Vereinfachte Integration von Fachwissen in Computerprogramme am Beispiel eines Planungssystems für die Bauindustrie

Jochen Weiß

Kurzfassung

Ein neues Konzept für eine vereinfachte Integration von Fachwissen in Computerprogramme wird vorgestellt. Ergebnis ist ein Planungssystem zur Unterstützung von Planungsprozessen. Das System ermöglicht die Abbildung von Produktzusammensetzungen einschliesslich ihrer Abhängigkeiten untereinander. Die Produktzusammensetzung kann durch Gültigkeitsregeln vervollständigt werden. Damit ist eine computerunterstützte Planung mit der Möglichkeit programmautomatischer Überprüfung der Gültigkeitsregeln möglich. Weiterhin ermöglicht das System die Verwaltung zahlreicher Produktfamilien und Produktversionen. Damit wird zum einen berücksichtigt, dass unterschiedliche Produktfamilien zur Planung zur Verfügung stehen und dass zahlreiche benutzer- beziehungsweise auftragsspezifische Konfigurationen geplant und verwaltet werden müssen. Zum anderen wird berücksichtigt, dass Produktkomponenten ständig weiterentwickelt werden und somit in neuen Versionen zur Verfügung stehen.

Die praktische Anwendbarkeit wird durch die Orientierung bei der Eingabe der Produktzusammensetzung an bekannten Strukturen erreicht.

Da die Regeln der Produktzusammensetzung von den Experten des Anwendungsgebiets in das System eingegeben werden, entfällt eine aufwändige Weitergabe dieses umfangreichen Fachwissens an die Experten der Softwareentwicklung. Fachspezifische Anwenderprogramme können auf der Grundlage dieses Systems schneller erstellt werden.

Abstract

A new concept for a simplified integration of knowledge into computer programs is presented. The result is a design system for the support of planning processes. The system makes it possible to map product compositions including their validity rules. With this, computer supported design with the possibility of an automated check of all validity rules is possible. Furthermore the system is capable of managing numerous product families and product versions. Thus, on the one hand it is taken into consideration that different product families are available for design and that numerous user- or order-specific configurations are to be planned and managed. On the other hand it is taken into account that product components are evolving and new versions are available.

Practical applicability is achieved through an orientation along known structures for the definition of the product composition.

A complex transmission of the extensive knowledge of product composition to the experts on software development is not necessary because the rules of product composition are entered into the system by the domain specific experts. Domain specific software can be developed faster using this system.

Die vorliegende Schrift ist eine von der Fakultät für Bauingenieur- und Vermessungswesen der Universität der Bundeswehr München genehmigte Dissertation.

Thema der Dissertation: Vereinfachte Integration von Fachwissen in Computerprogramme am Beispiel eines Planungssystems für die Bauindustrie

Verfasser: Herr Dipl.-Ing. Jochen Weiß

Promotionsausschuss:

Vorsitzender: Univ.-Prof. Dipl.-Ing. Franz Remmer

1. Berichterstatter: Univ.-Prof. Dr.-Ing. Stefan Holzer

2. Berichterstatter: Univ.-Prof. Dr. rer.nat. Ernst Rank

Tag der Prüfung: 06. Mai 2004

Mit der Promotion erlangter akademischer Grad: Doktor der Ingenieurwissenschaften (Dr.-Ing.)

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Stand der Technik

An der Planung, der Errichtung und dem Betrieb eines Bauwerks sind viele Experten unterschiedlicher Fachdisziplinen beteiligt. Weiterhin werden viele Bauprojekte in unternehmensübergreifenden Kooperationen durchgeführt. Die Zusammensetzung dieser Kooperationen wechselt ständig und die einzelnen Planerteams müssen ihre Projekte räumlich und zeitlich verteilt abwickeln [47].

Bisher sind die Softwarewerkzeuge zur Projektbearbeitung im Bauwesen auf die jeweilige Fachdisziplin, ihre Terminologie und ihre Sichtweise auf die Informationen angepasst [4]. Sie bilden voneinander unabhängige Insellösungen.

Erster Schritt zu einer Effizienzsteigerung war der Austausch von geometrischen Basisdaten. Da hierbei aber keinerlei weitergehenden Informationen zu den dargestellten Objekten weitergegeben wurden, mussten diese Informationen von jedem Bearbeiter neu erzeugt beziehungsweise rekonstruiert werden [49]. Darauf folgten Arbeiten an der Entwicklung von Produktmodellen, mit denen alle relevanten Daten eines Projektes abgebildet und übertragen werden können. Hierfür wurde Mitte der achtziger Jahre die Arbeitsgruppe ISO TC 184/SC gebildet. Die von ihnen erarbeitete Schnittstelle STEP soll alle im Produktlebenszyklus entstehenden Daten enthalten [28]. Haller hat in [28] auf der Basis von STEP ein Produktmodell für den Stahlbau entwickelt. Die Industry Foundation Classes bauen auf den Entwicklungen von STEP auf [4, 49]. Aktuelle Arbeiten im Bereich des Stahlbaus haben die Integration der Produktschnittstelle Stahlbau in die Industry Foundation Classes zum Ziel [31].

Die Industry Foundation Classes haben das Ziel, die für ein Bauprojekt relevanten Daten in einem objektorientierten Modell zur Verfügung zu stellen. Methoden, die das Verhalten der Klassen beschreiben, sind noch nicht Bestandteil der Spezifikationen. Werden von Projektbearbeitern verschiedener Fachdisziplinen räumlich und zeitlich getrennt Planungsschritte durchgeführt, so existieren keine Konzepte, wie programmautomatisch die Konsistenz der Daten überprüft und gegebenenfalls

wieder hergestellt werden kann. Weiterhin ist noch kein Konzept vorhanden, wie ein beliebiger Planungszustand zu einem späteren Zeitpunkt rekonstruiert werden kann.

In [49] wird explizit auf die Notwendigkeit integrierter Entwurfsdaten hingewiesen. Dabei sollen diese Daten die Möglichkeit eröffnen, den gesamten Entwurfsprozess, die Bauphasen und alle später eingeflossenen Änderungen nachvollziehbar zu machen. Hierfür ist es notwendig, die Daten mit Versionsinformationen zu versehen. Bittrich sagt in [4] aus, dass in der Version 2.0 der Industry Foundation Classes unter anderem das Konzept der Version geplant sei. Die Umsetzung wurde in der Version 2.0 nicht vorgenommen.

Zusammenfassend haben die Arbeiten an STEP und den Industry Foundation Classes das Ziel, die relevanten Daten zu strukturieren und so ein gemeinsames Projektmodell zu entwickeln.

Einen weiteren Beitrag zur Effizienzsteigerung leistet die internetbasierte Telekooperationsplattform von Müller in [47]. Müller berücksichtigt hierbei auch Aspekte einer verteilten Teamarbeit. Er geht davon aus, dass auch in naher Zukunft der überwiegende Teil der Informationen weiterhin schwach strukturiert in Form von Dokumenten vorliegen. Daher dient sein Informationsmanagementsystem dem Management von Dokumenten.

Die Arbeit von Müller hat das Ziel, durch Bereitstellung einer Kooperationsplattform für schwach- und unstrukturierte Daten die Abwicklung von Bauprojekten zu unterstützen. Dabei werden Methoden zur Beherrschung der Komplexität der Projektdurchführung erarbeitet. Ein Produktdatenmodell zur Beschreibung der Komponenten eines Bauwerks stellt Müller nicht zur Verfügung.

Die verteilte Bearbeitung einer strukturierten Menge von Objektversionen untersucht Firmenich in [21]. Er berücksichtigt hierbei die interaktive Bearbeitung von Teilmengen einer Planungsaufgabe in CAD-Programmen. Das System stellt sicher, dass ein Planungsschritt nicht auf überholten Objektversionen beruht.

Die Planung eines Bauwerks entspricht der Entwicklung eines komplexen Systems. Wichtige Aspekte bei der Durchführung von Planungsprojekten sind die Nachvollziehbarkeit aller Planungsschritte und Änderungen und die Konsistenz der Daten.

Für die Entwicklung und die Kontrolle von komplexen Systemen bietet die Informatik die Methoden des Konfigurationsmanagements an. Dort wird unterschieden zwischen SCM[1] und PDM[2]. In [19] und [59] werden die Gemeinsamkeiten und Unterschiede der beiden Bereiche dargestellt.

Eine Definition gibt Estublier in [20]. Estublier definiert SCM als die Kontrolle des Entwicklungsprozesses eines komplexen Systems und legt weiter dar, dass dies auch auf andere Ingenieurdisziplinen zutrifft. Er nennt Softwarewerkzeuge zum Konfigurationsmanagement in Ingenieurdisziplinen wie beispielsweise Elektrotechnik oder Maschinenbau PDM-Werkzeuge.

[1] Software Configuration Management
[2] Product Data Management

Die wesentlichen Funktionen eines Konfigurationsmanagementsystems sind nach [44] die Unterstützung der

- Versionierung,
- Zusammensetzung und
- Erzeugung

aller relevanten Bestandteile einer Konfiguration. Weiterhin beinhaltet das Konfigurationsmanagement die Unterstützung und die Kontrolle der damit zusammenhängenden Aktivitäten eines Teams von Bearbeitern. Konfigurationsmanagement ist somit ein zentraler Bestandteil jedes umfangreichen Ingenieurprojekts.

Hedin, Ohlsson und McKenna stellen in [29] ein Modell zur Produktkonfiguration vor. Dieses Modell erlaubt die Definition von Regeln, anhand derer eine Konsistenzprüfung möglich ist. Die in [49] geforderte Nachvollziehbarkeit wird nicht unterstützt, da keine Möglichkeit der Versionierung vorhanden ist.

Lin stellt in [39] ein Software-Konfigurationsmanagementsystem vor, das nicht wie viele andere derartige Systeme auf der Basis von Dateien und Verzeichnissen arbeitet, sondern auf Basis von Objekten. Dieses System erlaubt Versionierung, benutzerspezifische Arbeitsumbegungen, Modellierung von Produkten durch Graphen und die Identifikation von Konfigurationen. Nicht vorhanden ist die Möglichkeit der Definition von Regeln zur Konsistenzprüfung.

1.2 Ziel der Arbeit

In der vorliegenden Arbeit wird überprüft, wie die Methoden des Konfigurationsmanagements in der Durchführung von Bauplanungsprojekten eingesetzt werden können. Dies erfolgt wie in [47] unter Berücksichtigung von räumlich und zeitlich verteilten Planungsteams.

In dieser Arbeit wird ein Planungssystem entwickelt, das mit [29] die Möglichkeit der

- auftragsspezifischen Produktkonfiguration,
- Definition von Regeln und
- programmautomatischen Konsistenzprüfung

gemein hat.

Weiterhin hat dieses System mit [39] die

- Möglichkeit der Versionierung,

- Bereitstellung von benutzerspezifischen Arbeitsumgebungen und
- Verwaltung von Konfigurationen

gemein.

Kapitel 2 gibt einen Überblick über die Nutzung von DV-Systemen in Unternehmen. Hierbei wird die Entwicklung der Computerunterstützung aufgezeigt und die Randbedingungen dargelegt, die die Verwendung von DV-Systemen im Unternehmen kennzeichnen. Anschliessend werden bisher entwickelte Lösungen zum Datenaustausch oder zur gemeinsamen Nutzung von Daten während der Planung von Bauprojekten vorgestellt.

Kapitel 3 stellt wesentliche Grundlagen des Konfigurationsmanagements vor und erläutert Gemeinsamkeiten und Unterschiede der beiden Bereiche Software Configuration Management und Product Data Management.

Das in dieser Arbeit entwickelte System übernimmt Bestandteile des SCM-Systems von Lin [39] und des Produktkonfigurators von Hedin, Ohlsson und McKenna [29]. Die wesentlichen Eigenschaften dieser Systeme werden in **Kapitel 4** und **Kapitel 5** dargelegt.

Das Konzept für ein neues System erläutert **Kapitel 6**.

Im **Kapitel 7** wird zusammengestellt, welche Eigenschaften zur Versionierung das neue System haben soll und wie diese in der vorliegenden Arbeit realisiert werden.

Die notwendigen Eigenschaften zur Produktkonfiguration und die Umsetzung werden in **Kapitel 8** dargestellt.

Kapitel 9 zeigt eine mögliche Abbildung des Systems auf relationale Datenbanken.

Die Anwendung des Systems zeigen abschliessend Beipiele in **Kapitel 10** und **Kapitel 11**.

Kapitel 2

DV-Einsatz in den Unternehmen

2.1 Allgemeines

Der DV-Einsatz in den Unternehmen ist geprägt vom Einsatz der Personal-Computer. Die Entwicklung des Personal-Computer zum Standard-Arbeitsgerät geht einher mit der Entwicklung leistungsfähiger Programme zur Datenverarbeitung. Die Verfügbarkeit von günstigen Einzelplatzrechnern mit herstellerübergreifenden Standards für Hardware und Betriebssysteme hat der elektronischen Datenverarbeitung in den Unternehmen zu einer hohen Durchdringung verholfen. Die Probleme im Zusammenhang mit der Nutzung heutiger DV-Systeme sind nach [53] Ergebnis von organisatorischen und konzeptionellen Unzulänglichkeiten.

Zum einen wird bei der Anschaffung von Arbeitsgeräten zumeist abteilungsbezogen gehandelt und die Auswirkungen auf andere Bereiche werden nicht berücksichtigt [1]. Dies führt zu Insellösungen.

Zum anderen existieren in alteingesessenen Unternehmen oft historisch gewachsene Arbeitsabläufe, die in Zeiten ohne DV-Systeme festgelegt wurden [53].

Die folgenden Randbedingungen kennzeichnen die Verwendung von DV-Systemen in den Unternehmen.

Zunahme der Datenmenge: Die hohe Durchdringung der Unternehmen mit DV-Systemen und die Client-Server-Technologie führten in den Unternehmen dazu, dass nahezu alle Arbeitschritte im Rahmen der Produktentwicklung, der Produktion und der Auftragsabwicklung unter Einsatz von DV-Systemen durchgeführt werden. Dies führt dazu, dass die meisten relevanten Daten in digitaler Form vorliegen und die Datenmenge stetig wächst [53]. Werden hierbei in den unterschiedlichen Funktionsbereichen unterschiedliche DV-Systeme eingesetzt, führt dies entlang des Produktentstehungsprozesses zu grossen Datenmengen, die redundante Daten beinhalten.

[1] Ebenso wie Produktionsunternehmen in Funktionsbereiche aufgeteilt sind, fand bei der Entwicklung oder dem Kauf von DV-Programmen eine Orientierung an diesen Funktionseinheiten statt.

Mangelnde Informationstransparenz: Im Verlauf einer Produktentwicklung entstehen durch die Anwendung unterschiedlicher Programme eine Vielzahl von Dokumenten. Fehlt ein Werkzeug zur Verwaltung und Verteilung aller erzeugten Daten, sind Suchvorgänge nach Informationen sehr arbeits- und zeitintensiv. Damit ist der Datenbestand nur eingeschränkt nutzbar. Nach [53] besteht das Problem darin, dass sich die Historie eines Dokuments nicht oder nur schwer nachvollziehen lässt.

Mangelnde Systemintegration: Die Verwendung unterschiedlicher Programme führt zu Problemen, sobald Daten zwischen den eingesetzten Programmen ausgetauscht werden müssen.

Beispielsweise legt Haller in [28] dar, dass die DV-Programme der Bereiche Statik, Konstruktion und Fertigung im Stahlbau weitgehend unabhängig voneinander arbeiten. Es existieren zwar auch durchgängige Lösungen für die Projektbearbeitung, da diese aber umfangreiche Programmsysteme jeweils eines Herstellers sind, ist man in der Auswahl der Programmodule eingeschränkt.

Diese Situation kann man auf den unternehmensweiten Einsatz von Programmsystemen ausdehnen. Auch hier gibt es umfangreiche Programmsysteme jeweils eines Herstellers. Werden einzelne Programmodule verschiedener Hersteller eingesetzt, führen diese Insellösungen dazu, dass Daten mehrfach eingegeben werden müssen. Dies führt zu einem Mehraufwand und bei Eingabefehlern zu Inkonsistenzen in den Daten.

2.2 DV-Nutzung bei Bauplanungsprojekten

2.2.1 Situation

Nach Müller [47] ist die Bauplanung durch folgende Tatsachen gekennzeichnet.

- Bauwerke sind Null-Serien beziehungsweise Unikatprodukte.
- An der Planung und der Erstellung von Bauwerken sind zahlreiche Planer, Ausführende und Lieferanten aus unterschiedlichen Fachdisziplinen beteiligt.
- Kaum ein Unternehmen kann alle geforderten Leistungen selbst erbringen. Die Bauvorhaben werden in unternehmensübergreifenden Kooperationen ausgeführt.
- Die Zusammensetzung der Kooperationen wechselt ständig, für jedes Projekt werden die projektbeteiligten Unternehmen neu ausgewählt.
- Anforderungen und Zielvorgaben können zu Projektbeginn nicht vollständig formuliert werden. So existieren beispielsweise zu Projektbeginn explizite Vorstellungen bezüglich der Termine und der Kosten, gestalterische Qualitäten des Bauwerks sind noch unscharf formuliert. Diese werden im Projektverlauf iterativ festgelegt.

Kretz weist in [38] darauf hin, dass grössere Bauvorhaben zumeist arbeitsteilig von mehreren Planern abgewickelt werden. Die Gründe hierfür sind

- die Komplexität der zu lösenden Aufgaben,
- die Spezialisierung der Planer und
- der zunehmende Konkurrenzdruck, der eine schnelle Abwicklung fordert.

Er beschreibt, dass bei der Projektabwicklung während der gesamten Planungs- und Bauzeit mit Änderungen zu rechnen sind. Die Änderungen sind möglichst zeitnah umzusetzen und sind ein natürlicher Teil des Lösungsfindungsprozesses.

In dieser Situation müssen Daten ständig über DV-Systemgrenzen und gegebenenfalls auch länderübergreifend ausgetauscht beziehungsweise geteilt werden.

Für eine effiziente Durchführung des Projekts ist es daher notwendig, die in einem Planungsschritt erzeugten Daten allen anschliessenden Planungsprozessen zur Verfügung zu stellen. Eine erneute Eingabe bereits in einem anderen System vorhandener Daten soll vermieden werden.

Nach Brown et al. führt der Kostendruck, gekoppelt mit einer immer stärkeren Spezialisierung der Unternehmen und der zunehmenden Komplexität der Projekte, zur Forderung einer Integration der Projektinformationen [8]. Die Entwicklung neuer Softwareprogramme für die Bauindustrie, Forschritte in der Netzwerktechnologie, neue Modellierungsmethoden und die Definition von Standards für den Datenaustausch bringen neue Möglichkeiten für eine Integration.

Brown et al. führen in [8] weiter aus, dass zur Abwicklung von Bauprojekten oft virtuelle Teams zusammengestellt werden. Diese Teams bestehen nur für die Dauer des Projekts, nach Abschluss der Arbeiten werden die Teams aufgelöst.

Auch Gupta und Tiwari stellen in [27] dar, dass bei der Planung von Bauvorhaben viele Experten unterschiedlicher Fachrichtungen kooperieren müssen. Dabei hat jeder Spezialist eine seiner Fachrichtung entsprechende Sicht auf die Entwurfsdaten. Sie weisen darauf hin, dass Inkonsistenzen im Entwurf zu kostenintensiven Änderungen und zu Überschreitungen der Kosten- und Zeitziele führen.

2.2.2 Schnittstellen und Produktdatenmodelle

Der erste Ansatz zur Wiederverwendung bereits in einem System eingegebener oder generierter Daten ist der Datenaustausch über eine individuell für diesen Einsatzbereich konzipierte Schnittstelle. Die dabei auftretenden Probleme sind von Haller in [28] dargelegt.

- Die Programmierung dieser Schnittstellen ist wegen des Abstimmungsaufwands zeitaufwändig und teuer.
- Bei n eingesetzten DV-Programmen sind $n^2 - n$ Einzelschnittstellen notwendig.

- Das Interesse für Schnittstellen zwischen gleichartigen Programmen seitens der Softwarehersteller ist gering.

Zur Reduzierung der Anzahl an Einzelschnittstellen wurden in der Vergangenheit herstellerunabhängige Schnittstellen definiert. Soll von einem Softwarehersteller eine Schnittstelle unterstützt werden, dann ist von ihm lediglich ein Modul zum Datenimport und ein Modul zum Datenexport zu entwickeln. Stehen diese Funktionen zur Verfügung, kann das Programm mit allen weiteren Programmen Daten austauschen, die ebenfalls diese Schnittstellendefinition unterstützen.

Erste Schnittstellendefinitionen dienten lediglich zum Austausch von Geometriedaten [6]. Hierzu zählen die Austauschformate DXF und IGES [35]. Bittrich beschreibt in [4] den Ablauf eines typischen Planungsprojekts. Bei einem Austausch von Geometriedaten erhält ein Bearbeiter CAD-Zeichnungen eines anderen Projektbeteiligten. Er entfernt gegebenenfalls für ihn nicht relevante Informationen und fügt im Verlauf der Bearbeitung seiner Zeichnung neue Daten hinzu. Dadurch entstehen viele Partialmodelle, deren Informationen häufig redundant oder sogar inkonsistent sind. Bei jeder Transformation in ein anderes Partialmodell kommt es zu Datenverlusten.

Ein weiterer Schritt war die Entwicklung von Produktdatenmodellen, die das gesamte physikalische Bauwerksmodell vollständig und redundanzfrei abbilden sollen. Grundlage für die Definition einer Schnittstelle zur herstellerübergreifenden Verwendung ist ein Produktmodell, das alle relevanten Daten beinhaltet. Beispiele hierfür sind die Produktschnittstelle Stahlbau [17, 28] und die Industry Foundation Classes [4].

STEP und die Produktschnittstelle Stahlbau

Haller legt in [28] dar, dass vom ISO[2] Mitte der achtziger Jahre die Arbeitsgruppe ISO TC 184/SC gebildet wurde. Das Ziel war die Definition eines genormten Übertragungsformats, in dem alle ein Produkt beschreibenden Daten enthalten sein können. Diese Schnittstelle heisst STEP[3]. STEP ist nach [19] durch zwei Umstände entstanden.

