teile (Anlaufteile): keine

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Baa96] F. Baader. Using automata theory for characterizing the semantics of terminological cycles. *Annals of Mathematics and Artificial Intelligence*, 18:175–219, 1996.

[BCF$^+$01] R. Béjar, A. Cabiscol, C. Fernández, F. Mànya, and C. Gomes. Extending the reach of SAT with many-valued logics. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[BCH90] E. Boros, Y. Crama, and P.L. Hammer. Polynomial-time inference of all valid implications for Horn and related formulae. *Annals of Mathematics and Artificial Intelligence*, 1:21–32, 1990.

[BCP97] E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Trans. on Database Syst.*, 21(1):1–29, 1997.

[BÉM$^+$97] Y. Boufkhad, G. Éric, P. Marquis, B. Mazure, and S. Lakhdar. Tractable cover compilations. In *Proc. of the 15th Intl. Joint Conf. on Artificial Intelligence (IJCAI'97)*, pages 122–127, Nagoya, Japan, August 1997.

[BG01] D. Basin and H. Ganzinger. Automated complexity analysis based on ordered resolution. *Journal of the ACM*, 48(1):70–109, 2001.

[BH91] F. Baader and Ph. Hanschke. A schema for integrating concrete domains into concept languages. In *Proc. 12th Joint Conf. on Arificial Intelligence (IJCAI-91)*, pages 452–457, 1991.

[BJS97] R.J. Bayardo Jr. and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. 14th Nat. Conf. on Artificial Intelligence (AAAI'97)*, pages 203–208. AAAI Press, 1997.

[BKLW99] W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[BL85] R.J. Brachman and H.J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

[BL00] M. Benedikt and L. Libkin. Safe constraint queries. *SIAM J. on Computing*, 29(5):1652–1682, 2000.

[BMNPS03] F. Baader, D. McGuinness, P. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[BN03] F. Baader and W. Nutt. Basic description logic. In Baader et al. [BMNPS03], pages 47–100.

[BOBS89] V.E. Barker, D.E. O'Connor, J. Bachant, and E. Soloway. Expert systems for configuration at Digital: XCON and beyond. *Comm. ACM*, 32(3):298–318, 1989.

[Bor96] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82:353–367, 1996.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS85] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[BS96] M. Boehm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[BS99] F. Baader and U. Sattler. Expressive number restrictions in description logics. *J. of Logic and Computation*, 9(3):319–350, 1999.

[BSK03] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[Cau90] D. Caucal. Graphes canonique de graphes algébraiques. *Informatique théoretique et Applications (RAIRO)*, 24(4):339–352, 1990.

[CBRZ01] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC92] M. Conforti and G. Cornuéjols. A class of logical inference problems solvable by linear programming. In *Proc. 33rd Symp. on Foundations of Computer Science (FOCS'92)*, pages 670–675. IEEE Computer Society Press, 1992.

[CD97] M. Cadoli and F.M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3–4):137–150, 1997.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[CGS+89] R. Cunis, A. Günter, I. Syska, H. Peters, and H. Bode. PLAKON – an approach to domain-independent construction. In *Proc. 2nd Intl. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 866–874, Tullahoma, TN, June 1989. ACM Press.

[CH91] V. Chandru and J.N. Hooker. Extended horn sets in propositional logic. *Journal of the ACM*, 38(1):205–221, 1991.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *3rd Symp. on Theory of Computing*, pages 151–158. ACM Press, 1971.

[DAM01] *DAML+OIL (March 2001) Reference Description*, 2001. W3C Note 18 December 2001. Available at `http://www.w3.org/TR/daml+oil-reference`.

[Dan83] E. Dantsin. Two systems for proving tautologies, based on the split method. *J. Sov. Math.*, 22:1293–1305, 1983. Original Russian article appeared in 1981.

[Dav87] S. M. Davis. *Future Perfect*. Addison-Wesley, 1987.

[DGM96] G. De Giacomo and F. Massacci. Tableaux and algorithms for propositional dynamic logic with converse. In *13th Intl. Conf. on Automated Deduction (CADE'96)*, pages 613–627, New Brunswick, NJ, 1996. Springer.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DP89] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[dV94] A. del Val. Tractable databases: How to make propositional unit propagation complete through compilation. In *Proc. of the 4th Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 551–561, Bonn, Germany, May 1994.

[dV00] A. del Val. On some tractable classes in deduction and abduction. *Artificial Intelligence*, 116(1–2):297–313, 2000.

[FBC$^+$87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbæk, B. Mahbod, M.-A. Neimat, T.A. Ryan, and M.-C. Shan. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, 1987.

[FFJ00] A. Felfernig, G.E. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. *Intl. J. Software Engineering and Knowledge Engineering*, 10(4):449–469, 2000.

[FFJ+03] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker. Configuration knowledge representations for semantic web applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, 17(1):31–50, January 2003. Special issue on configuration.

[FFJS00] A. Felfernig, E.F. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. In *Proc. 14th European Conf. on Artificial Intelligence*, pages 146–150, Berlin, Germany, August 2000. IOS Press.

[FG03] J. Franco and A. Van Gelder. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Applied Mathematics*, 125(2–3):177–214, 2003.

[FL77] M.J. Fischer and R.E. Ladner. Propositional modal logic of programs. In *Proc. 9th ACM Symp. Theory of Comput.*, pages 286–294, 1977.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18:194–211, 1979.

[FLTZ93] C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution Methods for the Decision Problem*, volume 679 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1993.

[Fre95] J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, May 1995.

[Gab77] D. Gabbay. Axiomatizations of logics of programs, 1977. Unpublished.

[Gal86] J. H. Gallier. *Logic for Computer Science*. Harper & Row, New York, 1986.

[GLP93] G. Gallo, G. Longo, and S. Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.

[GOR97] E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Proc. 12th IEEE Symp. on Logic in Computer Science (LICS'97)*, pages 306–317. IEEE Computer Society Press, 1997.

[GW94] I. Gent and T. Walsh. The SAT phase transition. In *Proc. 11th European Conference on Artificial Intelligence (ECAI'94)*, pages 105–109, Amsterdam, The Netherlands, 1994. John Wiley and Sons.

[Haa98] A. Haag. Sales configuration in business processes. *IEEE Intelligent Systems*, 13(4):78–85, July/August 1998.

[Han92] Ph. Hanschke. Specifying role interaction in concept languages. In *Proc. 3rd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'92)*, pages 318–329. Morgan Kaufmann, 1992.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, pages 507–544. Kluwer, 1984.

[Hin62] J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, 1962.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HN90] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Technical Report RR-90-04, Deutsches Forschungszentrum für künstliche Intelligenz (DFKI), Kaiserslautern, Germany, 1990.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580,583, 1969.

[HR85] F. Hayes-Roth. Rule based systems. *Comm. ACM*, 28(9):921–932, 1985.

[HS82] D. Harel and R. Sherman. Looping vs. repeating in dynamic logic. *Information and Control*, 55(1–3):175–192, 1982.

[HV95] J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[HW74] L. Henschen and L. Wos. Unit refutations and horn sets. *Journal of the ACM*, 21(4):590–605, 1974.

[JCU01] B. Jurkowiak, Li. C.M., and G. Utard. Parallelizing Satz using dynamic workload balancing. In H. Kautz and B. Selman, editors, *LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[JH98] W.E. Jüngst and M. Heinrich. Using resource balancing to configure modular systems. *IEEE Intelligent Systems*, 13(4):50–58, July/August 1998.

[Joy69] W.H. Joyner. Resolution strategies as decision procedures. *Journal of the ACM*, 23(1):398–417, 1969.

[Jun01] U. Junker. Preference programming for configuration. In *Configuration Workshop Proceedings, 17th Intl. Joint Conf. on Artificial Intelligence (IJCAI-2001)*, pages 50–56, Seattle, WA, August 2001.

[Kai01] A. Kaiser. A SAT-based propositional prover for consistency checking of automotive product data. Technical report, Wilhelm-Schickard-Institut für Informatik, Eberhard-Karls-Universität Tübingen, Sand 13, 72076 Tübingen, Germany, 2001. Technical Report WSI-2001-16.

[Kay79] M. Kay. Functional grammar. In *Proc. 5th Annual Meeting of the Berkeley Linguistics Society*, Berkeley, CA, 1979. Berkeley Linguistics Society.

[KB82] R.M. Kaplan and J. Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–381. MIT Press, 1982.

[KBG89] W. Kim, E. Bertino, and J.F. Garza. Composite objects revisited. *ACM SIGMOD Record*, 18(2), 1989.

[KBL94] H. Kleine Büning and T. Lettmann. *Aussagenlogik: Deduktion und Algorithmen*. B.G. Teubner Stuttgart, 1994.