1. Jedes umfangreiche Produkt bezieht eine große Anzahl von Unterauftragnehmern mit ein. Diese müssen Anforderungen und Entwürfe austauschen.

2. Es existieren grosse Bibliotheken mit Standardkomponenten. Über ein genormtes Austauschformat können diese optimal genutzt werden.

Haller entwickelt in [28] auf der Basis von STEP ein Produktmodell, mit dem alle relevanten Daten eines Stahlbau-Projekts abgebildet werden können. Die Produktschnittstelle liegt aktuell in der Fassung vom April 2000 vor [17].

[2] International Organisation for Standardisation
[3] Standard for the Exchange of Product Model Data

Industry Foundation Classes

Bittrich legt in [4] dar, dass der digitale Datenaustausch bisher durch einen Datenaustausch zwischen den einzelnen Projektbeteiligten über zahlreiche Schnittstellen (Abbildung 2.1) geprägt ist. Daher ist das Ziel der Industry Foundation Classes die Integration aller für mehrere Fachdisziplinen relevanten Daten in ein zentrales Projektmodell (Abbildung 2.2). Ein weiteres Ziel ist nach [22] die Förderung der Zusammenarbeit aller an der Durchführung eines Bauprojekts beteiligten Fachleute.

Bei der Verwendung zahlreicher Einzelschnittstellen tritt bei jeder Übersetzung von Daten ein Informationsverlust auf (Abbildung 2.3). Bei der Verwendung eines zentralen Projektdatenmodells baut jede Bau- und Nutzungsphase auf Daten auf, die von DV-Programmen in vorhergehenden Projektphasen erzeugt und gespeichert wurden [49] (Abbildung 2.4).

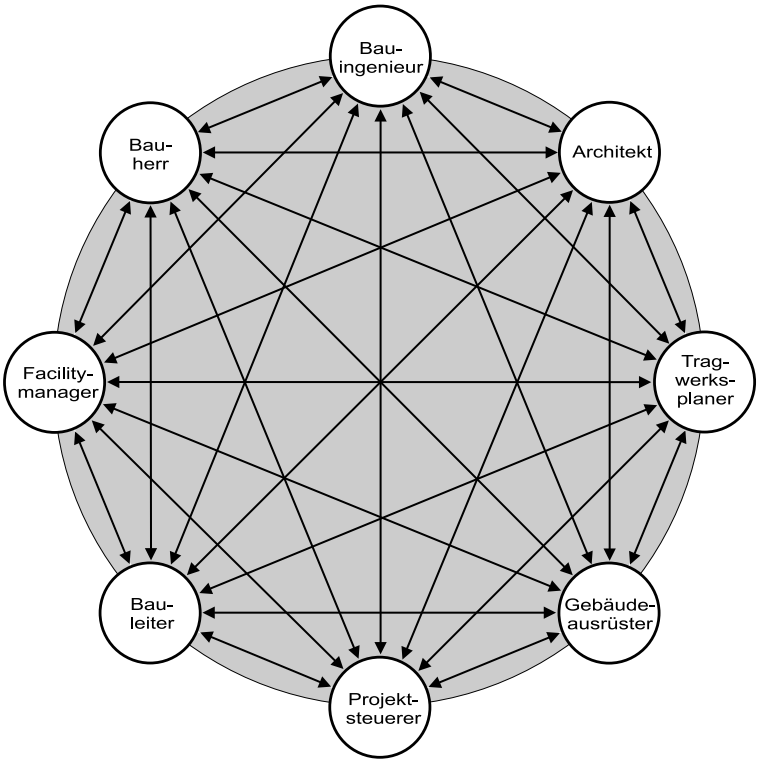

Abbildung 2.1: Konventioneller Informationsaustausch nach [4]

Bemerkungen zu Schnittstellen und Projektmodellen

Charakteristisch beim Einsatz von Schnittstellen ist weiterhin die Redundanz der vorliegenden Daten, da diese in unterschiedlichen Formaten vorgehalten werden. Sollen Daten von einem Programm A in einem Programm B weiterbearbeitet werden, so werden diese unter Zuhilfenahme der Schnittstelle in das Zielsystem übertragen. Ändern sich während der Weiterbearbeitung der Daten mit Programm B die

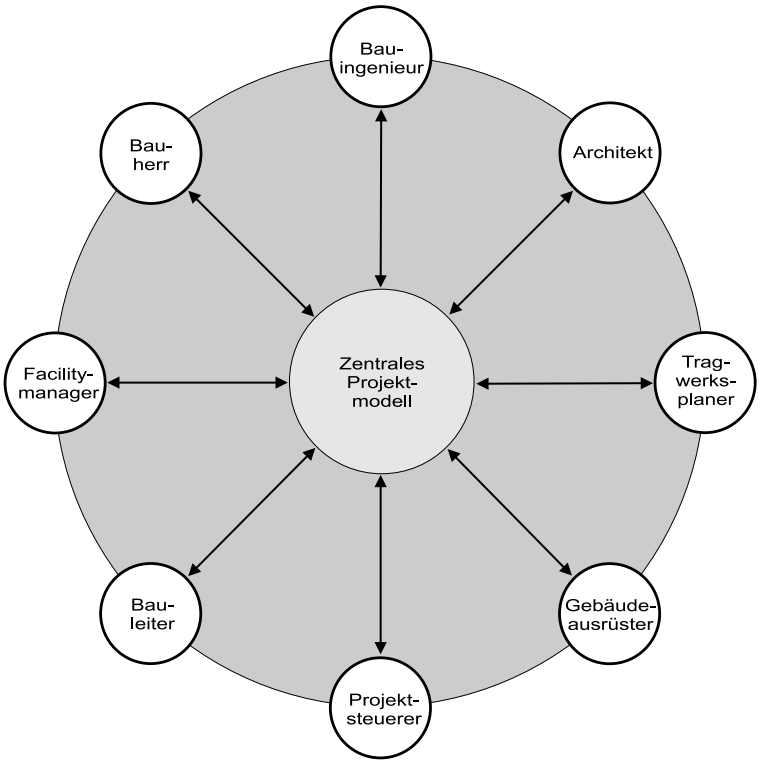

Abbildung 2.2: Interoperabilität nach [4]

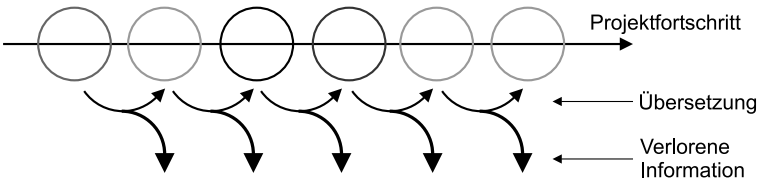

Abbildung 2.3: Herkömmliches Modell nach [49]

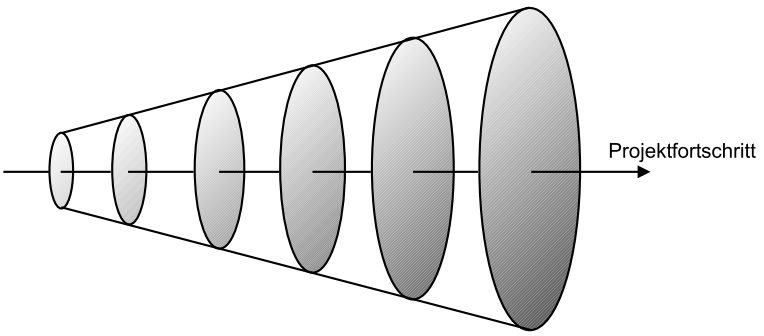

Abbildung 2.4: Gemeinsame Datennutzung

Ausgangsdaten im Programm A, liegt ein inkonsistenter Zustand vor. Während die Nutzung herstellerunabhängiger Schnittstellendefinitionen die Anzahl notwendiger Schnittstellen stark reduziert, bleibt weiterhin das Problem der Synchronisation der Datenbestände. Dieses Problem kann auch nicht dadurch beseitigt werden, dass die Daten redundanzfrei einmal in einem Datenpool bereitgestellt werden, da auch in diesem Fall eine Änderung der Ausgangsdaten keine Aktualisierung der davon abgeleiteten Daten erzwingt.

2.2.3 Prozessmodelle

Die internetbasierte Telekooperationsplattform von Müller in [47] unterstützt die Kooperation der an einem Bauplanungsprojekt beteiligten Personen. Hierbei konzentriert er sich auf das Management von schwach strukturierten Daten, die in Form von Dokumenten vorliegen. Ein Produktmodell ist nicht Bestandteil seiner Arbeit.

2.2.4 Integration von Produktdatenmodell und Prozessmodell

Bretschneider integriert in [6] ein Produktdatenmodell und ein Prozessmodell in seinem System zur kooperativen Arbeit in der Tragwerksplanung INDUSYS.

Das Prozessmodell COOPERATE

Bretschneider orientiert sich bei der Gliederung der Tragwerksplanung in Teilprozesse an den Angaben der HOAI [30]. In Anlehnung an die dort aufgeführten Grundleistungen

- Entwurfsplanung,
- Genehmigungsplanung,
- Ausführungsplanung und
- Vorbereitung der Vergabe

untersucht er die vier separaten Teilprozesse

- Vorentwurf/Bemessung,
- Strukturanalytische Berechnung,
- Nachweis/Bemessung und
- Konstruktive Durchbildung.

Weiterhin betrachtet er die Tätigkeiten der an der Tragwerksplanung beteiligten Fachleute und beschreibt die den Fachleuten zugewiesenen Rollen. Diese Rollen sind Grundlage für die Modellierung des Prozessmodells COOPERATE. Die typischen Fachleute und ihre Rollen sind in [6] wie folgt dokumentiert.

Der Bürochef ist mit der Aquisition betraut. Er benötigt Informationen über den Stand der Projektbearbeitung, greift aber nicht direkt in die Projektbearbeitung ein. Er berät das Planungsteam und arbeitet an der Lösung von Planungskonflikten.

Der Oberingenieur vertritt zusammen mit dem Projektleiter das Planungsteam nach aussen. Er berät das Planungsteam und überwacht den Projektfortschritt.

Der Projektleiter ist dafür zuständig, dass die Projektvorgaben in konkrete Planungen umgesetzt werden. Er ist für den Gesamterfolg des Projekts verantwortlich. Er plant den Arbeitsprozess, verteilt Aufgaben und koordiniert die einzelnen Tätigkeiten. Weiterhin kontrolliert er die Planungsergebnisse und er erstellt daraus den Statischen Bericht. Der Projektleiter hat darüber hinaus die Aufgabe, Konflikte innerhalb des Planungsteams zu lösen und er ist auch als normales Mitglied an der Projektarbeit beteiligt.

Der Statiker führt alle erforderlichen statischen Berechnungen aus. Hierfür entwickelt er ein statisches System, das im weiteren Projektverlauf Grundlage der Genehmigungsplanung ist.

Ingenieure unterstützen bei Bedarf den Projektleiter. Sie bearbeiten selbstständig grössere Teilprojekte, zumeist in der Entwurfs- und Genehmigungsplanung.

Techniker und CAD-Konstrukteure bearbeiten klar festgelegte Teilprobleme und führen gegebenenfalls einfache statische Berechnungen aus. Sie sind in der Genehmigungs- und Ausführungsplanung tätig.

Zeichner erstellen die technischen Zeichnungen von Details und Übersichten. Sie erstellen Stücklisten und sind mit der Massenermittlung betraut. Daher sind sie oft erst in der Ausführungsplanung an der Projektarbeit beteiligt.

Bretschneider hebt hervor, dass insbesondere paralleles Arbeiten die Durchlaufzeiten und damit die Kosten der Tragwerksplanung verringern kann. Voraussetzung hierfür ist, dass das Planungsprojekt in kleinere Teilprojekte aufteilbar ist. In der Praxis der Tragwerksplanung orientiert sich eine Aufteilung zumeist am Tragsystem (Substrukturtechnik) und an den erforderlichen Planungsaktivitäten (Aufgabenbezogene Dekomposition).

Das Produktdatenmodell PLAKON

Das Produktdatenmodell PLAKON modelliert aus ebenen oder räumlichen Teiltragsystemen aufgebaute Stahlskelett-Tragwerke des Industriebaus. Bretschneider

ermittelt die Anforderungen an Teilproduktmodelle für die vier genannten Teilprozesse.

Die sich überschneidenden Bereiche ergeben das Zentrale Produktmodell. In diesen Bereichen sind die Teilproduktmodelle aufeinander abgestimmt. Komponenten, die lediglich zur Erfüllung der Anforderungen genau eines Teilprozesses dienen, bilden bei Bretschneider Lokale Erweiterungen der einzelnen Teilproduktmodelle (Abbildung 2.5).

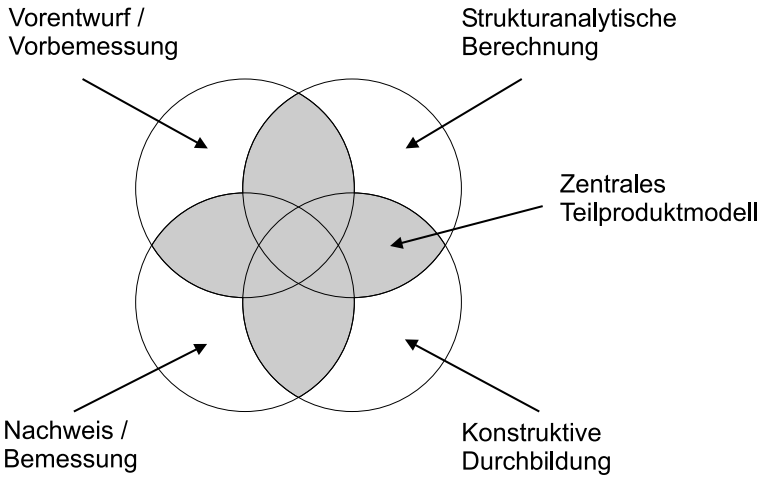

Abbildung 2.5: Produktmodellierungskonzept in [6]

2.2.5 Integrierte Projektdatenbanken

Amor und Faraj beurteilen in [3] die Möglichkeiten Integrierter Projektdatenbanken. Zunächst definieren sie, welche Eigenschaften eine Integrierte Projektdatenbank haben muss. Nach Meinung der Autoren sollen die folgenden Funktionalitäten unterstützt werden.

- Konfigurationsmanagement.
- Möglichkeit der Speicherung von Entwurfsbegründungen über alle Stadien des Projektlebenszyklus hinweg.
- Integration einer Informationsinfrastruktur zur Kommunikation.
- Visualisierung mit Multi-Media- oder Virtual-Reality-Werkzeugen.
- Offene Architektur, erweiter- beziehungsweise änderbar zur Anpassung an individuelle Projekt- und Teambedürfnisse.

Weiterhin geben sie einen Überblick über die bisherigen Ansätze. Nach [3] lassen sich diese Ansätze wie folgt klassifizeren.

Projektmodell als Referenzmodell: Dieser Ansatz wird nach Meinung der Autoren von den Praktikern bevorzugt. Er basiert auf der Anwendung von 3D-CAD- oder Virtual Reality-Modellen, die als Referenzmodell für das Projektteam dienen. In diesem Fall beinhaltet das Modell nicht zwingend alle Projektinformationen sondern dient als Zugang dazu.

Zentrale Projektdatenbank: Bei der Zentralen Projektdatenbank handelt es sich um eine zentrale Datenbank, zu der die Projektmitglieder Zugang haben. Da alle Projektinformationen in einer Datenhaltung abgespeichert werden, wird diese Datenbank sehr gross.

Verteilte Projektdatenbank: Bei der Verteilten Projektdatenbank werden die projektbezogenen Daten nicht in einer zentralen Datenbank abgespeichert, sondern in Datenbanken an mehreren Standorten abgelegt. Der Zugriff auf die Daten kann beispielsweise über CORBA [48] erfolgen. Als eine Folge hiervon müssen alle verwendeten Applikationen diese Standardschnittstelle unterstützen. Nach Amor und Faraj sind diese Verteilten Projektdatenbanken potentiell sehr effektiv.

Projektdatenbanken in neutralem Format: Die Daten können in der Projektdatenbank in einem neutralen Format abgespeichert werden. Hierzu wandelt jede Applikation die Daten in das neutrale Format um. Dieses Format kann beispielsweise auf STEP [36] basieren.

Proprietäre Ansätze: Zusätzlich zu den bisher genannten Ansätzen gab es auch zahlreiche proprietäte Ansätze. Einige dieser Systeme sind in [3] aufgeführt.

2.2.6 Missverständnisse im Bereich der Integrierten Projektdatenbanken

Amor und Faraj legen in [3] weiter dar, dass aus ihrer Sicht im Bereich der Ansätze für Integrierte Projektdatenbanken einige Missverständnisse existieren. Diese Missverständnisse werden im Folgenden dargestellt.

Missverständnis 1: „Objektorientierung stellt eine Komplettlösung bereit“

Die Eigenschaft der objektbasierten Systeme ist es, dass mit ihnen recht einfach die Sicht eines Nutzers auf die reale Welt modelliert werden kann. Daraus folgt aber auch, dass objektbasierte Systeme im Vergleich zu beispielsweise relationalen Systemen schneller Inkonsistenzen und Redundanzen beinhalten.

Die Objektorientierte Modellierung und Programmierung ist nach Meinung der Autoren nicht für sehr grosse Systeme geeignet. Gerade die Systeme der Bauindustrie gehören zu den grössten und komplexesten Systemen, die je entwickelt wurden.

Objektorientierte Systeme sind nur sehr schwer auf Konsistenz und Redundanzfreiheit hin zu überprüfen. Zur Prüfung existiert derzeit kein zugrundeliegender Mechanismus wie beispielsweise für relationale Systeme.

Zusammenfassend ergibt sich hieraus, dass Objektorientierte Modellierung und Objektorientierte Systeme kein Allheilmittel für die Systeme der Bauindustrie sind.

Missverständnis 2: „Genau ein Datenmodell wird sich entwickeln“

Die Idee zur Entwicklung genau eines Datenmodells entstand mit dem Beginn der Arbeiten am STEP-Datenmodell. Die bisherigen Erfahrungen zeigen, dass ein umfassendes Datenmodell zu gross wäre. Ein Gesamtdatenmodell müsste beispielsweise alle Informationen für die Anforderungen von Bauherren, Architekten, Ingenieuren, Konstrukteure, Facility Manager und anderen Fachleuten beinhalten. Hierbei existieren Überschneidungen zu anderen Industriezweigen, beispielsweise Schiffsbau und Klimatechnik. In vielen dieser Bereichen gibt bereits Datenmodelle. Ein umfassendes Datenmodell müsste zehntausende von Objekten beschreiben, wobei vielfältige Beziehungen zwischen diesen Objekten existieren. Die Autoren geben zu bedenken, dass derart umfangreiche und komplexe Systeme nicht zu beherrschen sind.

Missverständnis 3: „Die Datenmodelle repräsentieren die Realität“

Ein Datenmodell kann immer nur eine vereinfachte Abbildung der Realität sein. Die Modellierung bedingt immer Entscheidungen darüber, welche Eigenschaften abgebildet werden. Aus der Sicht der Autoren ist unklar, wie eine Modellierung überprüft werden soll.

Missverständnis 4: „Die Abbildung der Datenmodelle untereinander gelingt problemlos“

Die Entwicklung von Funktionen zur Abbildung eines Datenmodells auf ein anderes Datenmodell ist komplex. Dies offenbart sich insbesondere dann, wenn in den unterschiedlichen Datenmodellen ein unterschiedlicher Detaillierungsgrad für ein zu beschreibendes Objekt erforderlich ist. So ist beispielsweise in einem Datenmodell die Angabe eines Trägheitsmoments ausreichend, während in einem anderen Datenmodell die explizite Angabe von Bauteilabmessungen erforderlich ist. In solchen Fällen ist eine automatische Abbildung nicht möglich.

Missverständnis 5: „Das Internet löst alle Kommunikationsprobleme“

Das Internet bietet durch seine gute Verfügbarkeit eine Basis für einen Datenaustausch. Die Kontrollmechanismen sind jedoch gemessen an den Anforderungen zur Abwicklung von umfangreichen Bauprojekten sehr primitiv.

Missverständnis 6: „XML löst das Problem der Datenrepräsentation“

XML ist lediglich eine weitere Sprache zur Repräsentation hierarchich strukturierter Daten. Das STEP physical file format ist dazu vergleichbar. Wichtig ist, dass ein allgemein anerkannter Standard entsteht. Wenn vereinbarte Datenrepräsentationen existieren, dann kann die Nutzung von XML tatsächlich Vorteile bringen, weil Standard-XML-Werkzeuge Anwendung finden können.

Missverständnis 7: „Papierdokumente werden verschwinden“

An Bauprojekten arbeiten eine Vielzahl unterschiedlicher Beteiligter. Das Spektrum kann sich hierbei von Ein-Mann-Firmen bis hin zu multinationalen Organisationen erstrecken. Hierbei kann nicht davon ausgegangen werden, dass alle Projektbeteiligten identische Möglichkeiten zur Nutzung digitaler Daten besitzen. Insbesondere haben nicht alle Beteiligten identische Möglichkeiten, Daten elektronich mit anderen zu nutzen. In solchen Fällen werden Daten über Papierdokumente ausgetauscht.

Gedruckte Dokumente werden weiterhin existieren und wichtig bleiben. Beispiele hierfür sind ein ausgedruckter Plan mit Notizen, der mit auf eine Baustelle genommen wird, unterzeichnete Verträge und notarielle Urkunden.

Missverständnis 8: „CAD ist das Zentrum einer Integrierten Projektdatenbank“

Ein CAD-System ist nicht zwingend das Zentrum einer Integrierten Projektdatenbank. Beispielweise kann ein Produktmodell auch über mehrere DV-Systeme hinweg verteilt sein.

Missverständnis 9: „Integrierte Projektdatenbanken lösen das Problem der Eigentumsrechte der Information“

Die Unterscheidung von frei zugänglichen Daten und vertraulichen Daten ist von grosser Wichtigkeit. Daten, die von mehreren Projektbeteiligten genutzt werden sollen, müssen als frei zugängliche Daten deklariert werden. Hierzu ist ein Freigabeprozess erforderlich. Aktuelle Datenmodelle beinhalten bereits Mechanismen zur Unterscheidung von Schreib-, Lese- und Löschrechten. Problematisch ist die Klärung, wer diese Berechtigungen für einzelne Benutzer oder Gruppen festlegt und wie die Rechte und Pflichten verteilt werden. Hierüber muss jeweils Einigkeit erzielt werden.