[KH69] R. Kowalski and P.J. Hayes. Semantic trees in automated theorem proving. In *Machine Intelligence 4*, pages 87–101. Edinburgh University Press, 1969.

[KK01] A. Kaiser and W. Küchlin. Detecting inadmissible and necessary variables in large propositional formulae. In *Proc. Intl. Joint Conf. on Automated Reasoning: IJCAR 2001—Short Papers*, number 11/01 in Technical Report DII, pages 96–102, Siena, Italy, June 2001. University of Siena.

[KKR90] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51:26–52, 1990.

[Kök94] T. Kökény. A new arc consistenc algorithm for CSPs with hierarchical domains. In *Proc. 6th Intl. Conf. on Tools with Artificial Intelligence*, pages 439–445, New Orleans, LA, November 1994. IEEE Computer Society.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KS00] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, February 2000.

[Kul00] O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.

[Lut01] C. Lutz. Nexptime-complete description logics with concrete domains. In *Proc. 17th Intl. Joint Conf. on Automated Reasoning (IJCAR-2001)*, number 2083 in LNAI, pages 45–60. Springer, 2001.

[Mai98] D. Mailharro. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, 12(4):383–397, 1998.

[Mar95] P. Marquis. Knowledge compilation using theory prime implicates. In *Proc. of the 14th Intl. Joint Conf. on Artificial Intelligence (IJCAI'95)*, pages 837–845, Montréal, Canada, August 1995.

[Mar00] P. Marquis. Consequence finding algorithms. In D.M. Gabbay and Ph. Smets, editors, *Handbook of Defeasable Reasoning and Uncertainty Management Systems*, volume 5, pages 41–145. Kluwer, 2000.

[MBS$^+$93] D.P. Miranker, F.H. Burke, J.J. Steele, J. Kolts, and D.R. Haug. The C++ embeddable rule system. *Intl. J. on Artificial Intelligence Tools*, 2(1):33–46, 1993.

[McC80] J.L. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1–2):27–39, 1980.

[McD82] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[McG03] D.L. McGuinness. Configuration. In Baader et al. [BMNPS03], pages 397–414.

[MF89] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proc. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pages 1395–1401, Detroit, MI, August 1989.

[MF90] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. of the 8th Nat. Conf. on Artificial Intelligence*, pages 25–32, Boston, MA, 1990. AAAI Press.

[Min81] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind Design*. MIT Press, 1981.

[MSS96] J.P. Marques Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proc. IEEE Intl. Conf. on Tools with Artificial Intelligence (ICTAI'96)*, pages 467–469, Toulouse, France, November 1996.

[Neb90] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.

[Neb91] B. Nebel. Terminological cycles: Semantics and computational properties. In J.F. Sowa, editor, *Principles of Semantic Networks*, pages 331–361. Morgan Kaufmann, 1991.

[New42] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". In *Annals of Mathematics*, volume 43, pages 223–243. Princeton University Press, 1942.

[OWL02] *Web Ontology Language (OWL) Reference Version 1.0*, 2002. W3C Working Draft 12 November 2002. Latest version available at `http://www.w3.org/TR/owl-ref`.

[Pac94] F. Pachet. Proceedings of the OOPSLA'94 workshop on embedded object-oriented production systems (EOOPS). Technical Report 94/24, Laboratoire Formes et Intelligence Artificielle, Institut Blaise Pascal, Paris, France, December 1994.

[Par78] R. Parikh. The completeness of propositional dynamic logic. In *Proc. 7th Symp. on Math. Found. of Comput. Sci.*, volume 64 of *Lect. Notes in Comput. Sci.*, pages 403–415. Springer-Verlag, 1978.

[Pra79] V.R. Pratt. Models of program logics. In *Proc. 20th IEEE Symp. Comput. Sci.*, pages 115–122, 1979.

[Pre96] D. Pretolani. Hierarchies of polynomially solvable satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 17(3–4):339–357, 1996.

[Pri67] A.N. Prior. *Past, Present and Future*. The Clarendon Press, Oxford, 1967.

[PST00] L. Pacholski, W. Szwast, and L. Tendera. Complexity results for first-order two-variable logic with counting. *SIAM J. on Computing*, 29(4):1083–1117, 2000.

[PVG00] T.J. Park and A. Van Gelder. Partitioning methods for satisfiability testing on large formulas. *Information and Computation*, 162:179–184, 2000.