Missverständnis 10: „Integrierte Projektdatenbanken garantieren koordinierte und konsistente Informationen“

Das Hauptargument für eine Integrierte Projektdatenbank ist die Zusammenarbeit auf Grundlage von koordinierter und konsistenter Information. In diesem Bereich

– *Computer Supported Collaborative Work* [4] – wird derzeit aktiv geforscht, so beispielsweise in den folgenden Bereichen.

Datenzugriff: In diesem Bereich wird untersucht, welche Projektbeteiligten zu einem bestimmten Projektstadium welche Informationen benötigen.

Datenänderung: Ausgehend von der Frage, wer bestimmete Daten ändern darf wird weiterhin untersucht, welche Projektbeteiligten von einer Datenänderung unterrichtet werden müssen.

Sperrungen und Transaktionen: Bestimmte Arbeitsvorgänge erfordern, dass Projektbeteiligte für einen längeren Zeitraum kontinuierlich mit Inhalten der Projektdatenbank arbeiten. Hierbei ist zu klären, wir der gleichzeitige Zugriff auf die Daten geregelt werden soll.

Datenaufteilung und Datenintegrität: Arbeiten mehrere Projektbeteiligte an unterschiedlichen Daten der Projektdatenbank, so ist zu klären, wie diese Daten ausgelesen und gespeichert werden. Hierbei ist zu klären, ob beispielsweise die Projektbeteiligten nur die von ihnen genutzten Daten speichern oder ob immer das gesamte Modell gespeichert werden muss.

2.2.7 Verteilte Bearbeitung von Objekten in der Bauplanung

Firmenich stellt in [21] ein System zur verteilten Bearbeitung von strukturierten Mengen von Objektversionen vor. Er betrachtet hierbei die Anwendung von CAD im Bauplanungsprozess. Das Ziel der Arbeit ist die Unterstützung der an der Bauplanung beteiligten Bearbeitern in den drei Phasen

- Bildung von Teilmengen,
- Bearbeitung der Teilmengen und
- Speicherung der Ergebnisse.

In der Phase der *Bildung von Teilmengen* unterteilen die Planungsbeteiligten das gemeinsame Arbeitsmaterial. In der Phase der *Bearbeitung der Teilmengen* werden die Planungsschritte synchron oder asynchron durchgeführt. In der Phase der *Speicherung der Ergebnisse* erfolgt die Speicherung im gemeinsamen Arbeitsmaterial.

Firmenich bildet die einzelnen Objekte einer verteilten Bearbeitung in einem gerichteten Graphen ab. Die gerichteten Beziehungen untereinander zeigen an, welche Objekte voneinander abhängig sind. Weiterhin gibt es gerichtete Beziehungen zwischen unterschiedlichen Versionen des gleichen Objekts. Diese Beziehungen zeigen an, wer ein Versionsnachfolger eines Objekts ist.

Firmenich geht von einem konsistenten Ausgangszustand aus und prüft nach jeder Speicherung, ob der neue Zustand konsistent sein kann.

[4] Eine Darstellung von CSCW kann dem Anhang entnommen werden.

Beispiel aus [21]: Kooperation im Bauplanungsprozess

Die Arbeitsweise des Systems kann an einem Beispiel aus [21] gezeigt werden. Dort wird die Funktionsweise des Systems zur Unterstützung der Kooperation im Bauplanungsprozess erläutert.

Planungsschritt 1: Der Architekt legt die Geometrie **g** einer Betonstütze fest und speichert diese als Objekt $\mathbf{g}_1$ ab (Abbildung 2.6).

Abbildung 2.6: Festlegung der Stützengeometrie $\mathbf{g}_1$

Planungsschritt 2: Unter Berücksichtigung der Geometrie $\mathbf{g}_1$ ermittelt der Statiker die erforderliche Bewehrung und dokumentiert diese in einem statischen Bericht **s**. Dieser wird als Objekt $\mathbf{s}_2$ abgespeichert. Das Objekt $\mathbf{s}_2$ hängt von $\mathbf{g}_1$ ab. Diese Abhängigkeit wird ebenfalls gespeichert (Abbildung 2.7).

Abbildung 2.7: Statische Berechnung $\mathbf{s}_2$ für die erforderliche Bewehrung

Planungsschritt 3: Der Bewehrungsplaner konstruiert die Stützenbewehrung **b** und speichert diese als Objekt $\mathbf{b}_3$ ab. Diese Bewehrung $\mathbf{b}_3$ ist sowohl von der Geometrie $\mathbf{g}_1$ als auch von dem statischen Bericht $\mathbf{s}_2$ abhängig. Diese Abhängigkeiten werden gespeichert (Abbildung 2.8).

Planungsschritt 4: Der Architekt lädt die Geometrie $\mathbf{g}_1$ und verändert die Abmessungen der Stahlbetonstütze. Die veränderte Version der Geometrie speichert er als $\mathbf{g}_4$ ab (Abbildung 2.9). Mit diesem Schritt sind alle Objektversionen, die sich auf $\mathbf{g}_1$ beziehen überholt. Diese überholten Objektversionen können programmautomatisch ermittelt werden.

Planungsschritt 5: Der Bewehrungsplaner versucht, die Bewehrung an die Geometrie $\mathbf{g}_4$ anzupassen. Er lädt $\mathbf{b}_3$, modifiziert die Bewehrung und speichert

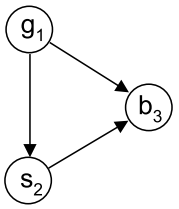

Abbildung 2.8: Konstruierte Bewehrung $\mathbf{b}_3$

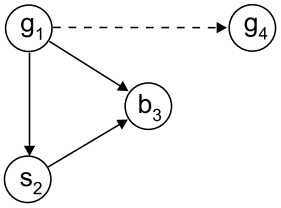

Abbildung 2.9: Veränderte Stützengeometrie $\mathbf{g}_4$

eine neue Objektversion $\mathbf{b}_5$ ab. Die Bindung von $\mathbf{b}_5$ an $\mathbf{g}_1$ entfernt er, da die neue Objektversion von $\mathbf{g}_4$ abhängig ist. Entsprechend wird eine Bindung von $\mathbf{b}_5$ an $\mathbf{g}_4$ gespeichert (Abbildung 2.10).

Der entstandene Zustand ist gemäss der Definitionen aus [21] inkonsistent. Die Objektversion $\mathbf{b}_5$ hängt von der Objektversion $\mathbf{g}_4$ ab. Weiterhin hängt $\mathbf{b}_5$ über die Objektversion $\mathbf{s}_2$ auch von der Objektversion $\mathbf{g}_1$ ab. Die Objektverision $\mathbf{g}_1$ ist aber ein Versionsvorgänger von $\mathbf{g}_4$.

Dies ist im entstandenen Graphen der Bindungen daran zu erkennen, dass ein Weg von $\mathbf{g}_4$ nach $\mathbf{b}_5$ und ein Weg vom Versionsvorgänger $\mathbf{g}_1$ über $\mathbf{s}_2$ nach $\mathbf{b}_5$ existiert.

Der Bewehrungsplaner erkennt, dass zunächst eine statische Berechnung erforderlich ist und verwirft als Folge hieraus die Objektversion $\mathbf{b}_5$.

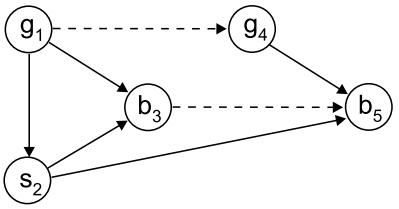

Abbildung 2.10: Veränderte Bewehrung $\mathbf{b}_5$

Planungsschritt 6: Der Statiker erstellt als Versionsnachfolger der Statik $\mathbf{s}_2$ eine aktualisierte statische Berechnung. Die statische Berechnung wird als $\mathbf{s}_6$ zusammen mit der Bindung an $\mathbf{g}_4$ gespeichert (Abbildung 2.11).

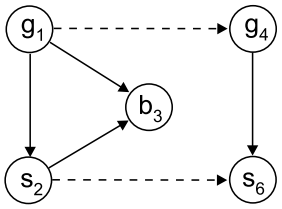

Abbildung 2.11: Aktualisierte Statik $\mathbf{s}_6$

Planungsschritt 7: Auf der Grundlage der Geometrie $\mathbf{g}_4$ und der Statik $\mathbf{s}_6$ kann der Bewehrungsplaner eine neue Bewehrung als Versionsnachfolger von $\mathbf{b}_3$ entwerfen und diese als Objektversion $\mathbf{b}_7$ speichern. Weiterhin speichert er die Bindungen von $\mathbf{b}_7$ an $\mathbf{g}_4$ und $\mathbf{s}_6$ (Abbildung 2.12). Damit ist wieder ein konsistenter Zustand erreicht.

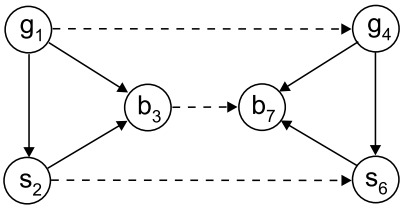

Abbildung 2.12: Neuer Bewehrungsentwurf $\mathbf{b}_7$

Zusammenfassend ergibt sich hieraus, dass eine Objektmenge inkonsistent ist, wenn im gerichteten Graphen der Bindungen eine Objektversion $\mathbf{y}_n$ existiert, die sowohl von einer Objektversion $\mathbf{x}_m$ als auch von einem Versionsvorgänger $\mathbf{x}_l$ der Objektversion $\mathbf{x}_m$ aus erreichbar ist.

Eine Konsistenzprüfung findet auf der Ebene der Objekte und ihrer Beziehungen untereinander statt. Die Objektattribute bleiben hierbei unberücksichtigt. Weiterhin können zwischen den Objektattributen keine Gültigkeitsregeln formuliert werden, welche programmautomatisch ausgewertet werden.

Das in [21] beschriebene System stellt sicher, dass ein Planungsschritt nicht auf überholten Objektversionen beruht.

2.3 Anforderungen an ein DV-System zur Bauplanung

2.3.1 Allgemeines

Lu et al. legen in [42, 43] dar, wie multidisziplinäre Entwicklungsteams bei der Durchführung von Entwurfsschritten durch Softwarewerkzeuge unterstützt werden

können.

Die Autoren unterteilen die existierenden Ansätze, die einzelne Elemente des Entwurfsprozesses behandeln, in drei Gruppen.

Die erste Gruppe entstammt den Ingenieurdisziplinen. Diese beschäftigt sich damit, wie technische Entwurfsentscheidungen getroffen und wie systematische Entwurfsmethoden eingeführt werden.

Eine zweite Gruppe entstammt Forschungen aus dem Projektmanagement. Diese betrachten den Entwurfsprozess als Workflow mit Abhängigkeiten und dem Austausch von Produktinformationen.

Die dritte Gruppe kommt aus dem Bereich CAD. Aus dieser Sicht ist eine fächerübergreifende Entwicklung ein Prozess, bei dem Individuen auf Produktdaten zugreifen und diese Daten teilen. In dieser Sicht ist der Entwurfsprozess das Management von Produktdaten in unterschiedlichen Stufen der Abstraktion.

Im Rahmen der Vorstellung eines neuen Modells legen die Autoren dar, dass das Management von Entwurfskonflikten ein erfolgskritischer Vorgang im Entwurfsprozess ist.

Daraus kann abgeleitet werden, dass ein DV-System zur Bauplanung die Möglichkeit der Definition von Anforderungen und der Detektion von Entwurfskonflikten haben muss.

2.3.2 Benachrichtigungen

Benachrichtigungen können genutzt werden, um Projektbeteiligte über Änderungen an den Projektdaten zu informieren, die für sie relevant sind. Dadurch sind die Projektbeteiligten stets auf dem aktuellen Stand [8]. Werden im Projektverlauf Regeln verletzt, die zuvor als Constraints definiert wurden, können diejenigen Bearbeiter benachrichtigt werden, die von dieser Verletzung betroffen sind oder die diese Verletzung korrigieren können [27, 32].

2.3.3 Versionierung

Im Rahmen der Durchführung eines Bauplanungsprojektes werden durch die unterschiedlichen Bearbeiter zahlreiche Planungsschritte durchgeführt und hierbei Daten verändert. Zur Nachvollziehbarkeit der Projektfortschritte ist es notwendig, den Planungszustand zu beliebigen Zeitpunkten rekonstruieren zu können. Hierfür müssen die verschiedenen Versionen der Objekte abgespeichert sein.

2.3.4 Konfigurationsmanagement

Die Möglichkeit der Versionierung geht einher mit der Möglichkeit der Definition von *Konfigurationen*. Die zu einem bestimmten Zeitpunkt im Verlauf der Projektplanung

aktuellen Versionen der Daten bilden zusammen eine *Konfiguration*. Die Möglichkeit der Definition und Verwaltung von Konfigurationen dient der Realisierung der Nachvollziehbarkeit der Planungsschritte, da jede zu einem beliebigen Zeitpunkt bestehende Konfiguration wiederhergestellt werden kann. Weiterhin unterstützt das Konfigurationsmanagement die Durchführung von Was-wäre-wenn-Analysen. Verschiedene Varianten einer Lösung können als verschiedene Versionen angelegt werden und dann in verschiedenen Konfigurationen zusammengestellt werden. Anschliessend kann eine Entscheidung darüber getroffen werden, welche Variante weitergeführt wird.

2.3.5 Constraints

Gupta und Tiwari schlagen in [27] die Verwendung von *Constraints* zur Definition von Anforderungen an den Entwurf vor. Zusätzlich beschreiben sie die Architektur eines verteilten Constraint-Management-Systems, welches Inkonsistenzen durch die programmautomatische Überprüfung der Constraints auffinden kann. Nach Gupta und Tiwari haben die Constraints die gleiche Wichtigkeit wie die Daten, auf denen diese Constraints definiert werden. In [32] ist ein DV-System beschrieben, bei dem für die Planung von Bauprojekten Constraints definiert werden können. Eine Verletzung dieser Constraints kann im Projektverlauf programmautomatisch festgestellt werden. Das in [32] beschriebene System arbeitet auf der Grundlage relationaler Datenbanken und erlaubt die Definition von Constraints durch eine Erweiterung der Datenbankabfragesprache SQL.

2.3.6 Produktstruktur und Komponentenmanagement

Ein DV-System zur Bauplanung muss die Bestandteile eines Bauwerks abbilden können. Bauwerke als Unikatprodukte bestehen aus in Serie hergestellten Komponenten. Entsprechend muss das System die Wiederverwendung und die Verwaltung von Standardkomponenten ermöglichen.

Kapitel 3

Grundlagen des Produktdatenmanagement und des Software-Konfigurationsmanagement

3.1 Allgemeines

Eine Definition für PDM[1] und SCM[2] geben Crnkovic et al. in [12].

> *Product Data Management (PDM) is the discipline of designing and controlling the evolution of a product design. Software Configuration Management (SCM) is the discipline of controlling the evolution of a software product design.*

Estublier legt in [19] dar, dass unterschiedliche Ingenieurdisziplinen, wie beispielsweise Bauingenieurwesen, Maschinenbau und Elektotechnik nach Methoden gesucht haben, unterschiedliche Produkte wie Bauwerke, Flugzeuge oder Autos zu entwerfen und zu realisieren. Bei all diesen Objekten handelt es sich um komplexe Produkte. Gemeinsam ist diesen komplexen Produkten, dass sie aus Teilen zusammengesetzt sind. Jedes dieser Teile hat eigene Eigenschaften. Die notwendige Kontrolle über eine grosse Anzahl an Komponenten und ihrer Weiterentwicklung sind der Grund für *Produktmanagement*. Eine alleinige Betrachtung der Entwurfsdaten ist das *Produktdatenmanagement*.

Ansätze und Lösungen im Bereich des Produktdatenmanagements gab es schon lange bevor Computer im Arbeitsalltag Einzug gehalten haben. Seit den Siebzigern des 20. Jahrhunderts entwickelt sich die Computerunterstützung für Produktionsprozesse. Programmsysteme zur Unterstützung des Produktionsprozesses entstehen. Diese können in zwei Kategorien eingeteilt werden.

[1] Produktdatenmanagement
[2] Software-Konfigurationsmanagement

1. Anwendungsspezifische Werkzeuge (Zeichenprogramme, Statikprogramme, Texteditoren), deren Ziel der Entwurf eines Teils des Produkts ist.

2. Werkzeuge für das Management des Produkts als Zusammenbau unterschiedlicher Teile (Speicherung als Produktmodell, Zusammensetzung des Produkts, Entwicklungsverlauf).

Weiterhin entsteht durch die einsetzende Nutzung von Computersystemen ein neues Produkt: Software.

Magnusson betont in [44] die Gemeinsamkeiten von Software-Konfigurationsmanagement und Produktdatenmanagement und fasst diese zum *Konfigurationsmanagement* zusammen. Das Konfigurationsmanagement beschäftigt sich nach [44] mit dem Management der Evolution von Familien von Systemen. Sie beinhaltet die Unterstützung der

- Versionierung,
- Zusammensetzung und
- Erzeugung

aller relevanten Bestandteile eines Produkts. Weiterhin beinhaltet das Konfigurationsmanagement die Unterstützung und die Kontrolle der damit zusammenhängenden Aktivitäten eines Teams von Bearbeitern. Konfigurationsmanagement ist ein zentraler Bestandteil jedes umfangreichen Ingenieurprojekts.

Die Dienste eines Systems zum Konfigurationsmanagement können nach [19] in vier Hauptklassen unterschieden werden.

1. Produktmodell – Modell der Komponenten und ihrer Beziehungen untereinander.

2. Versioniertes Produktmodell – Versionierung, Komposition, Auswahl.

3. Beziehungen mit den Ingenieurdisziplin-spezifischen Werkzeugen – Workspaces, Concurrent Engineering.

4. Prozessmodell – Kontrolle von Änderungen.

3.2 Produktdatenmanagement – PDM

3.2.1 Weitere Definitionen

Dahlkvist et al. stellen in [13] dar, dass zahlreiche unterschiedliche Bezeichnungen für Produktdatenmanagement existieren. Einige davon listet Tabelle 3.1 auf.

PDT	Product Data Technology
CPC	Collaborative Product Commerce
ePDM	electronic Product Data Management
ePLDM	electronic Product LifeCycle Definition Management
ICM	Intellectual Capital Management
PKM	Product Knowledge Management
VPDM	Virtual Product Data Management
PDM	Product Data Management
CM(II)	Configuration Management (II)
CM	Component Management
PIM	Product Information Management
TIM	Technical Information Management
EDB	Engineering Database
PDV	Produktdatenverwaltung
EDM	Engineering Data Management
EPDM	Enterprise Product Data Management
PDM II	Product Development Management

Tabelle 3.1: Bezeichnungen für PDM aus [13] und [53]

Die Bezeichnung PDM umfasst das gesamte Themengebiet und die Methodik während die Bezeichnung PDM-System ein Werkzeug bezeichnet.

Nach [13] ist für PDM folgende Definition wiedergegeben:

> *PDM manages product data throughout the enterprise, ensuring that the right information is available for the right people, at the right time, and in the right format.*

Für den Begriff PDM-System:

> *Product Data Management (PDM) is a tool that helps people manage both product data and the product development process. PDM systems keep track of the masses of data and information required to design, manufacture or build, and then support and maintain products – whether your product is an aeroplane, petrochemical plant, highway, railway system, pharmaceutical, automobile, consumer product, or service. PDM is used effectively in a multitude of industries.*

Schoettner schreibt in [53], dass Produktdatenmanagement der Ansatz sei, ein Fertigungsunternehmen im Hinblick auf den Produktentstehungsprozess mit einem Datenmodell abzubilden. Die auf diesem Ansatz aufgebauten PDM-Systeme sind modulare Softwaresysteme zur Steuerung aller Arbeitsprozesse und zur Verwaltung aller hierbei entstehenden Daten und Dokumente.

Westfechtel et al. verdeutlichen in [59], dass PDM das Management von Daten behandelt, die reale Objekte beschreiben.

3.2.2 Wesentliche Funktionen

Einen Überblick über die Funktionalitäten für Produktdatenmanagement-Systeme geben Dahlkvist et al. in [13].

Dokumentenmanagement: Der Bereich des Dokumentenmanagements eines PDM-Systems dient als zentrale Datenhaltung. Zu diesem Bereich gehören Funktionen zum Speichern und Suchen von Dokumenten und Funktionen zu einem rechtebasierten Zugriff auf die enthaltenen Daten. Weiterhin Funktionen zur Versionierung für Dokumente wie auch für Bauteile, Definition und Pflege von Beziehungen von Bauteilen untereinander. Zusätzlich können auch Funktionen für die Freigabe und Veröffentlichung von Dokumenten integriert sein.

Workflow- und Prozessmanagement: Die Definition der Arbeitsabläufe ist ein wesentlicher Teil eines PDM-Systems. Er dient dazu, die Prozesse zu steuern und nachvollziehbar zu machen. Das Prozessmanagement verteilt Informationen und Arbeitsaufträge an Mitarbeiter und steuert die Erzeugung, Änderung und Nutzung von Daten und Dokumenten [53].

Management der Produktstruktur: Funktionen zum Management der Produktstruktur erlauben die Definition und die Verwaltung von Produktkonfigurationen, die Verwaltung von Produktvarianten mit ihren Optionen, Alternativen und Ersatzkomponenten. Die Funktionen unterstützen die notwendigen fachspezifischen Sichten auf die Daten und die Produktstruktur. Hier sind auch Funktionen zum Austausch der Produktstruktur mit anderen Systemen angesiedelt.

Teile- und Komponentenmanagement: Das Teile- und Komponentenmanagement unterstützt die Wiederverwendung von Standardkomponenten, indem diese klassifiziert werden können und mit Informationen über Lieferanten abgespeichert und gesucht werden können.

Projektmanagement: Die Funktionen des Bereichts Projektmanagement ermöglichen die Festlegung notwendiger Arbeiten im Rahmen der Produktentstehung.

Hierbei sind die Funktionen für das

- Management der Produktstruktur und das
- Teile- und Komponentenmanagement

für das Produktdatenmanagement von zentraler Bedeutung.