[Qui67] M.R. Quillian. Word concepts: A theory and simulation of some basic capabilities. *Behavioral Science*, 12:410–430, 1967.

[RD00] I. Rish and R. Dechter. Resolution versus search: Two strategies for SAT. *J. Automated Reasoning*, 24(1–2):225–275, February 2000.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[SBJ87] E. Soloway, J. Bachant, and K. Jensen. Assessing the maintainability of XCON-in-RIME: Coping with the problems of a VERY large rule-base. In *Proc. of the 6th Nat. Conf. on Artificial Intelligence*, pages 824–829, Seattle, WA, 1987. AAAI Press.

[SBK01] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[Sch91] K. Schild. A correspondance theory for terminological logics: Preliminary report. In *Proc. 12th Joint Conf. on Arificial Intelligence (IJCAI-91)*, pages 466–471, 1991.

[SE89] R.S. Streett and E.A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–264, 1989.

[Seg71] K. Segerberg. An essay in classical modal logic. *Filosofiska Studier* 13(1–3), Department of Philosophy, Uppsala University, Sweden, 1971.

[Seg77] K. Segerberg. A completeness theorem in the modal logic of programs (preliminary report). *Notices Amer. Math. Soc.*, 24(6):A–552, 1977.

[SF96] D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. In G.F. Luger, editor, *Proc. Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, Albuquerque, NM, 1996. AAAI Press.

[Sin97] C. Sinz. Baubarkeitsprüfung von Kraftfahrzeugen durch automatisches Beweisen. Diplomarbeit, Universität Tübingen, December 1997.

[Sin02] C. Sinz. Knowledge compilation for product configuration. In *Configuration Workshop Proceedings, 15th European Conference on Artificial Intelligence (ECAI-2002)*, pages 23–26, Lyon, France, July 2002.

[SK94] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *JACM*, 43(2):193–224, 1994.

[SK01] C. Sinz and W. Küchlin. Dealing with temporal change in product documentation for manufacturing. In *Configuration Workshop Proceedings, 17th Intl. Joint Conf. on Artificial Intelligence (IJCAI-2001)*, pages 71–77, Seattle, WA, August 2001.

[SK02] C. Sinz and W. Küchlin. *Formale Produktdokumentation für Magnetresonanztomografen*. Steinbeis Technologietransferzentrum Objekt- und Internet-Technologien an der Universität Tübingen, 2002. Machbarkeitsstudie.

[SKK00] C. Sinz, A. Kaiser, and W. Küchlin. SAT-based consistency checking of automotive electronic product data. In *Configuration Workshop Proceedings, 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 74–78, Berlin, Germany, August 2000.

[SKK03] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, 17(1):75–97, January 2003. Special issue on configuration.

[SLSK02] C. Sinz, T. Lumpp, J. Schneider, and W. Küchlin. Detection of dynamic execution errors in IBM System Automation's rule-based expert system. *Information and Software Technology*, 44(14):857–873, November 2002.

[Smo88] G. Smolka. A feature logic with subsorts. LILOG Report 33, IWBS, IBM Deutschland, Stuttgart, Germany, May 1988.

[SS89] M. Schmidt-Schauß. Subsumption in KL-ONE is undecidable. In *Proc. 1st Intl. Conf. on the Principles of Knowledge Representation and Reasoning (KR'89)*, pages 421–431, Los Altos, 1989. Morgan Kaufmann.

[SSS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[Ste95] M. Stefik. *Introduction to Knowledge Systems*. Morgan Kaufmann, 1995.

[Str82] R.S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1/2):121–141, 1982.

[Stu97] M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2):111–125, 1997.

[SW98] D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems*, 13(4):42–49, July/August 1998.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 55:285–309, 1955.

[TG87] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, 1987.

[Tru98] K. Truemper. *Effective Logic Computation*. John Wiley & Sons, New York, 1998.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[vB84] J. van Benthem. Correspondence theory. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, pages 167–247. Kluwer, 1984.

[WS92] R.J. Waldinger and M.E. Stickel. Proving properties of rule-based systems. *Intl. J. Software Engineering and Knowledge Engineering*, 2(1):121–144, 1992.

[ZBH96] H. Zhang, M.P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[Zha00] H. Zhang. Lemma generation in the Davis-Putnam method. In *Proc. CADE-17 Workshop on Model Computation*, June 2000.

[ZMMM01] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Intl. Conf. on Computer-Aided Design (ICCAD 2001)*, pages 279–285, San Jose, CA, November 2001. ACM Press.

Veröffentlichungen

Zeitschriftenpublikationen

[1] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

Abstract. We address the parallelization and distributed execution of an algorithm from the area of symbolic computation: propositional satisfiability (SAT) checking with dynamic learning. Our parallel programming models are strict multithreading for the core SAT checking procedure, complemented by mobile agents realizing a distributed dynamic learning process. Individual threads treat dynamically created subproblems, while mobile agents collect and distribute pertinent knowledge obtained during the learning process. The parallel algorithm runs on top of our parallel system platform DOTS (Distributed Object-Oriented Threads System), which provides support for our parallel programming models in highly heterogeneous distributed systems. We present performance measurements evaluating the performance gains by our approach in different application domains with practical significance.

[2] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, January 2003. Special issue on configuration.

Abstract. Compilation and maintenance of correct product data is a sophisticated and demanding task. We show how application of formal methods can support this task by enabling automatic checks of standardized consistency properties. The consistency maintenance tool BIS, an extension to the product data management (PDM) system used at DaimlerChrysler AG to configure the Mercedes lines of passenger cars and commercial vehicles, realizes this approach. The PDM system maintains a data base of sales options and parts together with a set of Boolean constraints expressing valid configurations and their transformation into manufacturable products. BIS plugs into this constraints set and uses a propositional logic satisfiability (SAT) checker to detect inconsistencies. BIS also supports manufacturing decisions by calculating implications of product or production changes on the set of required parts. We give a comprehensive account of BIS: the formalization of the business processes underlying its construction, the modifications of SAT-checking technology we found necessary in this context, and the software technology used to package the product as a client / server information system.

[3] Carsten Sinz, Thomas Lumpp, Jürgen Schneider, and Wolfgang Küchlin. Detection of dynamic execution errors in IBM System Automation's rule-based expert system. *Information and Software Technology*, 44(14):857–873, November 2002.

Abstract. We formally verify aspects of the rule-based expert system of IBM's System Automation software for IBM's zSeries mainframes. Starting with a formalization of the expert system in Propositional Dynamic Logic (PDL), we encode termination and determinism properties in PDL and its extension ΔPDL. We then translate our decision problems to propositional logic and apply advanced SAT techniques for automated proofs. In order to locate real program bugs for each failed proof attempt, we apply extra formalization steps and represent propositional error formulae in concise normal form as Binary Decision Diagrams (BDDs). In our experiments, we revealed residual non-termination bugs in a tested program version close to shipment, and, after correcting them, we formally verified the absence of this class of bugs in the production code.

[4] Wolfgang Küchlin and Carsten Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, February 2000.

Abstract. We present a formal specification and verification approach for industrial product data bases containing boolean logic formulae to express constraints. Within this framework, global consistency assertions about the product data are converted into propositional satisfiability problems. Today's state-of-the-art provers turn out to be surprisingly efficient in solving the SAT-instances generated by this process. Moreover, we introduce a method for encoding special non-monotonic constructs in traditional boolean logic. We have successfully applied our method to industrial automotive product data management and could establish a set of commercially used interactive tools that facilitate the management of change and help raise quality standards.

Konferenzbeiträge

[1] Carsten Sinz, Amir Khosravizadeh, Wolfgang Küchlin, and Viktor Mihajlovski. Verifying CIM models of Apache web server configurations. In *Proc. of the 3rd International Conference on Quality Software (QSIC 2003)*, pages 290–297, Dallas, TX, November 2003. IEEE Computer Society.

Abstract. We show how configuration properties of the Apache Web-server can be formally verified, so that an installation is safe with respect to both universal and site specific local constraints. Our approach starts from an existing semi-formal component model of the Web-server in the Common Information Model (CIM) standard. Hence our approach is applicable also to the verification of other systems for which a CIM model exists.

[2] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. A universal parallel SAT checking kernel. In Hamid R. Arabnia and Youngsong Mun, editors, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, volume 4, pages 1720–1725, Las Vegas, NV, June 2003. CSREA Press.

Abstract. We present a novel approach to parallel Boolean satisfiability (SAT) checking. A distinctive feature of our parallel SAT checker is that it incorporates all essential heuristics employed by the state-of-the-art sequential SAT checking algorithm. This property makes our parallel SAT checker applicable in a wide range of different application domains. For its distributed execution a combination of the strict multithreading and the mobile agent programming model is employed. We give results of run-time measurements for problem instances taken from different application domains, indicating the usefulness of the presented method.