3.2.3 Produktmodelle

Ein Produktmodell ist die Grundlage aller Funktionen für das Management der Produktstruktur und das Teile- und Komponentenmanagement.

Ungeachtet des Produkts sind die grundlegenden Konzepte für ein Produktmodell die eines *Bauteils* und einer *Baugruppe*. Zwischen einer Baugruppe und einem Bauteil besteht eine *Besteht-aus-Beziehung* [19].

In PDM-Produktmodellen existieren unterschiedliche Arten der Abbildung von Produktzusammensetzungen. Eine Möglichkeit der Abbildung einer Produktzusammensetzung zeigt Abbildung 3.1. Dargestellt sind die Komponenten von Fahrrädern und ihre Besteht-aus-Beziehungen. Die Komponenten besitzen keine direkte Entsprechung in der realen Welt, sondern stellen Modelle dar. Diese Abbildung einer Produktzusammensetzung entspricht dem Prinzip "nodes are models". Die Abbildung eines konkreten Fahrrads zeigt Abbildung 3.2. Jede der dargestellten Komponenten hat jetzt eine direkte physische Entsprechung. Diese Abbildung einer Produktzusammensetzung entpricht dem Prinzip "nodes are instances".

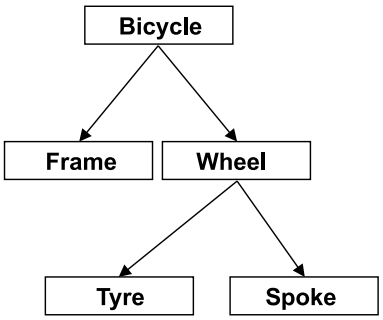

Abbildung 3.1: Beispiel für "nodes are models" aus [19]

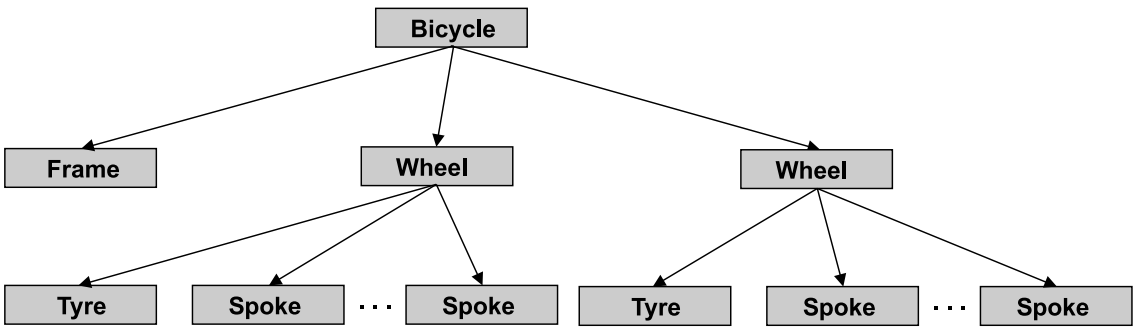

Abbildung 3.2: Beispiel für "nodes are instances" aus [19]

3.2.4 Herstellerübergreifende Standards zum Datenaustausch

Haller beschreibt in [28] die Entwicklung der Produktmodelle. Ausgangspunkt für die Entwicklung waren Vereinbarungen von Datenaustauschformaten für CAD-Systeme. Wie in Kapitel 2.2.2 dargelegt, bildete die ISO (International Organisation for Standardisation) zur Standardisierung von Schnittstellen Mitte der achziger Jahre die Arbeitsgruppe ISO TC 184/SC. Das Ziel war die Schaffung einer Norm, die ein Datenaustauschformat definert, mit dem alle produktbeschreibenden Daten erfasst werden können. Diese Schnittstelle heisst STEP (Standard for the Exchange of Product Model Data) [36].

Mit STEP sollen alle im Produktlebenszyklus entstehenden Daten bearbeitet werden können und es ist das Ziel, alle Ingenieurdisziplinen wie beispielsweise Maschinenbau, Elektrotechnik, Schiffbau, Architektur und Bauwesen abzudecken [28]. STEP beinhaltet zur Produktmodellierung die Sprache EXPRESS.

3.2.5 Datenmodelle

Im Produktdatenmanagement ist EXPRESS das Datenmodell für STEP. EXPRESS ist eine objektorientierte Modellierungssprache für die statische Definition strukturierter Produkte [19]. Mit EXPRESS können alle Aspekte zur Beschreibung eines Produktes formuliert werden [28]. Ein in EXPRESS beschriebenes Produktmodell kann programmautomatisch auf seine Syntax überprüft werden. Die Objektklassen definieren eine Menge an Attributen und eine Menge an Einschränkungen für diese Attribute. Ein Attributwert kann auch eine Referenz zu einem anderen Objekt sein. So lassen sich Kompositionen realisieren. EXPRESS beschreibt keine dynamischen Aspekte wie das Datenmanagement und das Verhalten. Weiterin stellt EXPRESS keine Methoden zur Verfügung [19].

3.2.6 Versionierung

Das Versionierungskonzept für das Produktdatenmanagement kennt drei Dimensionen.

Historisches Versionieren: Das Historische Versionieren dient dem Management unterschiedlicher Revisionen, die im zeitlichen Verlauf einer Produktentwicklung entstehen.

Logisches Versionieren: Jedes Teil eines Produkts kann in verschiedenen Versionen existieren. Dabei handelt es sich beispielsweise um verschiedene Ausführungen eines Bauteils. Für ein herzustellendes Produkt kann jeweils eine Ausführung des Bauteils ausgewählt werden.

Domänenbezogenes Versionieren: Jeder an der Entwicklung eines komplexen Produkts beteiligte Spezialist benötigt zur Durchführung seiner Aufgabe lediglich den Zugriff und die Sicht auf diejenigen Daten, die für ihn relevant sind.

3.2.7 Arbeitsumgebung

Im Produktdatenmanagement werden zur Bearbeitung von Produktdaten spezielle Editoren, beispielsweise 3D-CAD-Programme und Berechnungswerkzeuge eingesetzt.

Hierbei sind die relevanten Daten im jeweils notwendigen Format bereitzustellen. Änderungen an den Daten müssen an das PDM-System weitergegeben werden.

3.2.8 Prozessmodell

In STEP basiert die Prozessunterstützung auf dem Genehmigungs-Konzept. Zu jeder Genehmigung gehört eine Person, ein Datum und eine Organisation. Der Status einer Genehmigung kann

- genehmigt,
- noch nicht genehmigt,
- abgelehnt oder
- zurückgenommen

sein.

3.3 Software-Konfigurationsmanagement – SCM

3.3.1 Weitere Definitionen

Lin legt in [39] dar, dass jedes umfangreiche Softwareprodukt aus einer Vielzahl unterschiedlicher Komponenten zusammengesetzt ist. Zu diesen Komponenten gehören beispielsweise

- Source-Code,
- Binärcode,
- Spezifikationen und
- Dokumentationen.

Die Version einer jeden Komponente, mit der eine bestimmte Version des Endprodukts erzeugt wird, nennt man die *Konfiguration* dieser Version des Endprodukts. *Software-Konfigurationsmanagement* ist die Identifizierung, Organisation und Kontrolle von Änderungen an einem Softwaresystem, das von einem Team von Programmierern entwickelt wird. Die wichtigsten Probleme des Software-Konfigurationsmanagements sind der Systemaufbau und die Versionskontrolle.

3.3.2 Wesentliche Funktionen

Die Funktionen eines SCM-Systems resultieren aus den Aufgaben und Zielen der an einer Softwareentwicklung beteiligten Personen und der *Rollen*, die diese Personen einnehmen. Dart stellt in [14] ein typisches Szenario der Benutzer eines SCM-Systems dar.

Der *Projektmanager* ist dafür verantwortlich, ein Softwareprodukt innerhalb des spezifizierten zeitlichen Rahmens fertig zu stellen. Daher beobachtet er den Fortschritt der Entwicklung, er erkennt auftretende Probleme und reagiert auf diese. Hierfür generiert und analysiert er Berichte über den Status des Software-Systems.

Aufgabe des *Konfigurationsmanagers* ist es, sicher zu stellen, dass Regelungen für das Erzeugen, Ändern und Testen von Source-Code eingehalten werden. Weiterhin muss er sicherstellen, dass Informationen über das Projekt verfügbar sind.

Das Ziel der *Softwareentwickler* ist es, das Projekt effizient umzusetzen. Das bedeutet, dass diese sich nicht unnötig gegenseitig beim Erzeugen und Testen von Code und bei der Dokumentation stören. Gleichzeitig müssen sie effizient miteinander kommunizieren und kooperieren.

Der *Kunde* benutzt das Softwareprodukt. Da ein SCM-System eingesetzt wird, folgt der Kunde bei Änderungsanfragen und Fehlermitteilungen festgelegten Vorgehensweisen.

Ein ideales SCM-System unterstützt alle in diesem Szenario beschriebenen Rollen und Aufgaben. Die Rollen definieren die benötigte Funktionalität des SCM-Systems. Jede Rolle erfordert einen entsprechenden Vorrat an Funktionen.

Für die Rolle des Projektmanagers bietet ein SCM-System Funktionen eines Planungsinstruments, für die Rolle des Konfigurationsmanagers Kontroll- und Koordinierungsfunktionen und für die Rolle des Softwareentwicklers bietet es Funktionen zum Anlegen und Ändern von Daten. Aus der Rolle des Kunden bietet das KM-System Funktionen eines Qualitätssicherungssystems.

Nach [14] können die rollenspezifischen Anforderungen an ein SCM-System in die folgenden Funktionsbereiche klassifiziert werden.

Komponenten: Die Funktionen dieses Bereichs dienen der Identifizierung, Klassifizierung und dem Zugriff auf die Komponenten des projektierten Softwareprodukts. Benutzer speichern Versionen ihrer Komponenten, ihre Änderungen und die Gründe, die diese Änderungen notwendig machen. Weiterhin legen sie Komponenten und ihre Versionen fest, die eine Konfiguration bilden.

Struktur: Dieser Bereich beinhaltet Funktionen zur Repräsentation der Struktur des Produkts. Benutzer definieren und Bearbeiten die Struktur des Produkts mit Hilfe eines Modells, das die Komponenten dieses Produkts repräsentiert. Sie spezifizieren Schnittstellen zwischen den Komponenten und wählen kompatible Komponenten als Bestandteile des Produkts aus.

Konstruktion: Innerhalb dieses Funktionsbereichs befinden sich Funktionen zur Unterstützung beim Aufbau des Produkts. Benutzer erzeugen eine definierte Version des Produkts im Sinne einer systemgesteuerten Übersetzung in ausführbaren Programmcode. Weiterhin befinden sich hier Funktionen zum Einfrieren oder Abspeichern einer aktuellen Konfiguration.

Audit: Zum Bereich Audit zählen Funktionen zur Ermittlung des aktuellen Entwicklungsstands des Produkts. Benutzer erhalten eine Darstellung aller durchgeführten Änderungen. Weiterhin werden Funktionen zur Aufzeichnung aller durchgeführten Arbeiten bereitgestellt.

Informationsbeschaffung: Dieser Bereich bietet Funktionen zum einfachen Erstellen von Berichten über alle Aspekte des Produkts. Benutzer erstellen Statistiken über die Entwicklung und den Entwicklungsprozess.

Kontrolle: Mit Funktionen dieses Bereichs wird kontrolliert, wie und an welchen Stellen Änderungen durchgeführt wurden. Sie garantieren einen kontrollierten Zugriff auf die Komponenten im System. Dienste zur Online-Unterstützung gehören ebenfalls in diesen Bereich. Hierzu zählen Funktionen zur Beantragung von Änderungen und zum Berichten über Probleme.

Prozess: In diesem Funktionsbereich werden Funktionen zusammengefasst, die das Management unterstützen. Benutzer erhalten Informationen, wie das Produkt sich entwickelt und Unterstützung beim Bestimmen von Art und Umfang noch zu erledigender Arbeiten. Ebenfalls enthalten sind Funktionen zur Kommunikation zwischen den Mitarbeitern. Ein weiterer wesentlicher Bestandteil sind Dienste zur Dokumentation von Wissen über das Produkt.

Team: Funktionen dieses Bereichs ermöglichen es einem Team, Softwareprodukte zu entwickeln und zu warten. Sie stellen Arbeitsbereiche für Gruppen oder einzelne Personen zur Verfügung und ermöglichen das Zusammenführen von Änderungen und die Analyse auftretender Konflikte. Weitere Funktionen bieten Unterstützung beim Erstellen einer Familie von Produkten.

3.3.3 Herstellerübergreifende Standards zum Datenaustausch

Nach [19] ist in der Softwareindustrie eine ausgeprägte Struktur mit vielen Unterauftragnehmern noch kein Hauptanliegen. Ein zu STEP vergleichbares Konzept ist nicht vorhanden. Jedes SCM-System beinhaltet seine eigenen Konzepte und Mechanismen.

3.3.4 Datenmodelle

Ein standardisiertes Datenmodell ist nach [19] nicht vorhanden. Aktuell setzen die meisten Systeme auf den Konzepten der Dateien und der Verzeichnisse auf. Damit ist ein Management eines komplexen Objekts nicht möglich. Lin beschreibt in [39] davon abweichend ein System mit einem objektorientierten Modell.

3.3.5 Produktmodelle

Zahlreiche SCM-Werkzeuge basieren auf der Grundlage, dass ein Softwareprodukt einen Baum aus Dateien und Ordnern darstellt [19]. Ihr Modell ist demnach ein Dateisystem-Modell, und die Struktur einer Software wird durch einen Baum repräsentiert. Zur Darstellung detaillierterer Produktmodelle wurden Systemmodelle mit dem Ziel eingeführt, Beziehungen zwischen den Komponenten zu definieren. Diese Produktmodelle sind Graphen mit Dateien als Knoten. Das Problem dieser SCM-Systeme ist, dass die Struktur der Abhängigkeiten – modelliert als Graph – die Dateisystem-Struktur nicht ersetzt. Vielmehr existieren beide, und sie sind nicht identisch. Darüber hinaus arbeiten Softwareentwickler mit einer weiteren Strukturierung, die auf Konzepten wie beispielsweise Modul, Subsystem und System basiert.

Der Versuch, ein Standard-Produktmodell basierend auf einem Standard-Datenmodell zu definieren, wurde nach [19] mehrfach unternommen. Ein Standard vergleichbar zu STEP ist nicht vorhanden.

3.3.6 Versionierung

Bei SCM basiert die Versionierung auf der Basis von Dateien, die sich jeweils in der Abfolge von Revisionen entwickelt.

3.3.7 Arbeitsumgebung

In SCM werden zur Bearbeitung spezielle Editoren, Texteditoren, Compiler und Linker eingesetzt. Zur Bereitstellung der Daten, die in Form von Textdateien vorliegen, hat sich die Anwendung von check-in/check-out-Protokollen verbreitet. Diese verhindern, dass ein Datensatz in getrennten Arbeitsumgebungen zeitgleich geändert werden kann.

3.3.8 Prozessmodell

In SCM ist die Prozessunterstützung nach [19] schwach ausgeprägt.

3.4 Gemeinsamkeiten und Unterschiede

Die Tabelle 3.2 aus [19] listet die Gemeinsamkeiten und Unterschiede von Produktdatenmanagement und Software-Konfigurationsmanagement auf.

Vergleicht man den Bereich des Software-Engineerings mit anderen Ingenieurdisziplinen, werden zwei Hauptunterschiede deutlich.

1. Im Bereich der Produktentwicklung gibt es eine klare Trennung zwischen dem Modell und seiner Entsprechung in der realen Welt. Im Software-Engineering ist der Quellcode das Modell, aus dem ein Übersetzer beinahe ohne Kosten das Produkt erzeugt. Die Software ist Modell und Produkt zugleich.

2. Die Struktur eines Produkts und die Eigenschaften seiner Komponenten sind begrenzt durch die Realität. Ein Softwareprogramm ist ein intellektuelles Konstrukt.

In beiden Bereichen – PDM und SCM – ist es für die Softwarewerkzeuge insbesondere erforderlich,

- Versionierung und
- Konfigurationsmanagement

zu unterstützen.

Im Bereich PDM basieren die Softwaresysteme auf dem Management von Objekten, in SCM findet zumeist ein Management von Dateien statt. Lin beschreibt in [40, 41] ein SCM-System, das auf dem Management von Objekten statt auf dem Management von Dateien basiert.

	PDM	SCM
Produkt	Modell $\neq$ Produkt	Modell $\approx$ Produkt
Struktur	Constraints durch Realität	Keine Begrenzungen
Reifegrad	+	−
Datenmodell	z.B. STEP	kein Standard
Hauptaufgabe	Modellierung von Objekten	Verwalten von Dateien
Komponenten	Einzelteile und Baugruppen	Module, Dateien
Relationen	Komposition	Abhängigkeit, Dateihierarchie
Modelle vs. Instanzen	Konzept des Auftretens und der Anzahl	Keine duplizierten Komponenten
Standard	Standardisierte Produktmodelle	Kein Standard
Versioniertes Datenmodell	In EXPRESS nicht vorhanden	Enthalten
Historisches Versionieren	Revisionen	Revisionen
Logisches Versionieren	Alternative, Ersatz Option	Variante oder Zweig
Domänenbezogenes Versionieren	Sichten	−
Wahl einer Konfiguration	Sicht, Revision Optionen, Varianten	Basierend auf Attributen
Basis	Datenbank	Dateisystem
Prozessmodell	Änderungen und Genehmigungen	Werkzeugspezifisch

Tabelle 3.2: Vergleich von PDM und SCM nach [19]

Kapitel 4

Das Software-Konfigurationsmanagementsystem von Lin

4.1 Klassisches Software-Konfigurationsmanagement

Die klassischen Werkzeuge zum Software-Konfigurationsmanagement stellen wie beschrieben ihre Funktionalität auf der Basis von Dateien und Verzeichnissen zur Verfügung. Die Systemarchitekten und Softwareentwickler entwerfen und strukturieren ihr Konzept in Klassen, Funktionen und Schnittstellen.

4.2 Das Modell von Lin

Lin entwickelt in [39] ein SCM-System, das auf Funktionen und Klassen des Quelltextes operiert. Um dies zu erreichen werden die Funktionen und Klassen eines Softwareprodukts als Objekte repräsentiert. Eine Menge an Verweisen repräsentieren die Beziehungen der Objekte untereinander. Jedem Objekt sind die Methoden zum Übersetzen des Quellcodes und zum Verwalten seiner verschiedenen Versionen zugeordnet.

4.2.1 Grundlegende Konzepte

Das Modell von Lin besitzt vier Grundkonzepte. Dies sind die *Software Units*, die *Subsystems*, die *Workareas* und die *Classes of Software Units*.

- Eine Software Unit repräsentiert eine Funktion oder eine Klasse.
- Ein Subsystem repräsentiert diejenigen Software Units, die über Verweise von einer spezifizierten Software Unit aus erreicht werden können.

- Eine Workarea enthält die Software Units, an denen ein Programmierer arbeitet.

- Eine Class of Software Units definiert das Verhalten einer bestimmten Art der Software Unit.

Software Units

Ein SCM-System nach Lin verwaltet Software Units, die untereinander über Verweise verknüpft sind. Eine Software Unit enthält, wie in Abbildung 4.1 dargestellt, eine Menge von Datenattributen, eine Menge von möglichen Operationen und eine Menge von Verweisen. Die Datenattribute sind nicht direkt, sondern nur über entsprechende Operationen modifizierbar. Diese Operationen können von anderen Software Units oder von den Programmierern ausgeführt werden. Verweise werden denjenigen Software Units zugeordnet, von denen aus sie auf andere Software Units verweisen.

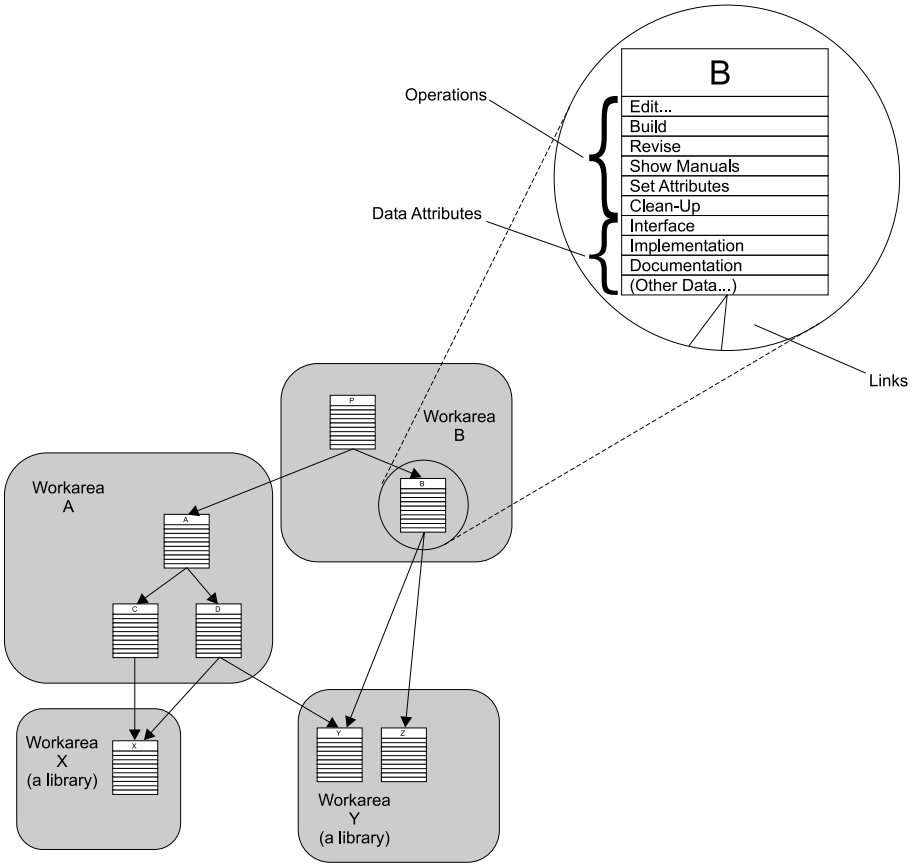

Abbildung 4.1: Modell von Lin aus [39]

Subsystems

Ein Subsystem $S(X)$ einer Software Unit X ist die Menge der Software Units, die über Verweise von X aus erreicht werden können. Lin nennt X die *Root Software Unit* von $S(X)$. Die Subsystems können sich überlappen.