[3] Jörg Denzinger, Carsten Sinz, Jürgen Avenhaus, and Wolfgang Küchlin. Teamwork-PaReDuX: Knowledge-based search with multiple parallel agents. In *Proceedings of the International Conference on Massively Parallel Computing Systems (MPCS 2002)*, Ischia, Italy, April 2002. National Technological University Press, Fort Collins, CO.

Abstract. We present the combination of a distribution approach and a parallelization concept for knowledge-based search. We formally characterize distribution and parallelization and present one instantiation of each, Teamwork and PaReDuX. Teamwork-PaReDuX, the combination of them, employs collaborating parallel search agents to prove equational theorems. Our experiments indicate that the speed-ups obtained by the single approaches are multiplied when using the combination, thus making good use of networks of multi-processor computers and allowing us to solve harder problems in acceptable time.

[4] Carsten Sinz, Thomas Lumpp, and Wolfgang Küchlin. Towards a verification of the rule-based expert system of the IBM SA for OS/390 automation manager. In *Proceedings of the 2nd Asia-Pacific Conference on Quality Software (APAQS 2001)*, pages 367–374, Hong Kong, December 2001. IEEE Computer Society.

Abstract. We formally verify consistency aspects of the rule-based expert system of IBM's System Automation software for IBM's e-server zSeries. Starting with a formalization of the expert system in propositional dynamic logic (PDL), we are able to encode termination and determinism properties. To circumvent direct proofs in PDL or its extension ΔPDL, we further translate versions of the occurring decision problems to propositional logic, where we can apply advanced SAT and BDD techniques. In our experiments we could reveal some inconsistencies and, after correcting them, we successfully verified a non-looping property for a part of the expert system.

[5] Carsten Sinz, Jörg Denzinger, Jürgen Avenhaus, and Wolfgang Küchlin. Combining parallel and distributed search in automated equational deduction. In *Parallel Processing and Applied Mathematics PPAM 2001*, number 2328 in LNCS, Naleczow, Poland, September 2001. Springer-Verlag.

Abstract. We present an automated deduction system for equational reasoning combining two different parallelization/distribution schemes: Strategy-compliant parallelization on the level of individual deduction steps (PaReDuX) and distributed cooperation of multiple agents with different search strategies (TEAMWORK). In our experiments we mainly observed a multiplication of the speed-ups of each approach in our combined system.

[6] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel consistency checking of automotive product data. In *Proc. of Intl. Conference Parallel Computing (ParCo 2001)*, Naples, Italy, September 2001.

Abstract. This paper deals with a parallel approach to the verification of consistency aspects of an industrial product configuration data base. The data base we analyze is used by DaimlerChrysler to check the orders for cars and commercial vehicles of their Mercedes lines. By formalizing the ordering process and employing techniques from symbolic computation we could establish a set of tools that allow the automatic execution of huge series of consistency checks, thereby ultimately enhancing the quality of the product data. However, occasional occurrences of computation intensive checks are a limiting factor for the usability of the tools. Therefore, a prototypical parallel re-implementation using our Distributed Object-Oriented Threads System (DOTS) was carried out. Performance measurements on a heterogeneous cluster of shared-memory multiprocessor Unix workstations and standard Windows PCs revealed considerable speed-ups and substantially reduced the average waiting time for individual checks. We thus arrive at a noticeable improvement in usability of the consistency checking tools.

[7] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Distributed parallel SAT checking with dynamic learning using DOTS. In T. Gonzales, editor, *Proc. of the IASTED Intl. Conference Parallel and Distributed Computing and Systems (PDCS 2001)*, pages 396–401, Anaheim, CA, August 2001. ACTA Press.

Abstract. We present a novel method for distributed parallel automatic theorem proving. Our approach uses a dynamically learning parallel SAT checker incorporating distributed multi-threading and mobile agents. Individual threads process dynamically created subproblems, while agents collect and distribute new knowledge created by the learning process. As parallelization platform we use the Distributed Object-Oriented Threads System (DOTS) that provides support for both distributed threads and mobile agents. We present experiments indicating the usefulness of the presented approach for different application domains.

[8] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Detection of inconsistencies in complex product model data using extended propositional SAT-checking. In I. Russell and J. Kolen, editors, *Proceedings of the 14th International FLAIRS Conference*, pages 645–649, Key West, FL, May 2001. AAAI Press.

Abstract. We present our consistency support tool BIS, an extension to the electronic product data management system (EPDMS) used at DaimlerChrysler AG to configure the Mercedes lines of passenger cars and commercial vehicles. BIS allows verification of certain integrity aspects of the product data as a whole. The underlying EPDMS maintains a data base of sales options and parts together with a set of logical constraints expressing valid configurations and their transformation into manufacturable products. Due to the complexity of the products and the induced complexity of the constraints, maintenance of the data base is a nontrivial task and error-prone. By formalizing DaimlerChrysler's order processing method and converting global consistency assertions about the product data base into formulae of an extended propositional logic, we are able to employ a satisfiability checker integrated into BIS to detect inconsistencies, and thus increase the quality of the product data.

[9] Carsten Sinz. System description: ARA - an automatic theorem prover for relation algebras. In D. McAllester, editor, *Automated Deduction CADE-17*, number 1831 in LNAI, pages 177–182, Pittsburgh, PA, June 2000. Springer-Verlag.

Abstract. ARA is an automatic theorem prover for various kinds of relation algebras. It is based on Gordeev's Reduction Predicate Calculi for n-variable logic (RPC_n) which allow first-order finite variable proofs. Employing results from Tarski/Givant and Maddux we can prove validity in the theories of simple semi-associative relation algebras, relation algebras and representable relation algebras using the calculi RPC_3, RPC_4 and RPC_ω. ARA, our implementation in Haskell, offers different reduction strategies for RPC_n, and a set of simplifications preserving n-variable provability.

[10] Ralf-Dieter Schimkat, Wolfgang Blochinger, Carsten Sinz, Michael Friedrich, and Wolfgang Küchlin. A service-based agent framework for distributed symbolic computation. In M. Bubak, R. Williams, H. Afsarmanesh, and B. Hertzberger, editors, *Proc. 8th Intl. Conf. on High Performance Computing and Networking Europe, HPCN 2000*, number 1823 in LNCS, pages 644–656, Amsterdam, Netherlands, May 2000. Springer-Verlag.

Abstract. We present Okeanos, a distributed service-based agent framework implemented in Java, in which agents can act autonomously and make use of stationary services. Each agent's behaviour can be controlled individually by a rule-based knowledge component, and cooperation between agents is supported through the exchange of messages at common meeting points (*agent lounges*). We suggest this general scheme as a new parallelization paradigm for Symbolic Computation, and demonstrate its applicability by an agent-based parallel implementation of a satisfiability (SAT) checker.

[11] Reinhard Bündgen, Carsten Sinz, and Jochen Walter. ReDuX 1.5: New facets of rewriting. In Harald Ganzinger, editor, *Rewriting Techniques and Application RTA-96*, number 1103 in LNCS, pages 412–415, New Brunswick, NJ, July 1996. Springer-Verlag.

Abstract. The ReDuX system is a term rewriting laboratory that allows the user to experiment with completion procedures. It features Knuth-Bendix completion with critical pair criteria, Peterson-Stickel completion for commutative and/or associative-commutative theories, inductive completion procedures based on positional ground reducibility tests, tools to analyze the set of irreducible ground terms, an unfailing completion procedure base on ordered rewriting and a random term generator.

Begutachtete Workshop-Beiträge

[1] Carsten Sinz. Knowledge compilation for product configuration. In *Configuration Workshop Proceedings, 15th European Conference on Artificial Intelligence (ECAI-2002)*, pages 23–26, Lyon, France, July 2002.

Abstract. In this paper we address the application of knowledge compilation techniques to product configuration problems. We show that both the process of generating valid configurations, as well as validation of product configuration knowledge bases, can potentially be accelerated by compiling the instance independent part of the knowledge base. Besides giving transformations of both tasks into logical entailment problems, we give a short summary on knowledge compilation techniques, and present a new algorithm for computing unit-resolution complete knowledge bases.

[2] Carsten Sinz. Formal verification in an industrial context. In *Symposium on the Effectiveness of Logic in Computer Science (ELICS'02)*, pages 59–63, Saarbrücken, Germany, March 2002. In Technical Report MPI-I-2002-2-007.

Abstract. We present two case studies employing formal verification in an industrial context. Our first example deals with product configuration for the automotive industry, the second one examines a rule-based expert system controlling IBM's high-availability System Automation software. We identify common requirements to both the logical encoding and the decision procedures for the purpose of verification. Moreover we summarize experiences gained during these projects.