Workareas

Software Units werden, wie in Abbildung 4.1 dargestellt, in Workareas eingeteilt. Workareas können sich nicht überlappen. Zu jeder Workarea gehören eine Menge an Software Units. Umgekehrt gehört eine Software Unit zu genau einer Workarea. Jeder Workarea ist genau ein Softwareentwickler zugeordnet, nur dieser kann die Software Units in seiner Workarea bearbeiten. Jede Workarea enthält veränderliche Software Units, die sich noch in Bearbeitung befinden. Diese werden *Active Versions* genannt. Weiterhin enthält sie Software Units, die sich nicht mehr verändern. Diese unveränderlichen Versionen werden *Fixed Versions* genannt.

Classes of Software Units

Jede Software Unit besitzt alle notwendigen Daten und Methoden, um das entsprechende Programm zum Bearbeiten der Datenattribute oder zum Übersetzen des Quellcodes zu starten. Damit nicht für jede neue Software Unit diese Daten neu eingeben werden müssen, können Schablonen benutzt werden. Darüber hinaus ist es möglich, Übereinstimmungen innerhalb der Schablonen durch Hierarchien abzubilden.

4.2.2 Systemaufbau

Der Systemaufbau eines Programms, dessen Root Software Unit X ist, wird durch Anwendung der Build-Operation auf X initiiert. Von X ausgehend wird dann eine *Build Message* an alle Software Units gesendet, auf die von X aus verwiesen wird. Die Methode des Durchlaufens aller Software Units ist eine Tiefensuche auf dem Graph[1], der durch die Software Units und ihre Verweise dargestellt wird.

Jede Software Unit prüft, ob sie die Build Message an Nachfolger weitergeben muss. Weiterhin wird eine Übersetzung nur durchgeführt, wenn eine existierende Übersetzung nicht mehr aktuell ist.

4.2.3 Versionskontrolle

Für die Versionskontrolle werden die Software Units in *Fixed Versions* und *Active Versions* unterschieden. Eine Fixed Version ist eine Version, die nicht mehr verändert

[1]Die grundlegenden Algorithmen der Tiefen- und der Breitensuche in Graphen können dem Anhang entnommen werden.

wird. Eine Active Version ist eine Version, an der gerade gearbeitet wird.

Eine Active Version bietet eine *Snapshot Operation* an, die eine Fixed Version als ihren Vorgänger erzeugt. Eine Fixed Version bietet eine *Revise Operation* an, die eine Active Version als ihren Nachfolger erzeugt. Wird eine neue Software Unit erzeugt, so wird diese als Active Version erzeugt. Wiederholtes Snapshot und Revise erzeugt für eine Software Unit einen *History Tree*, wie in Abbildung 4.2 dargestellt.

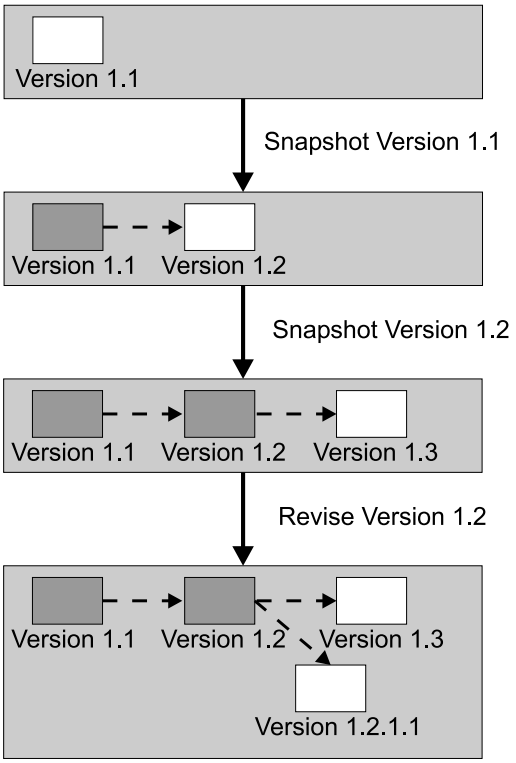

Abbildung 4.2: Versionskontrolle einer einzelnen Software Unit aus [39]

Wird eine Snapshot- oder Revise-Operation auf eine Software Unit X angewendet, so beeinflusst diese das gesamte Subsystem $S(X)$. Eine Snapshot Operation auf X erzeugt für alle Software Units von $S(X)$ Fixed Versions und verbindet sie untereinander in der gleichen Weise wie in der Ursprungsversion.

Die Funktionsweise der Operationen Snapshot und Revise zeigt das folgende Beispiel aus [39].

Abbildung 4.3 zeigt das Subsystem $S(A)$ im Ausgangszustand. Alle Software Units von $S(A)$ sind Fixed Versions.

Die Operation *Revise* A erzeugt eine Active Version von A und damit auch Active Versions aller Nachkommen von A (Abbildung 4.4). Danach besteht weiterhin $S(A)$ mit den Software Units als Fixed Versions. Die Active Version von A ist A'.

Die Software Units A und Y werden nun durch Anwendung von Edit-Operationen bearbeitet (Abbildung 4.5).

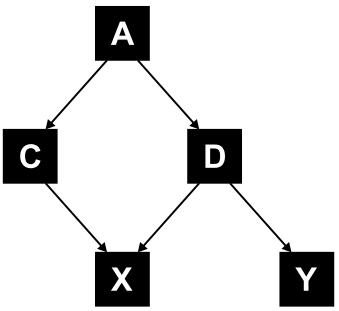

Abbildung 4.3: Subsystem *A*

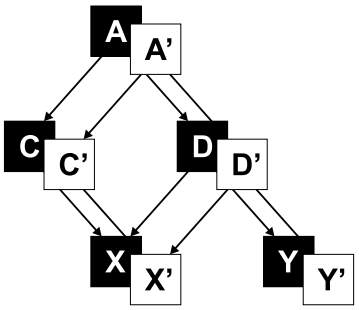

Abbildung 4.4: Subsystem *A* nach der Operation *Revise A*

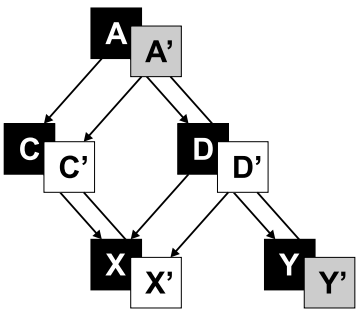

Abbildung 4.5: Subsystem *A* nach den Operationen *Edit A* und *Edit Y*

Anschliessend wird die Snapshot Operation von A aufgerufen. C' und X' wurden nicht verändert, diese Versionen können entfernt werden. Der Verweis von A' nach C' kann in einen Verweis von A' nach C geändert werden. Ebenso wird der Verweis von D' nach X' zu einem Verweis von D' nach X. D' kann nicht entfernt werden, weil der Nachkomme Y' verändert wurde (Abbildung 4.6).

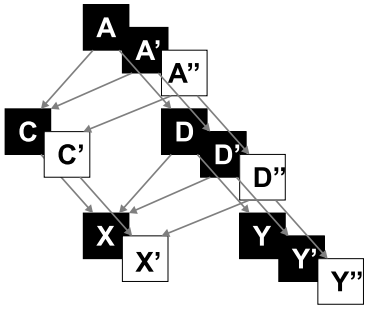

Abbildung 4.6: Subsystem A nach der Operation *Snapshot* A'

4.2.4 Wiederverwendung von Softwarekomponenten

Zur Verwendung einer Komponente genügt ein Verweis auf die Software Unit, die diese Komponente repräsentiert. Dadurch stehen dem Softwareentwickler alle verfügbaren Informationen über diese Komponente zur Verfügung.

4.2.5 Kooperatives Programmieren

Die Strategie bei der Unterstützung des Kooperativen Programmierens ist es, dass ein Programmierer stets eine ältere und daher stabilere Version einer Komponente eines anderen Programmierers verwendet und dabei an neueren Versionen seiner eigenen Komponenten arbeitet. Zur Verwendung einer Komponente eines anderen Programmierers genügt ein Verweis auf die entsprechende Software Unit. Die Verwendung einer fremden Komponente entspricht daher der Wiederverwendung einer Komponente.

Zur Vermeidung von gegenseitigen Störungen bei der Kooperativen Softwareentwicklung können Workspaces genutzt werden. Wird nur auf Fixed Software Units innerhalb fremder Workspaces verwiesen, können die Arbeiten innerhalb dieser Workspaces die eigenen Arbeiten nicht beeinflussen.

4.3 Eigenschaften des SCM-Systems

Das System von Lin besitzt die Möglichkeit der

- Versionierung von Komponenten,

- Definition und Verwaltung beliebig vieler Konfigurationen und
- Bearbeitung der Komponenten durch räumlich und zeitlich verteilte Planungsteams durch die Definition von Workspaces.

Dabei berücksichtigt das System, dass die

- Komponenten einer Konfiguration Elemente eines gerichteten, azyklischen Graphen sind.

Unter Berücksichtigung der vorgesehenen Verwendung zur Unterstützung von Planungsprojekten existieren zwei Beschränkungen.

- Das System von Lin unterscheidet als SCM-System keine mehrfachen Instanzen einer Komponente.
- Es existiert keine Möglichkeit der Definition von Gültigkeitsregeln und ihrer programmautomatischen Prüfung.

Kapitel 5

Der Produktkonfigurator von Hedin, Ohlsson und McKenna

5.1 Allgemeines

Komplexe Produkte werden oft als Produktfamilien entworfen, bei denen jedes individuelle Produkt eine Konfiguration von zusammenhängenden Komponenten darstellt. Ein Produktkonfigurator ist ein Werkzeug zur Unterstützung des Konfigurationsprozesses. Er stellt sicher, dass alle Entwurfs- und Konfigurationsregeln, die im Produktkonfigurationsmodell enthalten sind, erfüllt werden. Weiterhin ist der Produktkonfigurator in der Lage, über alle gültigen Konfigurationen dieses Produkts Auskunft zu geben.

5.2 Anforderungen an einen Produktkonfigurator

Im Idealfall ist ein Produktkonfigurator derart entworfen, dass das Produktmodell von den Ingenieuren des Fachgebiets gelesen, erstellt und bearbeitet werden kann, ohne auf Programmierexperten zurückgreifen zu müssen.

Hedin et al. beschreiben in [29] einen Produktkonfigurator, der auf Objektorientierung und einer Attributengrammatik basiert. Dieser Produktkonfigurator wird zur Zusammenstellung von benutzerspezifisch konfigurierbaren Produkten eingesetzt, die aus serienproduzierten Einzelteilen bestehen. Dabei ermöglicht das System die Definition von Regeln.

Hierfür haben die Autoren OPG[1], eine objektorientierte Beschreibungssprache, entwickelt.

Die Unterstützung der Objektorientierung ermöglicht eine angemessene Modellierungsunterstützung, und die Attributengrammatik ermöglicht die Definition von Konfigurationsregeln.

[1] Object-oriented Product Grammar

5.3 Das Modell zur Produktkonfiguration

Hedin et al. erläutern ihr Modell zur Produktkonfiguration am Beispiel eines Plattenwärmetauschers. Derartige Wärmetauscher (plate heat exchanger) bestehen aus einem Plattenpaket (plate package) und einem Rahmen (frame).

Abbildung 5.1 zeigt eine Komponentenhierarchie für einen Plattenwärmetauscher aus [29]. Bemerkenswert ist hierbei, dass sowohl

- Konfigurationskomponenten (Configuration Components) als auch
- Prototypkomponenten (Prototypical Components)

modelliert werden. Komponenten mit einer physischen Entsprechung durch ein in Serie produziertes Produkt sind Prototypkomponenten. Konfigurationskomponenten besitzen keine physische Entsprechung.

Eine Produktkonfiguration wird als Menge von miteinander verbundenen Komponenten betrachtet. Jede dieser Komponenten besitzt eine Anzahl an Eigenschaften. Regeln über die Komponenten und ihre Eigenschaften definieren die Gültigkeit einer Konfiguration. Eine Konfiguration ist genau dann gültig, wenn alle definierten Regeln erfüllt sind.

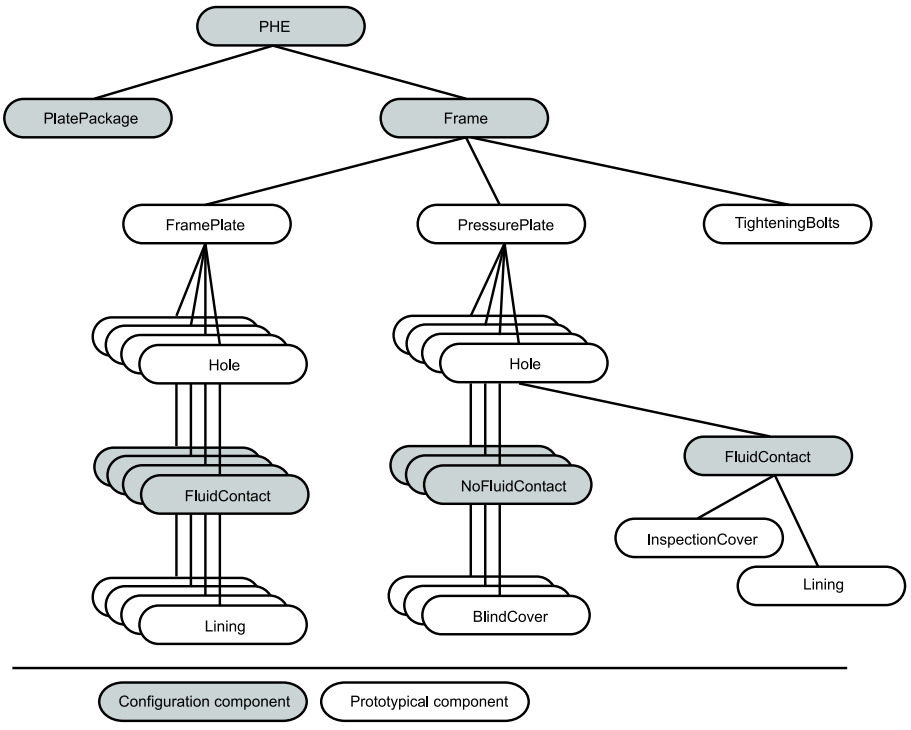

Abbildung 5.1: Komponentenhierarchie aus [29]

Das Modell von Hedin et al. untergliedert die Produktkonfiguration in drei verschiedene Bereiche.

Type Level: Im Type Level werden die grundlegenden Komponenten eines Produkts definiert.

Prototype Level: Im Prototype Level werden mögliche Ausprägungen der Komponenten des Type Level definiert.

Configuration Level: Im Configuration Level werden auf der Grundlage der Festlegungen des Type Level und des Prototype Level kunden- beziehungsweise auftragsbezogene Konfigurationen erstellt.

5.3.1 Type Level

Im Type Level werden die Komponenten definiert, die in den Produktkonfigurationen enthalten sind. Die Komponenten sind in einer Hierarchie von allgemeinen Komponenten hin zu spezifischen Komponenten angeordnet. Jede dieser Komponenten beinhaltet bereits Deklarationen und Regeln.

Abbildung 5.2 zeigt ein Beispiel für die Definitionen im Type Level. Es zeigt, dass die Komponente *FramePlate* vier *Hole*-Komponenten als Unterkomponente besitzt. In Abhängigkeit von der Konfigurationskomponente *HoleContact* mit ihren Untertypen *FluidContact* und *NoFluidContact* existieren *Lining*-Komponenten als Unterkomponenten von den *Hole*-Komponenten.

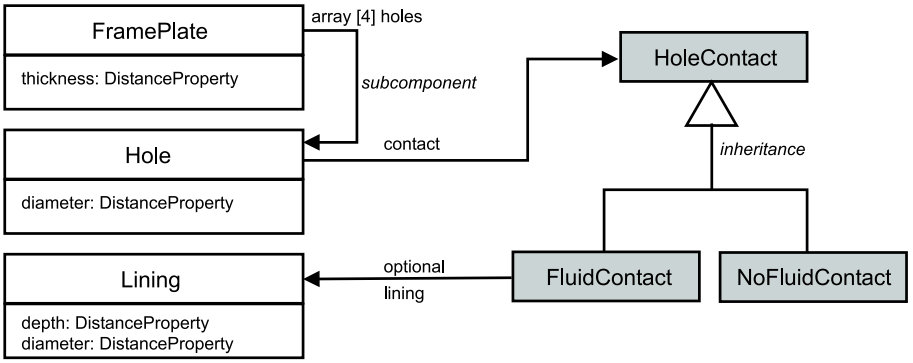

Abbildung 5.2: Beispiel für den Type Level aus [29]

5.3.2 Prototype Level

Im Prototype Level werden die möglichen Instanzen von Prototypkomponenten des Type Level definiert. Dadurch wird festgelegt, welche möglichen physischen Ausprägungen eine Prototypkomponente haben kann. Jeder mögliche Prototyp besitzt Werte für einige oder alle seine Eigenschaften. Die Verwendung von Prototypen ist nach [29] eine direkte Anwendung des Erzeugungsmusters Prototyp aus [24]. Abbildung 5.3 zeigt die Prototypen $FP1$ und $FP2$ für die Prototypkomponente *FramePlate* und die Prototypen $L1$ und $L2$ für die Prototypkomponente *Lining*.

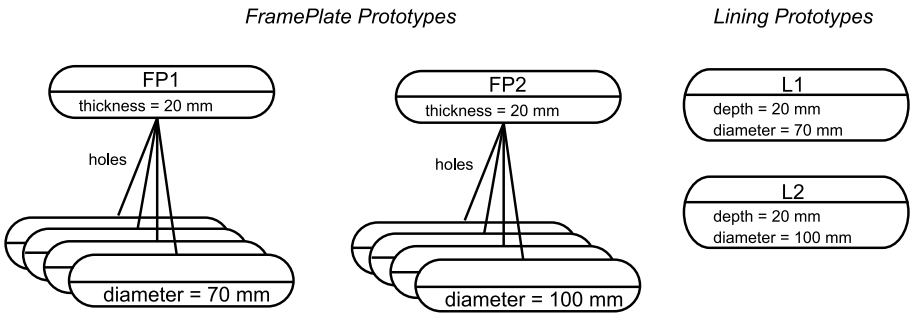

Abbildung 5.3: Beispiel für den Prototype Level aus [29]

5.3.3 Configuration Level

Im Configuration Level wird eine kunden- beziehungsweise auftragsbezogene Konfiguration modelliert. In diesem Level nehmen alle Eigenschaften der Prototypkomponenten Werte an. Eine kunden- beziehungsweise auftragsbezogene Konfiguration zeigt die Abbildung 5.4.

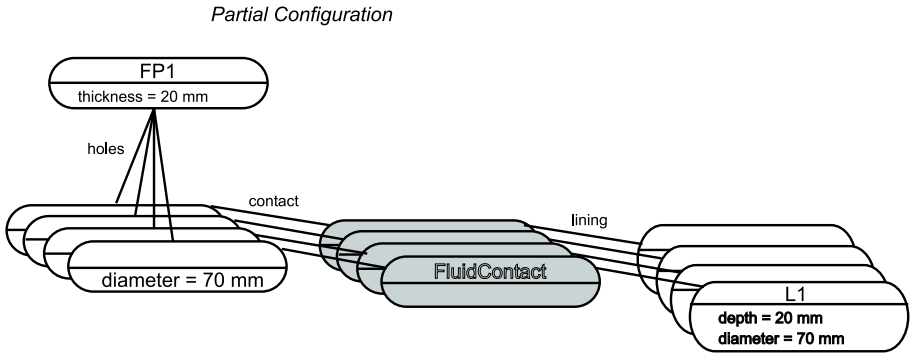

Abbildung 5.4: Beispiel für den Configuration Level aus [29]

5.4 Die Definition von Konfigurationsregeln mit OPG

Die objektorientierte Beschreibungssprache OPG kennt zwei Typen von Regeln.

Definitionsregeln: Eine Definitionsregel definiert den Wert einer abgeleiteten Eigenschaft.

Gültigkeitsregeln: Eine Gültigkeitsregel ist ein boolescher Ausdruck über Eigenschaften. Eine Konfiguration ist gültig, wenn alle Gültigkeitsregeln, die in der Konfiguration definiert sind, erfüllt werden.

Die Abbildung 5.5 zeigt, wie Regeln in OPG definiert werden. Ein Rahmen (Frame) wird im Modell als ein Bauteil definiert, das aus einer Rahmenplatte (FramePlate) und Bolzen (TighteningBolts) besteht. Die Regeln in OPG sind direkt Bestandteil der modellierten Komponenten. Die Rahmenplatte enthält eine Definitionsregel über die Anzahl an Bohrungen für die Bolzen. Die Bolzen werden in einer Komponente zusammengefasst, die mit einer Definitionsregel die Anzahl an verwendeten Bolzen festlegt. Für den Rahmen gilt folgende Gültigkeitsregel: Die Anzahl an Bohrungen für die Bolzen muss mit der Anzahl an Bolzen übereinstimmen.

Diese Gültigkeitsregel kann programmautomatisch überprüft werden.

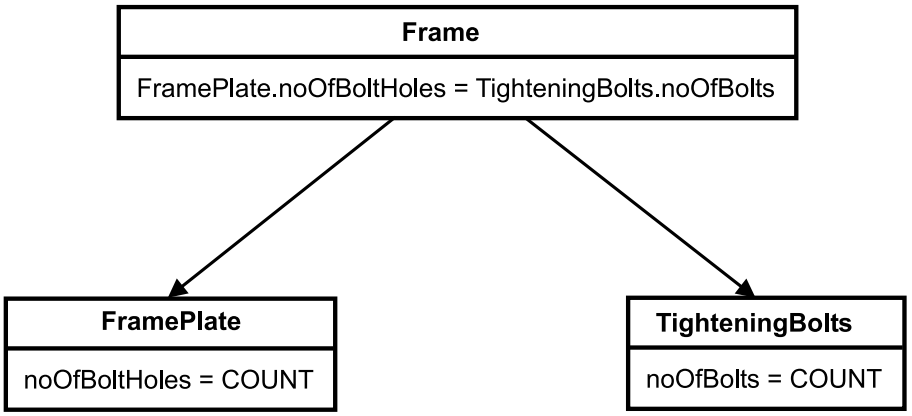

Abbildung 5.5: Einfaches Beispiel aus [29]

In OPG gibt es zwei Arten des Zugriffs auf Eigenschaften von Komponenten, um diese in Regeln zu verwenden.