[3] Carsten Sinz and Wolfgang Küchlin. Dealing with temporal change in product documentation for manufacturing. In *Configuration Workshop Proceedings, 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 71–77, Seattle, WA, August 2001.

Abstract. Product documentation for manufacturing is subject to temporal change: Exchange of parts by successor models, shift from in-house production to external procurement, or assembly line reconfiguration are just a few examples. In this paper we show how DaimlerChrysler AG manages configuration for manufacturing of their Mercedes lines. Furthermore, we identify typical situations of change, their representation on the product documentation level, and how to keep the documentation consistent over time. We then develop two verification methods for computing the induced change at the parts level. Finally, we show how our methods can be applied to handle model year change and production relocation.

[4] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and ap-

plications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

Abstract. We present PaSAT, a parallel implementation of a Davis-Putnam-style propositional satisfiability checker incorporating dynamic search space partitioning, intelligent backjumping, as well as lemma generation and exchange; the main focus of our implementation is on speeding up SAT-checking of propositional encodings of real-world combinatorial problems. We investigate and analyze the speed-ups obtained by parallelization in conjunction with lemma exchange and describe the effects we observed during our experiments. Finally, we present performance measurements from the application of our prover in the areas of formal consistency checking of industrial product documentation, cryptanalysis, and hardware verification.

[5] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. SAT-based consistency checking of automotive electronic product data. In *Configuration Workshop Proceedings, 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 74–78, Berlin, Germany, August 2000.

Abstract. Complex products such as motor vehicles or computers need to be configured as part of the sales process [McD82]. If the sale is electronic, then the configuration and some validity checking of the order must be done electronically as part of an electronic product data management system (EPDMS). The EPDMS typically maintains a data base of sales options and parts together with a set of logical constraints expressing valid combinations of sales options and their transformation into manufacturable products. Due to the complexity of these constraints, creation and maintenance of the configuration data base is a nontrivial task and error-prone. We present our system BIS which is commercially used to check global consistency assertions about the product data base used by the EPDMS of a major car and truck manufacturer. The EPDMS uses Boolean logic to encode the constraints, and BIS translates the consistency assertions into problems which it solves using a propositional satisfiability checker. We expect our approach to be especially suited for rapidly changing complex products as they increasingly appear in electronic commerce.

Sonstiges

[1] Andrea Sinz and Carsten Sinz. Software development for determining cross-linking sites in proteins. Poster, 35. Diskussionstagung der

Deutschen Gesellschaft für Massenspektrometrie (DGMS 2002), Heidelberg, Germany, March 2002.

Abstract. Chemical cross-linking, a conceptually simple approach, has gained renewed interest in combination with mass spectrometric analysis of the reaction products by its possibilities to elucidate threedimensional structure of proteins as well as interfaces among interacting proteins in their natural environment. The identification of a large number of cross-linking sites from the complicated mixtures generated by chemical cross-linking however remains a daunting task. To facilitate determination and classification of cross-linking sites, we developed a novel software program named Calc-XL.

[2] Wolfgang Küchlin and Carsten Sinz. Proving consistency assertions for automotive product data management. In I. Gent, H. van Maaren, and T. Walsh, editors, SAT2000 – Highlights of Satisfiability Research in the Year 2000, volume 63 of *Frontiers in Artificial Intelligence and Applications*. IOS Press 2000. (Also appeared in *J. Automated Reasoning*.)

Lebenslauf

Name	Carsten Sinz
Geburtsdatum	22.07.1970
Geburtsort	Albstadt-Ebingen
Familienstand	ledig
Staatsangehörigkeit	deutsch
Konfession	evangelisch

Schulbildung

08/77-07/81	Oststadt-Grundschule Ebingen
08/81-05/90	Gymnasium Ebingen, Abschluss Abitur

Zivildienst

06/90-09/91	Fachklinik Zollern-Alb

Studium

10/91-07/98	Studium der Informatik mit Nebenfach Physik an der Universität Tübingen

Berufliche Tätigkeit

08/98-07/99	Projektmitarbeiter am Steinbeis-Transferzentrum Objekt- und Internettechnologien (OIT)
08/99-	Wissenschaftlicher Mitarbeiter am Arbeitsbereich Symbolisches Rechnen des Wilhelm-Schickard-Instituts für Informatik der Universität Tübingen