Referenzierung von Unterkomponenten: Eine Regel kann auf die Eigenschaften von direkten Unterkomponenten zugreifen. Diese Form des Zugriffs wird im Beispiel der Abbildung 5.5 verwendet.

Referenzierung der einschliessenden Komponente: Eine Regel kann auf die Eigenschaften der sie einschliessenden Komponente zugreifen.

5.5 Eigenschaften des Produktkonfigurators

Der vorgestellte Produktkonfigurator ermöglicht die

- Definition einer Produktzusammensetzung und die
- Erstellung einer benutzer- beziehungsweise auftragsbezogenen Produktkonfiguration.

Weiterhin unterstützt der beschriebene Produktkonfigurator die

- Defintion von Gültigkeitsregeln und die
- programmautomatische Überprüfung der Korrektheit einer Konfiguration.

Hierbei wird von dem Produktkonfigurator überprüft, ob alle im Produktmodell definierten Gültigkeitsregeln erfüllt werden.

Eine Unterstützung des Konfigurationsprozesses durch Anzeigen der jeweils gültigen Wahlmöglichkeiten ist nicht umgesetzt.

Der Produktkonfigurator unterstützt ausschliesslich Konfigurationen, die durch eine Baumstruktur abgebildet werden können. Zur leistungsfähigeren Definition von Regeln müssen die Gültigkeitsregeln mit beliebigen Komponenten verknüpft werden können. Die wandelt die Baumstruktur in einen Graphen, der im System verwaltet werden muss.

Produktfamilien bleiben nicht konstant, sie entwickeln sich im zeitlichen Verlauf. Die dadurch entstehenden unterschiedlichen Versionen von Komponenten und ihre Beziehungen untereinander können nicht abgebildet werden. Nach [29] muss das System hierfür um die Funktionalitäten eines Revisionskontrollsystems – oder Konfigurationsmanagementsystems – erweitert werden. Weiterhin beinhaltet der Produktkonfigurator keine Aufteilung der Komponenten in getrennte Arbeitsbereiche, die unterschiedlichen Bearbeitern zugeordnet sind.

Zusammenfassend existieren drei wesentliche Beschränkungen in dem beschriebenen System.

- Keine Unterstützung von Graphen.
- Keine Unterstützung von Versionierung.
- Keine Unterstützung von räumlich- und zeitlich getrennten Planungsteams.

Kapitel 6

Neues Konzept für ein Planungssystem

6.1 Allgemeines

Das Ziel der Arbeit ist es, die Methoden des Konfigurationsmanagements im Rahmen der Durchführung von Bauplanungsprojekten einzusetzen. Dies ist möglich, weil Bauwerke ebenso wie andere komplexe Produkte aus einzelnen, in Serie produzierten Bauteilen zusammengesetzt sind. Zur Unterstützung wird ein neues Planungssystem entwickelt.

Das Planungssystem soll

- Produktfamilien verwalten,
- benutzer- beziehungsweise auftragsspezifische Konfigurationen verwalten,
- mehrbenutzerfähig sein und
- programmautomatisch die Korrektheit der spezifischen Konfigurationen überprüfen.

Das neue Konzept für ein Planungssystem umfasst zwei Bestandteile,

- Versionierung und
- Produktmodell und Produktkonfiguration.

Das in dieser Arbeit vorgestellte Planungssystem verwendet ein Verfahren zur Versionierung, bei dem die Komponenten des Produktmodells versioniert werden können. Weiterhin können Konfigurationen bestehend aus bestimmten Versionen von Komponenten verwaltet werden.

Das Produktmodell des Planungssystem erlaubt die Konfiguration benutzer- beziehungsweise auftragsspezifischer Produkte. Die Gültigkeit einer spezifischen Konfiguration kann anhand der im Produktmodell definierten Regeln programmautomatisch überprüft werden.

Wie in Kapitel 11 beschrieben wird, können Programme zur Produktplanung derart entwickelt werden, dass die Regeln der Produktzusammensetzung direkt im Quellcode des Programms umgesetzt sind.

Damit ergibt sich eine Situation entsprechend Abbildung 6.1.

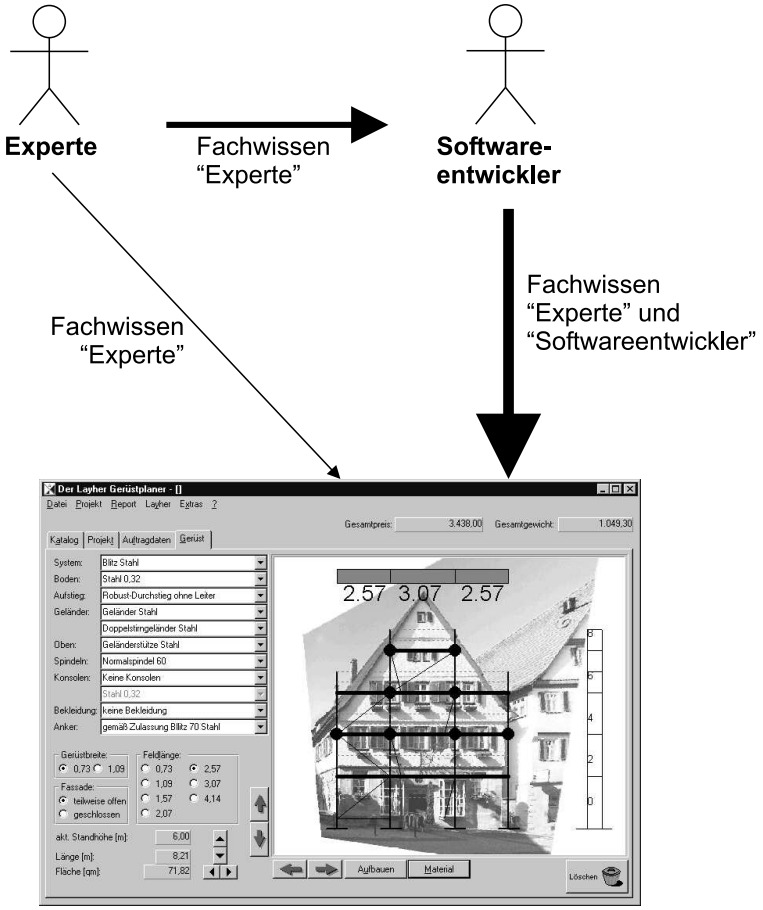

Abbildung 6.1: Integration von Fachwissen durch den Softwareentwickler

Die Experten eines Anwendungsgebiets müssen zunächst ihr umfangreiches Fachwissen an die Experten zur Softwareerstellung weitervermitteln. Diese erstellen dann ein Computerprogramm zur Produktkonfiguration.

Eine direkte Integration von Fachwissen durch die Experten in das Computerprogramm findet kaum statt.

Daraus ergeben sich die im Abschnitt 11.4 beschriebenen Nachteile.

- Der Programmierer muss zusätzlich zu seinem umfangreichen Fachwissen alle spezifischen Regeln des Anwendungsbereichs kennen.

- Bei Veränderungen an den Regeln oder der Produktzusammensetzung muss der Quellcode des Programms durch einen Programmierer angepasst und neu übersetzt werden.

Das neue Konzept für ein Planungssystem sieht vor, dass die Experten eines Fachgebiets die Produktkomponenten, ihre Beziehungen untereinander und die Gültigkeitsregeln selbst und ohne Programmierkenntnisse in das Planungssystem eingeben können. Ein weiterer Bestandteil dieses Planungssystem dient der Konfiguration auftrags- beziehungsweise kundenbezogener Produkte.

Abbildung 6.2 zeigt die neue Situation. Die Experten des Anwendungsgebiets definieren das Produktmodell und relevante Regeln in einem Programmteil. Der Softwareentwickler erstellt ein Programmteil, welches unter Verwendung des eingegebenen Produktmodells eine Produktkonfiguration erstellen kann. Seine Arbeit umfasst im Wesentlichen die Entwicklung einer geeigneten Benutzeroberfläche. Hierfür benötigt er zusätzich zu seinem eigenen Fachwissen nur noch ein Basiswissen über das Anwendungsgebiet.

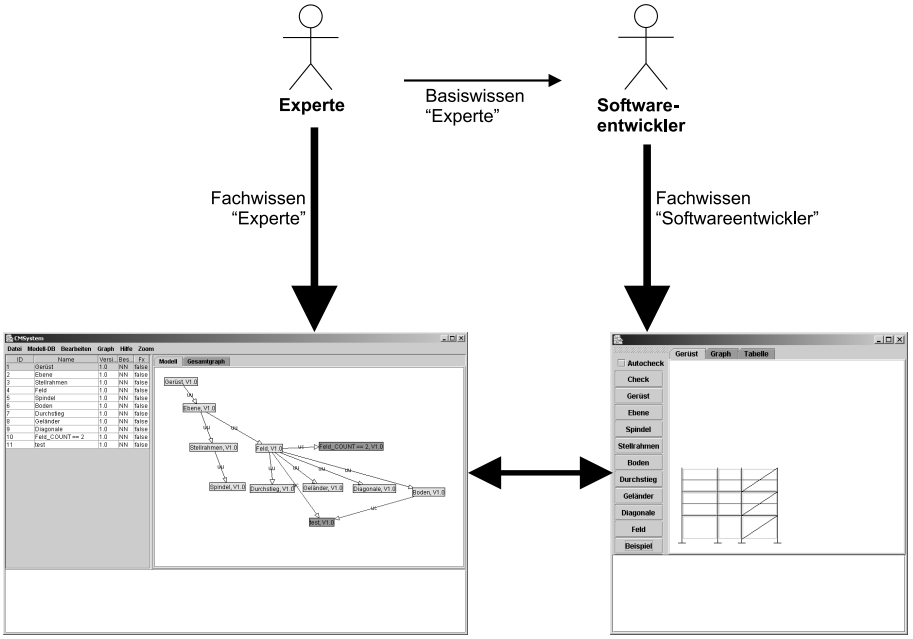

Abbildung 6.2: Integration von Fachwissen durch den Experten des Anwendungsgebiets

6.2 Das System von Lin als Basis für die Versionierung

Das entwickelte Planungssystem basiert auf dem Software-Konfigurationsmanagementsystem von Lin [39], von dem die Funktionalitäten für die

- versionsgestütze Verwaltung von Komponenten,
- Definition von Konfigurationen und
- Organisation der Komponenten in benutzerspezifische Arbeitsbereiche

übernommen werden.

In dem System von Lin [39] sind die *Software Units* eine grundlegende Komponente. In der vorliegenden Arbeit werden zum einen *Units* zur Abbildung von Prototypkomponenten und Konfigurationskomponenten und *Constraints* zur Abbildung von Gültigkeitsregeln als grundlegende Komponenten verwendet. Somit ist das hier beschriebene System zur Versionierung eine Erweiterung des Systems aus [39]. Die dort beschriebenen Operationen können übernommen werden. Notwendig sind Anpassungen, damit die Datenhaltung nicht nur wie in [39] einen Komponententyp – die Software Units – sondern zwei Komponententypen – Units und Constraints – verwalten kann.

6.3 Das System von Hedin et al. als Basis für die Produktkonfiguration

Die Produktkonfiguration des neuen Planungssystems orientiert sich an dem in Hedin et al. in [29] vorgestellten Modell für einen Produktkonfigurator, von dem die Funktionalitäten für die

- Definition einer Produktzusammensetzung,
- Erstellung benutzer- beziehungsweise auftragsbezogener Produktkonfigurationen und
- Definition und programmautomatische Überprüfung von Gültigkeitsregeln

übernommen werden. Dieses Modell wird erweitert, damit eine flexiblere Verwendung von Gültigkeitsregeln möglich ist. Dies wird erreicht durch die Abbildung der Gültigkeitsregeln als eigenständige Objekte im Modell.

6.4 Vorteile der neuen Lösung

Die Produktzusammensetzung und die relevanten Regeln können ohne Programmierkenntnisse in einem Computerprogramm eingegeben werden. Die hierfür notwendigen Grundlagen sind Kenntnisse über die Modellierung einer Produktzusammensetzung in Form von *Besteht-aus-Beziehungen.*

Ändern sich die Gültigkeitsregeln oder die Produktzusammensetzung, so können diese Veränderungen ebenfalls ohne Programmierkenntnisse in einem Computerprogramm vorgenommen werden. Kundenspezifische Konfigurationen auf der Basis des neuen Modells sind sofort ohne ein Übersetzen des Programms möglich.

Die Möglichkeit der Verwaltung zahlreicher Produktmodelle und zahlreicher spezifischer Konfigurationen eröffnet Möglichkeiten zur Durchführung von *Was-wäre-wenn-Analysen*. So können beispielsweise für einen Auftrag auf der Grundlage mehrerer unterschiedlicher Produktmodelle Planungen durchgeführt werden und die am besten geeignete Version weiterverwendet werden.

Die Abbildung 6.3 zeigt, wie auf der Grundlage der beschriebenen Bestandteile eine Planungssystem-Software aufgebaut ist.

Die Planungssystem-Software besteht aus den folgenden Komponenten.

modelDB: Eine Datenhaltung zur versionierten Verwaltung der Produktzusammensetzung und der zugehörigen Regeln.

instanceDB: Eine Datenhaltung zur versionierten Verwaltung der benutzer- oder auftragsspezifischen Konfigurationen. Diese sind auf der Grundlage der Definitionen durchgeführt, die im *modelDB* gespeichert sind.

modelDBController: Eine Komponente zur Steuerung der Datenhaltung für das Produktmodell.

instanceDBController: Eine Komponente zur Steuerung der Datenhaltung für die benutzer- beziehungsweise auftragsspezifischen Konfigurationen.

Admin-UI: Eine Benutzerschnittstelle zur Eingabe und Bearbeitung von Produktmodellen. Weiterhin können die Produktmodelle versioniert verwaltet werden.

Client-UI: Eine fachspezifische Benutzeroberfläche, mit der auftrags- beziehungsweise benutzerspezifische Konfigurationen erstellt und versionsgestützt verwaltet werden können.

Hierbei ist ersichtlich, dass lediglich ein Programmteil speziell für den Anwendungsbereich erstellt werden muss. Dieses ist die Benutzeroberfläche für Anwender, welche spezifische Konfigurationen erstellen.

Die einzelnen Systemkomponenten kommunizieren über definierte Schnittstellen miteinander. Abbildung 6.4 zeigt wesentliche Operationen der Schnittstelle *CMCore-Interface*. Über diese Schnittstelle werden die Datenhaltungen zur versionierten Verwaltung der Konfigurationen gesteuert. Weiterhin steht über diese Schnittstelle die Operation zur Evaluierung der Gültigkeitsregeln zur Verfügung.

Die Schnittstelle *Configuration-Interface* stellt die Operationen zur Produktkonfiguration zur Verfügung. Wesentlich sind hierbei die in Abbildung 6.5 genannte

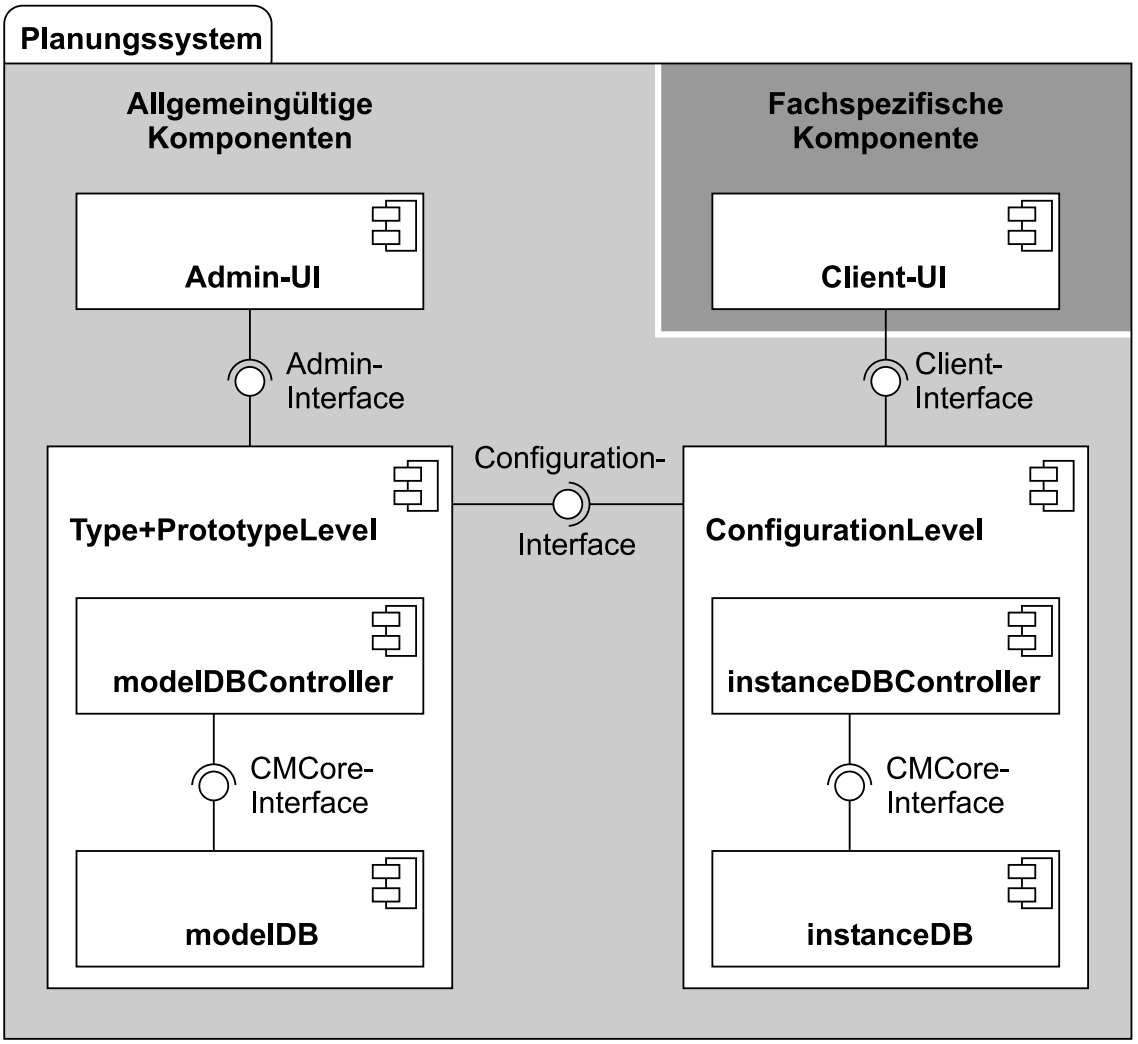

Abbildung 6.3: Struktur einer Software nach dem neuen Konzept

Operation *Add()* zum schrittweisen Aufbau einer auftrags- beziehungsweise benutzerspezifischen Produktkonfiguration und die Operation *ConnectConstraints()* zum programmautomatischen Einbinden von Gültigkeitsregeln.

Die Schnittstellen *Admin-Interface* und *Client-Interface* dienen der Kommunikation der Benutzerschnittstellen *Admin-UI* beziehungsweise *Client-UI* mit den entsprechenden Komponenten.

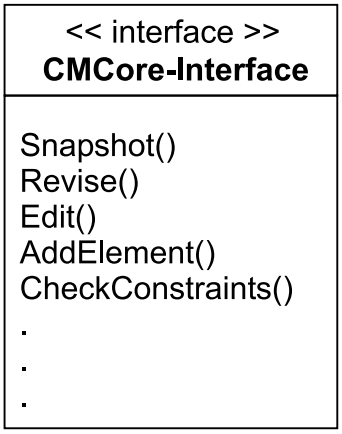

Abbildung 6.4: CMCore-Interface

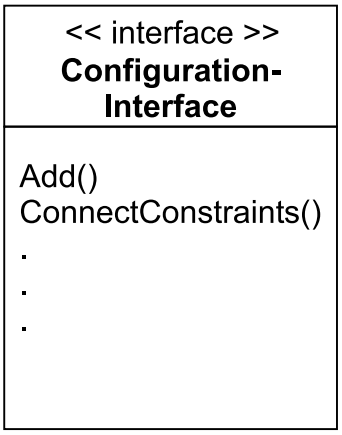

Abbildung 6.5: Configuration-Interface

Kapitel 7

Die Versionierung des neuen Konzepts

7.1 Allgemeines

Wie in Kapitel 6 dargelegt, umfasst das neue Konzept die zwei Bestandteile

- Versionierung und
- Produktmodell und Produktkonfiguration.

In diesem Kapitel werden die grundlegenden Elemente und die Methoden vorgestellt. Auf der Basis dieser Elemente und Methoden erfolgt die versionierte Verwaltung von allgemeinen als auch von auftrags- oder benutzerspezifischen Produktmodellen und der zugehörigen Regeln. Somit bildet das vorgestellte System zur Versionierung die Grundlage für darauf aufbauende Programmstrukturen wie beispielsweise die Methoden zur Produktkonfiguration, welche im Kapitel 8 beschrieben werden.

7.2 Grundlegende Elemente

7.2.1 Units

Die *Units* repräsentieren Bestandteile eines Produkts. Hierbei können diese sowohl Prototypkomponenten als auch Konfigurationskomponenten repräsentieren.

Die Units enthalten eine Menge an Attributen, Operationen und Links.

Eine Unit enhält die Attribute (ID, $NAME$, $DESCR$, AL, NUU, NUC, [NUP][1], VP, FX). Die Tabelle 7.1 führt die verwendeten Attribute und ihre Bedeutung auf.

[1]Das Attribut NUP ist optional. Die nachfolgenden Abschnitte erläutern, unter welchen Bedingungen dieses Attribut notwendig ist.

ID	Eindeutiger Bezeichner.
$NAME$	Name des Elements.
$DESCR$	Textuelle Beschreibung der Unit.
AL	Attributliste, in der die produktmodellspezifischen Daten gespeichert werden.
NUU	Menge an IDs von Units, auf die die aktuelle Unit verweist.
NUC	Menge an IDs von Constraints, die mit dieser Unit verknüpft sind.
NUP	Optionales Attribut. Menge an IDs von Units, die eine physische Ausprägung dieser Unit repräsentieren.
VP	Zeiger auf den Versionsvorgänger.
FX	Fixed Flag, ein boolescher Wert, der TRUE ist, falls die Unit eine unveränderliche Unit ist.

Tabelle 7.1: Attribute der Units

Abbildung 7.1 stellt die Units als Klassen in der Modellierungssprache UML dar. Die Units implementieren die in Abbildung 7.2 dargestellte Schnittstelle *CMElement*. Die Schnittstelle stellt grundlegende Operationen zur Verfügung, die in Abbildung 7.2 ohne Angabe von Rückgabewerten und Übergabeparametern dargestellt sind. Eine detaillierte Darstellung der Operationen erfolgt in Abschnitt 7.7.

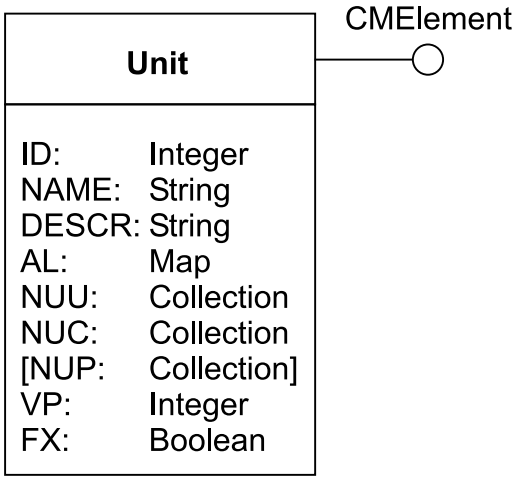

Abbildung 7.1: Klasse Unit in UML-Darstellung

Links zur Repräsentation von Beziehungen

Es existieren drei grundlegende Arten von Links.

Uses Unit-Links bilden Verweise auf andere Units ab.

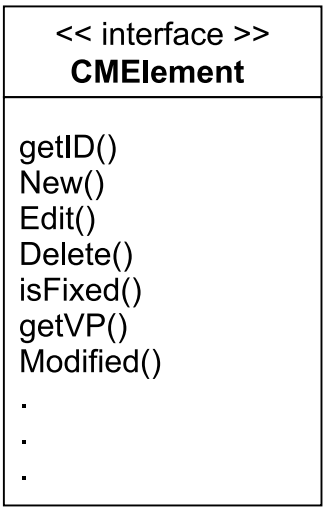

Abbildung 7.2: Schnittstelle CMElement in UML-Darstellung

Uses Constraint-Links zeigen auf Constraint-Elemente. Mit einem Constraint-Link wird angezeigt, dass die Attribute der verweisenden Unit zur Evaluierung der Bedingung des referenzierten Constraint-Elements notwendig sind.

Uses Prototype-Links bilden Verweise auf Prototype-Units.

Die Abbildung 7.3 zeigt die Units a, b, c, d, e, f, g, h und die Constraints x, y, z. Die Uses Unit- und Uses Constraint-Links sind als Pfeile dargestellt und entsprechend gekennzeichnet.

Uses Prototype-Links werden in Abhängigkeit von der Wahl der Abbildung des *Prototype Level* benötigt. Wie in Abschnitt 8.2.2 von Kapitel 8 beschrieben wird, existieren zwei unterschiedliche Möglichkeiten zur Abbildung des *Prototype Level*.

1. Prototype Level als eine vom Type Level getrennte Datenhaltung.

2. Prototype Level integriert in den Type Level.

Die Vor- und Nachteile der unterschiedlichen Möglichkeiten werden in Abschnitt 8.2.2 dargelegt. Eine Möglichkeit der Integration der Elemente des Prototype Level in den Type Level ergibt sich durch die Hinzunahme der Uses Prototype-Links. Diese Links verweisen von einer Unit zu keiner, einer oder beliebig vielen Units, die mögliche physische Entsprechungen repräsentieren. Die Erweiterung durch einen zusätzlichen Link-Typ ist ohne wesentliche Veränderungen der hier beschriebenen Methoden des Konfigurationsmanagements möglich. Auf die notwendigen Änderungen an den Algorithmen wird jeweils hingewiesen. Hierbei werden die optional erforderlichen Anweisungen und Fragmente in eckige Klammern gesetzt.

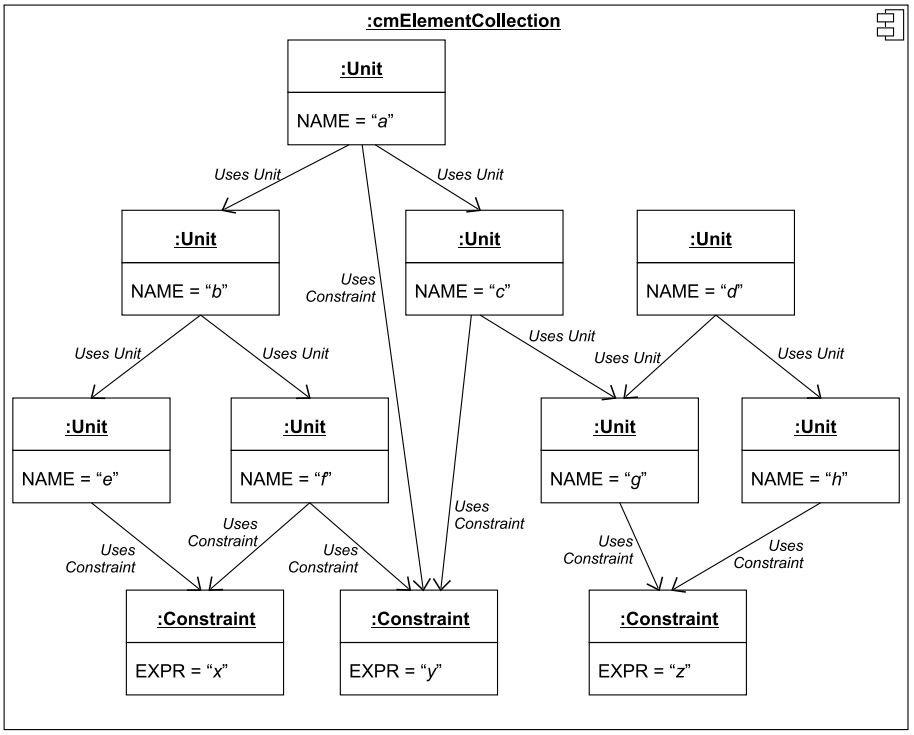

Abbildung 7.3: Elemente mit UsesUnit- und UsesConstraint-Links

Versionsvorgänger

Der Versionsvorgänger $y = \text{VPred}(D, x)$ einer Unit x in einer Elementsammlung D ist diejenige Unit für die

$$y.ID = x.VP$$

gilt.

7.2.2 Attribute

Die *Attribute* dienen der Speicherung der produktmodellspezifischen Attribute und ihrer Werte.

Ein Attribut ist ein Wertepaar ($NAME$, $VALUE$). Die Tabelle 7.2 führt die verwendeten Attribute und ihre Bedeutung auf.

$NAME$	Name des Attributs.
$VALUE$	Wert des Attributs.

Tabelle 7.2: Attribute zur Abbildung produktmodellspezifischer Daten

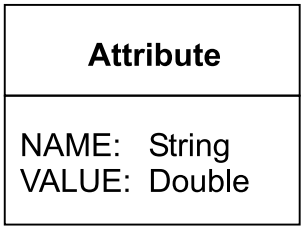

Abbildung 7.4: Klasse Attribute in UML-Darstellung

7.2.3 Constraints

Die *Constraints* bilden numerische Bedingungen ab, die programmautomatisch auf wahr oder falsch geprüft werden können.

Hierfür enthalten Constraints die Attribute (ID, $DESCR$, $EXPR$, VP, FX). Die Tabelle 7.3 führt die verwendeten Attribute und ihre Bedeutung auf.

ID	Eindeutiger Bezeichner.
$DESCR$	Textuelle Beschreibung der Komponente.
$EXPR$	Ausdruck zur Repräsentation einer Regel.
VP	Zeiger auf den Versionsvorgänger.
FX	Fixed Flag, ein boolescher Wert, der TRUE ist, falls der Constraint unveränderlich ist.

Tabelle 7.3: Attribute der Constraints

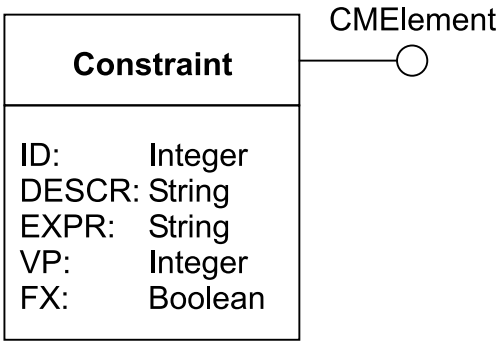

Abbildung 7.5: Klasse Constraint in UML-Darstellung

Versionsvorgänger

Der Versionsvorgänger y =VPred(D, x) eines Constraint x in einer Elementsammlung D ist derjenige Constraint für den

$$y.ID = x.VP$$

gilt.

7.3 Beziehungen der Elemente zueinander

Die Beziehungen der Elemente untereinander können mit den folgenden Aussagen beschrieben werden.

- Eine Unit verweist über Uses Unit-Links auf keine, eine oder beliebig viele weitere Units.
- Eine Unit verweist über Uses Constraint-Links auf keinen, einen oder beliebig viele Constraints.
- Werden Uses Prototype-Links verwendet, so verweist eine Unit auf keinen, einen oder beliebig viele Constraints.
- Units und Constraints verweisen jeweils über Version Predecessor-Links auf keinen oder genau einen Versionsvorgänger.
- Eine Unit enthält keine, eine oder beliebig viele Attribute. Attribute sind von der zugehörigen Unit existenzabhängig.

Abbildung 7.6 zeigt die Beziehungen der Elemente zueinander.

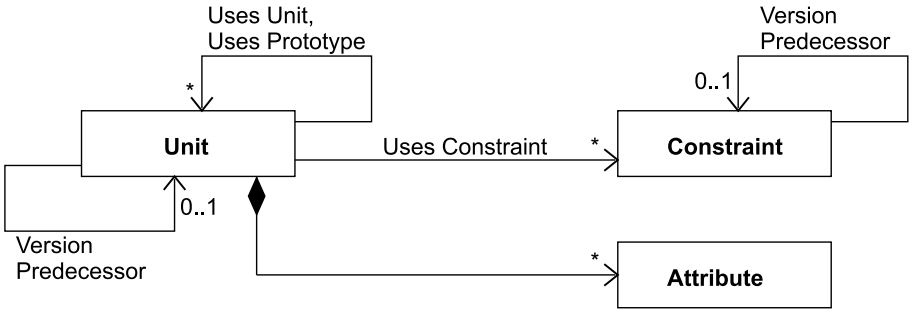

Abbildung 7.6: Beziehungen der Elemente untereinander

7.4 Subsysteme

Ein Subsystem $\mathrm{XS}(D, x)$ beschreibt die Menge aller Elemente, die von der Unit x aus über Uses Unit-Links und Uses Constraint-Links und gegebenenfalls über Uses Prototype-Links erreicht werden können. Die Unit x selbst ist ebenfalls ein Element von $\mathrm{XS}(D, x)$. Die Abbildung 7.7 zeigt die Subsysteme $\mathrm{XS}(D, a)$ und $\mathrm{XS}(D, d)$ einer Elementsammlung D. Die Sammlung D enthält die Units a, b, c, d, e, f, g, h und die Constraints x, y, z.

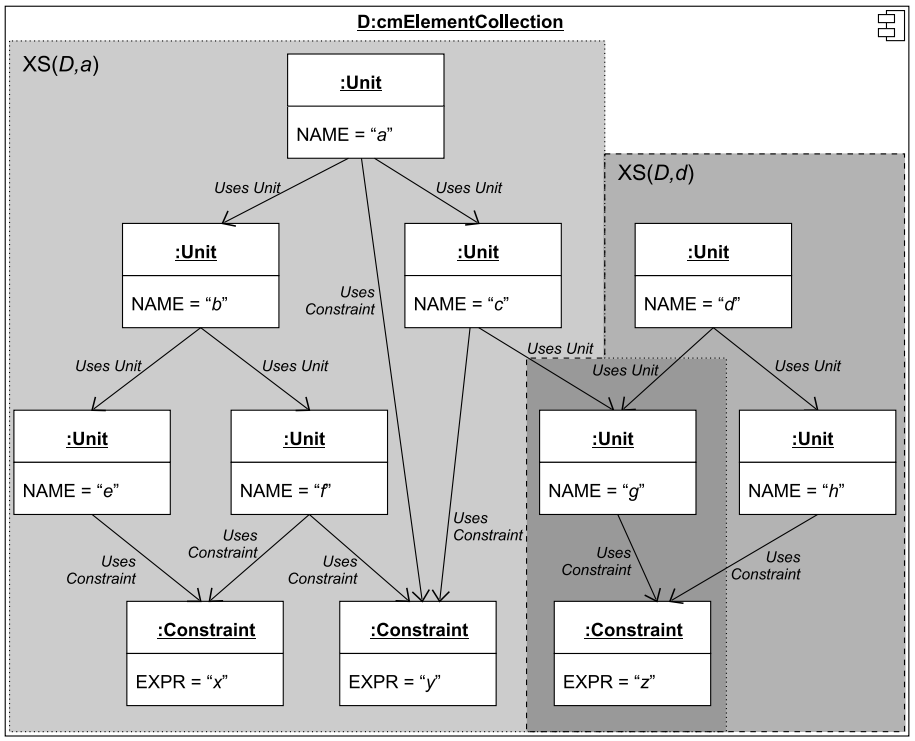

Abbildung 7.7: Subsysteme XS(D, a) und XS(D, d) in Elementsammlung D

7.5 Elementsammlung

Die Elementsammlung *cmElementCollection* ist eine Sammlung, die sowohl Units als auch Constraints speichern kann. In Abbildung 7.7 ist eine Elementsammlung dargestellt, die die die Units a, b, c, d, e, f, g, h und die Constraints x, y, z enthält.

7.6 Grammatik der Constraints

Abbildung 7.8 dokumentiert für die Nichtterminalzeichen in Backus-Naur-Form die Grammatik[2] für die Bedingungen der Constraints, welche in der Beispielanwendung aus Kapitel 11 verwendet wird. Weiterhin ist es zweckmässig, Ausdrücke der Form

```
IF <Bedingung> THEN <BedingungA>
```

und

```
IF <Bedingung> THEN <BedingungA> ELSE <BedingungB>
```

zu ermöglichen. Hierbei sind `<Bedingung>`, `<BedingungA>` und `<BedingungB>` Ausdrücke der Grammatik gemäss Abbildung 7.8.

Die Anwendung der Regeln zeigt Abbildung 7.9. Die Regeln greifen über den Dereferenzierungsoperator "_" auf die Attributwerte der Unit zu. So repräsentiert im

[2]Eine Einführung in Grammatiken kann dem Anhang entnommen werden.

```
Start                        ::= ( Expression <EOF> | <EOF> )
Expression                   ::= OrExpression
OrExpression                 ::= AndExpression ( ( "||" AndExpression ) )*
AndExpression                ::= EqualExpression ( ( "&&" EqualExpression ) )*
EqualExpression              ::= RelationalExpression (
                                   ( "!=" RelationalExpression )
                                   | ( "==" RelationalExpression ) )*
RelationalExpression         ::= AdditiveExpression ( ( "<" AdditiveExpression )
                                   | ( ">" AdditiveExpression )
                                   | ( "<=" AdditiveExpression )
                                   | ( ">=" AdditiveExpression ) )*
AdditiveExpression           ::= MultiplicativeExpression (
                                   ( "+" MultiplicativeExpression )
                                   | ( "-" MultiplicativeExpression ) )*
MultiplicativeExpression     ::= UnaryExpression ( ( PowerExpression )
                                   | ( "*" UnaryExpression )
                                   | ( "/" UnaryExpression )
                                   | ( "%" UnaryExpression ) )*
UnaryExpression              ::= ( "+" UnaryExpression )
                                   | ( "-" UnaryExpression )
                                   | ( "!" UnaryExpression )
                                   | PowerExpression
PowerExpression              ::= UnaryExpressionNotPlusMinus (
                                   ( "^" UnaryExpression ) )?
UnaryExpressionNotPlusMinus ::= AnyConstant
                                   | ( Function | Variable )
                                   | "(" Expression ")"
Variable                     ::= ( Identifier )
Function                     ::= ( Identifier "(" ArgumentList ")" )
ArgumentList                 ::= ( Expression ( "," Expression )* )?
Identifier                   ::= <IDENTIFIER>
AnyConstant                  ::= ( <STRING_LITERAL> | RealConstant | Array )
Array                        ::= "[" RealConstant ( "," RealConstant )* "]"
RealConstant                 ::= ( <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> )
```

Abbildung 7.8: Grammatik der Constraints nach [23]

Beispiel der Abbildung 7.9 der Term `A_u1` des Constraints den Wert des Attributs $u1$ der Unit mit Namen **A**.

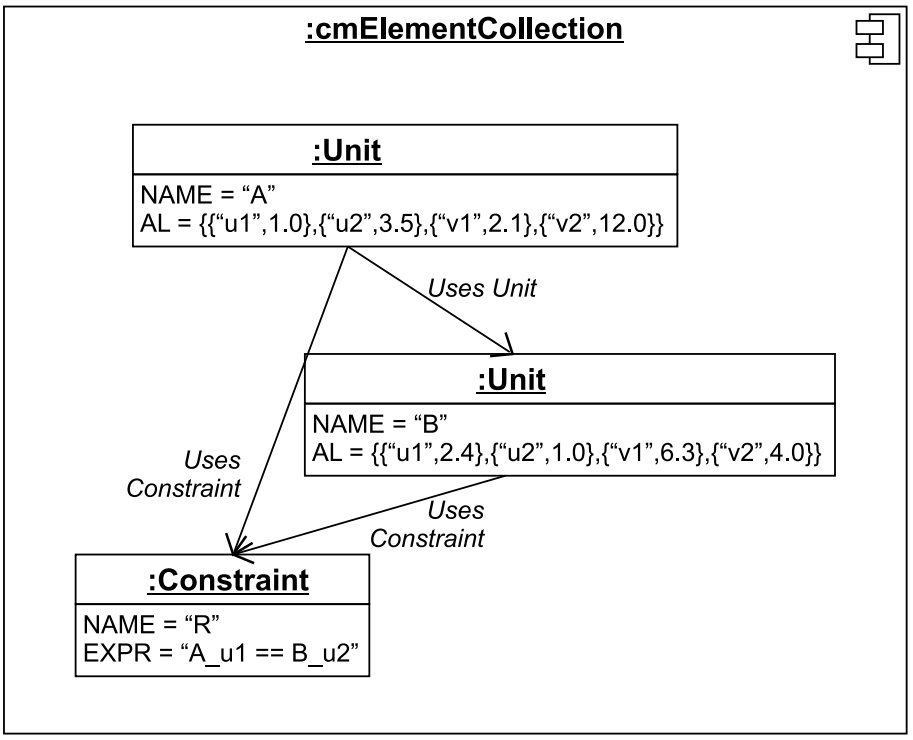

Abbildung 7.9: Gültigkeitsregeln für Attributwerte

Zusätzlich sind in den Ausdrücken Terme des Typs

```
[Unitname]_COUNT
```

möglich. Zur Evaluierung des Terms wird die Anzahl an Units mit dem Namen `[Unitname]` bestimmt. Das Beispiel in Abbildung 7.10 zeigt, welche Units bei der Evaluierung der Regel berücksichtigt werden.

Die Unit **a** verweist auf einen Constraint mit dem Ausdruck `c_COUNT == 4`. Zur Auswertung des Ausdrucks werden alle Units mit Namen **c** im Subsystem von **a** gezählt. Im Subsystem von **a** sind vier Units mit Namen **c** enthalten.

Die Unit **b** verweist auf einen Constraint mit dem Ausdruck `c_COUNT == 2`. Zur Auswertung des Ausdrucks werden alle Units mit Namen **c** im Subsystem von **b** gezählt. Im Subsystem von **b** sind zwei Units mit Namen **c** enthalten. Damit repräsentieren beide Constraints wahre Aussagen.

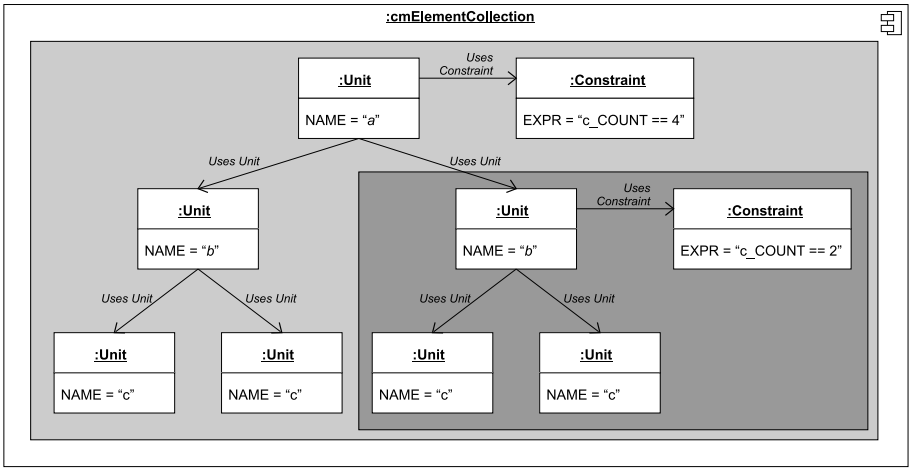

Abbildung 7.10: Regeln zur Kontrolle der Anzahl an Units

7.7 Grundlegende Operationen

7.7.1 Units

New-Operation

Die Operation *New* addiert eine neue aktive Unit zur Elementsammlung. Sie ist in Algorithmus 7.1 dargestellt.

Algorithmus 7.1 New(D: cmElementCollection): cmElementCollection

1: unit $new\_u := (\text{NewID}(D), \text{NULL}, \text{NULL}, \{\}, \{\}, \{\}, [\{\},]\text{NULL}, \text{FALSE})$;
2: RETURN$(D \cup \{new_u\})$;

Edit-Operation

Edit, Algorithmus 7.2 ändert den Inhalt einer bestehenden Unit. Alle Attribute ausser dem ID, dem VP und dem Fixed Flag können geändert werden. Dargestellt wird die Edit-Operation als eine Funktion, die eine Unit $x \in D$ durch ein x' ersetzt. Die Unit x' enthält die neuen Werte von x.

x wird durch x' ersetzt, wenn

$$\begin{aligned} x &\in D\,, \\ x.ID &= x'.ID\,, \\ x.VP &= x'.VP\,, \\ x.FX &= \text{FALSE} \quad \text{und} \\ x'.FX &= \text{FALSE} \end{aligned}$$

gilt.

Algorithmus 7.2 Edit(D: cmElementCollection, x: unit, x': unit): cmElementCollection

1: **if** $(x \notin D) \vee (x.FX = \text{TRUE}) \vee (x'.FX = \text{TRUE}) \vee (x.ID \neq x'.ID) \vee (x.VP \neq x'.VP)$ **then**
2: RETURN(D);
3: **else if** there is an ID in $(x'.NUU \cup x'.NUC\,[\cup x'.NUP])$ that does not correspond to the ID of any component in D **then**
4: RETURN(D);
5: **else**
6: RETURN $(D - \{x\} \cup \{x'\})$;
7: **end if**

Delete-Operation

Die *Delete*-Operation entfernt eine aktive Unit. Falls diese Unit von anderen Units verwendet wird, kann die Unit nicht gelöscht werden. Die Delete-Operation ist dann eine NULL-Operation. Die Funktion zeigt Algorithmus 7.3.

Algorithmus 7.3 Delete(D: cmElementCollection, x: unit): cmElementCollection

1: **if** $(x \notin D) \vee (x.FX \neq \text{FALSE})$ **then**
2: RETURN(D);
3: **else if** there is a y in $D - \{x\}$ such that $x \in (y.NUU[\cup y.NUP])$ **then**
4: RETURN(D);
5: **else**
6: RETURN $(D - \{x\})$;
7: **end if**

Snapshot-Operation

Die Operation *Snapshot*, Algorithmus 7.4 erzeugt eine unveränderliche Version eines Subsystems als Versionsvorgänger eines existierenden aktiven Subsystems.

ConnectSnapshot-Operation

Algorithmus 7.5, die *ConnectSnapshot*-Operation wird von der Snapshot-Operation benötigt.

Modified-Operation

Algorithmus 7.6, die *Modified*-Operation gibt TRUE zurück, falls die Unit gegenüber seinem Versionsvorgänger geändert wurde. Falls die Unit keinen Versionsvorgänger hat, gibt Modified ebenfalls TRUE zurück.

Algorithmus 7.4 Snapshot(D: cmElementCollection, x: unit)

1: **if** $(x \notin D) \vee (x.FX = \text{TRUE})$ **then**
2: RETURN(D);
3: **end if**
4: cmElementCollection $D' := D$;
5: SET of cmElements $mod\_set := \{\}$; /*elements that are modified*/
6: **for all** $y \in XS(D, x)$ **do**
7: **if** Modified(D, y) **then**
8: $mod\_set := mod\_set \cup y$;
9: **end if**
10: **end for**
11: SET of cmElements $mod\_xs\_rts := \{\}$; /*root elements of modified subsystems*/
12: **for all** $y \in XS(D, x)$ **do**
13: **if** XS(D, y) $\cap\, mod\_set \neq \{\}$ **then**
14: $mod\_xs\_rts := mod\_xs\_rts \cup y$;
15: **end if**
16: **end for**
17: **for all** $y \in mod\_xs\_rts$ **do**
18: **if** y is a unit **then**
19: new_u := unit:New(
NewID(D'), $y.NAME, y.DESCR, y.AL, \{\}, \{\}, [\{\},] y.VP$, TRUE);
20: **else if** y is a constraint **then**
21: new_u := constraint:New(NewID(D'), $y.DESCR, y.EXPR, y.VP$, TRUE);
22: **end if**
23: $D' := D' \cup \{new\_u\}$;
24: $y.VP := new\_u.ID$;
25: **end for**
26: **for all** $y \in mod\_xs\_rts$ **do**
27: VPred(D, y).NUU := ConnectSnapshot($D, y, y.NUU$);
28: VPred(D, y).NUC := ConnectSnapshot($D, y, y.NUC$);
29: [VPred(D, y).NUP := ConnectSnapshot($D, y, y.NUP$);]
30: **end for**
31: RETURN(D')

Algorithmus 7.5 ConnectSnapshot(D: cmElementCollection, y: unit, L: SET of cmElements): SET of identities

```
1: SET of identities result := {};
2: for all cmElement z ∈ L do
3:    if z.FX =TRUE then
4:       result := result ∪ {z.ID};
5:    else
6:       result := result ∪ {VPred(D, z).ID};
7:    end if
8: end for
```

Algorithmus 7.6 Modified(D: cmElementCollection, x: unit): BOOLEAN

```
1: if x.FX =TRUE then
2:    RETURN(FALSE);
3: else if x.VP = NULL then
4:    RETURN(TRUE);
5: else if (x.NAME ≠ VPred(D, x).NAME) ∨ (x.DESCR ≠
   VPred(D, x).DESCR) ∨ (x.AL ≠ VPred(D, x).AL) ∨ (x.NUU ≠
   VPred(D, x).NUU) ∨ (x.NUC ≠ VPred(D, x).NUC) [∨(x.NUP ≠
   VPred(D, x).NUP)] then
6:    RETURN(TRUE);
7: else
8:    RETURN(FALSE);
9: end if
```

Revise-Operation

Die Operation *Revise*, dargestellt als Algorithmus 7.7 erzeugt eine aktive Version eines Subsystems als Versionsnachfolger eines existierenden Subsystems.

Algorithmus 7.7 Revise(D: cmElementCollection, x: unit, W: SET of cmElements): cmElementCollection

```
1: if ¬x.FX ∨ x ∉ D then
2:    RETURN(D);
3: end if
4: D' := D;
5: SET of cmElements revise_set := XS(D, x) ∩ W;
6: SET of cmElements new_u_set := {};
7: for all y ∈ revise_set do
8:    if y is a unit then
9:       new_u := unit:New(
         NewID(D'), y.NAME, y.DESCR, y.AL, {}, {}, [{},]y.ID, FALSE);
10:   else if y is a constraint then
11:      new_u := constraint:New(NewID(D'), y.DESCR, y.EXPR, y.ID, FALSE);
12:   end if
13:   new_u_set := new_u_set ∪ {new_u};
14:   D' := D' ∪ {new_u};
15: end for
16: for all y ∈ new_u_set do
17:   y.NUU := ConnectRevisions(D, y, y.NUU, new_u_set);
18:   y.NUC := ConnectRevisions(D, y, y.NUC, new_u_set);
19:   [y.NUP := ConnectRevisions(D, y, y.NUP, new_u_set);]
20: end for
21: RETURN(D');
```

ConnectRevisions-Operation

Die Operation *ConnectRevisions*, dargestellt als Algorithmus 7.8 wird von der Revise-Operation gebraucht.

CheckConstraints-Operation

Algorithmus 7.9 repräsentiert die Operation *CheckConstraints*. Diese Operation durchläuft alle Constraints, die von einer Unit aus über Links erreicht werden können und wertet diese aus. Sind alle ausgewerteten Ausdrücke wahre Aussagen, gibt die Funktion den Wert TRUE zurück.

Algorithmus 7.8 ConnectRevisions(D: cmElementCollection, x: unit, L: SET of cmElements, N: SET of cmElements): SET of identities

```
1: SET of identities result := {};
2: for all w ∈ L do
3:    if there is a z in N such that VPred(z)=w then
4:       result := result ∪ {z.ID};
5:    else
6:       result := result ∪ {w.ID};
7:    end if
8: end for
```

Algorithmus 7.9 CheckConstraints(D: cmElementCollection, x: unit): BOOLEAN

```
1: SET of constraints xcs := XCS(D, x); /*all constraints, that are element of
   XS(D, x)*/
2: BOOLEAN result :=TRUE;
3: for all c ∈ xcs do
4:    SET of units check_set := all units u where (u ∈ XS(D, x) ∧ c ∈ u.NUC);
5:    for all y ∈ check_set do
6:       add y.AL to parser; /*add definition rules*/
7:    end for
8:    result := result ∧ parseExpression(c.EXPR); /*evaluate validation rules*/
9: end for
10: RETURN(result);
```

7.7.2 Constraints

New-Operation

Die Operation *New* addiert einen neuen aktiven Constraint zur Komponentensammlung. Diese Operation ist als Algorithmus 7.10 dargestellt.

Algorithmus 7.10 New(D: cmElementCollection): cmElementCollection

1: constraint $new\_c := (\text{NewID}(D), \text{NULL}, \text{NULL}, \text{NULL}, \text{FALSE})$;
2: RETURN($D \cup \{new\_c\}$);

Edit-Operation

Edit, Algorithmus 7.11, ändert den Inhalt eines bestehenden Constaint. Die Attribute ID, VP und FX können nicht geändert werden. Implementiert wird die Edit-Operation als eine Funktion, die einen Constraint $x \in D$ durch ein x' ersetzt. Der Constraint x' enthält die neuen Werte von x.

x wird durch x' ersetzt, wenn

$$\begin{aligned} x &\in D\,, \\ x.ID &= x'.ID\,, \\ x.VP &= x'.VP\,, \\ x.FX &= \text{FALSE} \quad \text{und} \\ x'.FX &= \text{FALSE} \end{aligned}$$

gilt.

Algorithmus 7.11 Edit(D: cmElementCollection, x: constraint, x': constraint): cmElementCollection

1: **if** $(x \notin D) \vee (x.FX = \text{TRUE}) \vee (x'.FX = \text{TRUE}) \vee (x.ID \neq x'.ID) \vee (x.VP \neq x'.VP)$ **then**
2: RETURN(D);
3: **else**
4: RETURN($D - \{x\} \cup \{x'\}$);
5: **end if**

Delete-Operation

Die *Delete*-Operation entfernt einen aktiven Constraint. Wie in Algorithmus 7.12 ersichtlich, kann der Constraint nicht gelöscht werden, falls dieser von anderen Units verwendet wird. Die Delete-Operation ist dann eine NULL-Operation.

Algorithmus 7.12 Delete(D: cmElementCollection, x: constraint): cmElementCollection

```
1: if (x ∉ D) ∨ (x.FX ≠ FALSE) then
2:    RETURN(D);
3: else if there is a y in D such that x ∈ y.NUU ∨ x ∈ y.NUC then
4:    RETURN(D);
5: else
6:    RETURN (D − {x});
7: end if
```

Modified-Operation

Die *Modified*-Operation, Algorithmus 7.13 gibt TRUE zurück, falls der Constraint gegenüber seinem Versionsvorgänger geändert wurde. Falls der Constraint keinen Versionsvorgänger hat, gibt Modified ebenfalls TRUE zurück.

Algorithmus 7.13 Modified(D: cmElementCollection, x: constraint): BOOLEAN

```
1: if x.FX = TRUE then
2:    RETURN(FALSE);
3: else if x.VP = NULL then
4:    RETURN(TRUE);
5: else if (x.DESCR ≠ VPred(D, x).DESCR) ∨ (x.EXPR ≠
   VPred(D, x).EXPR) then
6:    RETURN(TRUE);
7: else
8:    RETURN(FALSE);
9: end if
```

Kapitel 8

Der Produktkonfigurator des neuen Konzepts

8.1 Allgemeines

Dieses Kapitel beschreibt die Komponenten und Methoden zur Produktkonfiguration. Die Produktkonfigurationen und ihre zugehörigen Regeln werden als gerichtete, azyklische Graphen modelliert. Die versionierte Verwaltung der Konfigurationen erfolgt wie in Kapitel 7 beschrieben.

8.2 Produktmodell

Zur Abbildung beliebiger Produkte erfolgt wie in [29] dargelegt eine Aufteilung in die Bereiche

- Type Level,
- Prototype Level und
- Configuration Level.

Die Struktur eines Produkts wird im *Type Level* definiert. Die Zusammenstellung aller möglichen Ausprägungen der im Type Level definierten Elemente erfolgt im *Prototype Level* . Eine kunden- beziehungsweise auftragsbezogene Konfiguration wird im Configuration Level erstellt. Tabelle 8.1 stellt diese Aufteilung dar.

Der Aufbau einer benutzer- beziehungsweise auftragsspezifischen Produktkonfiguration im Configuration Level greift zunächst auf die Abbildung des Produkts im Type Level zurück. In diesem Level ist die Produktzusammensetzung definiert. In einem unmittelbar darauffolgenden Schritt wird dann beispielsweise durch den Benutzer eine der möglichen Ausprägungen aus dem Prototype Level gewählt, womit den Attributen Werte zugewiesen werden.

Type Level	Struktur des Produkts
Prototype Level	Ausprägungen der Elemente des Type Level
Configuration Level	Kunden- beziehungsweise auftragsbezogene Konfiguration

Tabelle 8.1: Produktmodell

8.2.1 Type Level

Die Abbildung einer Produktkonfiguration erfolgt wie in [29] dargelegt in einer Menge miteinander verbundener Elemente. Die Elemente können Attribute besitzen. Die Abbildung von Gültigkeitsregeln erfolgt nicht wie bei Hedin et al. in [29], indem diese Regeln als Bestandteile der Elemente modelliert werden. Die Gültigkeitsregeln werden als eigenständige Elemente im Type Level abgebildet. Der wesentliche Vorteil ist hierbei, dass die Gültigkeitsregeln auf die für ihre Auswertung relevanten Elemente, Attribute und Attributwerte direkt zugreifen können. Als Konsequenz hiervon wird ein Produkt inklusive Gültigkeitsregeln nicht mehr durch einen Baum sondern durch einen Graphen abgebildet. Das Produktmodell ohne diese Regeln ist weiterhin durch eine Baumstruktur abbildbar.

Die Abbildung 8.1 zeigt eine einfache Produktstruktur. Das modellierte Produkt **A** besteht aus den Komponenten **B** und **C**. Die Komponente **B** wiederum besteht aus den Komponenten **X** und **Y**. Für die Komponenten **B** und **Y** soll für die Werte der Attribute $u1$ von **B** und $u2$ von **Y** die Beziehung

$$\mathbf{B}\_u1 = \mathbf{Y}\_u2$$

gelten. Die entstandene Struktur ist ein *gerichteter azyklischer Graph.*

Das Produktmodell im Type Level besitzt damit zusammenfassend die folgenden Eigenschaften.

- Eine Produktkonfiguration ist eine Anordnung miteinander verbundener Elemente.
- Eine Produktkonfiguration ohne Gültigkeitsregeln kann durch einen Baum abgebildet werden.
- Durch Hinzunahme von Gültigkeitsregeln entsteht aus dem Baum ein gerichteter azyklischer Graph.
- Eine Produktkonfiguration hat genau eine Wurzelkomponente.

Als Datenhaltung für den Type Level dient eine Elementsammlung, die in Abschnitt 7.5 beschrieben ist.

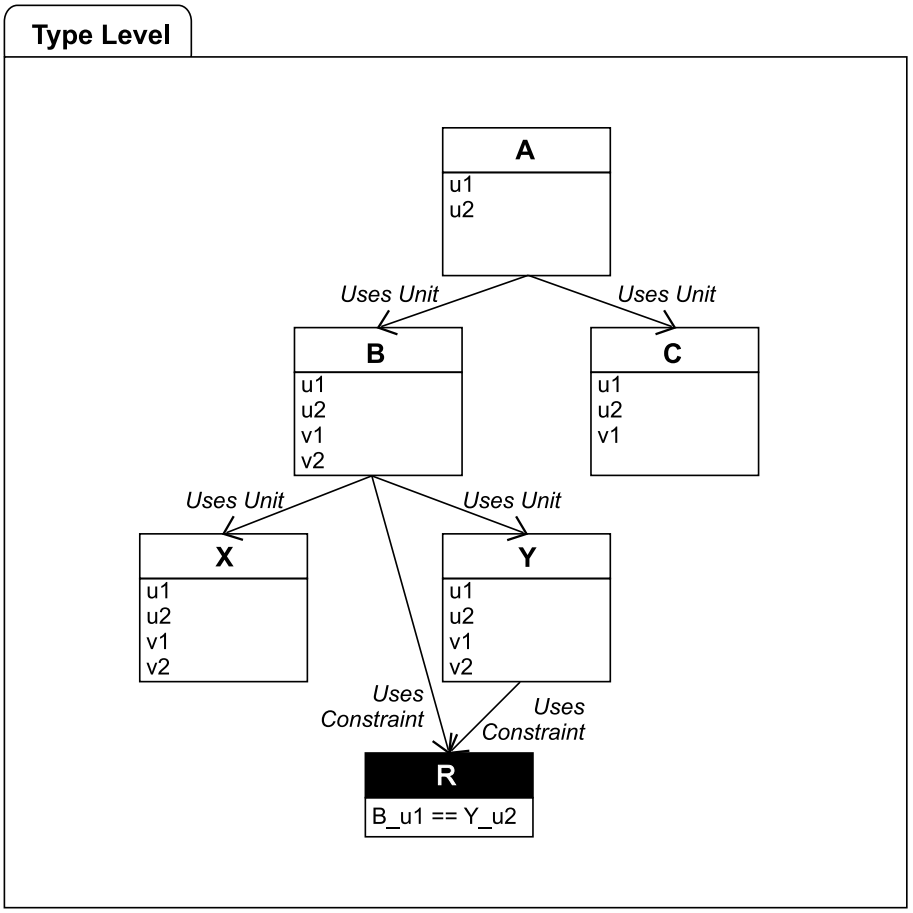

Abbildung 8.1: Modell im Type Level mit einer Gültigkeitsregel **R**

8.2.2 Prototype Level

Wie beschrieben erfolgt die Definition aller möglichen Ausprägungen eines Prototypelements des Type Level im Prototype Level. Die Integration des Prototype Level kann grundsätzlich auf zwei unterschiedliche Arten erfolgen.

1. Prototype Level als eine vom Type Level getrennte Datenhaltung.
2. Prototype Level integriert in die Datenhaltung des Type Level.

Da die Versionierung für jede Datenhaltung unabhängig voneinander erfolgt, ergeben sich Konsequenzen aus der unterschiedlichen Realisierung.

Getrennte Datenhaltung für Prototype Level und Type Level

Eine Darstellung einer getrennten Datenhaltung der Elemente des Type Level und des Prototype Level zeigt Abbildung 8.2. Der Type Level und der Prototype Level werden in je einer Elementsammlung gemäss Abschnitt 7.5 verwaltet.

Bei der getrennten versionierten Datenhaltung des Type Level und des Prototype Level sind

- weniger komplexe Strukturen zu verwalten, was zu einer vergleichsweise
- besseren Performance der Datenhaltung führt.
- Weiterhin entwickelt sich der Protoype Level und der Type Level unabhängig voneinander.

Nachteilig ist, dass die Elemente des Prototype Level nicht Bestandteil der versionierten Konfigurationen des Type Level sind. Abhängig vom Anwendungsbereich des Systems ist diese Trennung möglich, beispielsweise bei Produkten, bei denen sich im zeitlichen Verlauf aussschliesslich die Elemente des Prototype Level verändern.

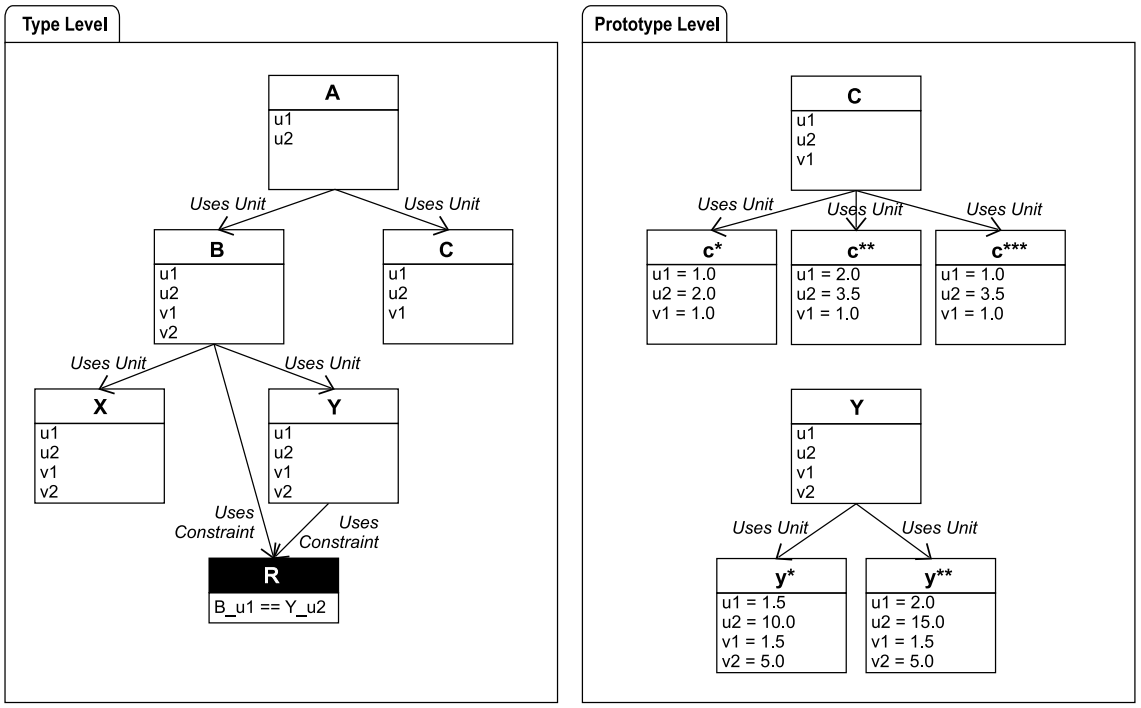

Abbildung 8.2: Type Level und Prototype Level getrennt

Gemeinsame Datenhaltung für Prototype Level und Type Level

Bei einer integrierten Datenhaltung werden die Elemente des Prototype Level zusammen mit den Elementen des Type Level in einer Elementsammlung nach Abschnitt 7.5 verwaltet.

Damit werden

- alle Elemente geschlossen versionsgestützt verwaltet und

- es ist jederzeit klar, welche Elemente des Prototype Level innerhalb einer bestimmten Konfiguration verfügbar sind.

Als Folge hiervon wird der zu verwaltende Graph komplexer, was eine tendenziell schlechtere Performance der Datenhaltung zur Folge haben kann. Abbildung 8.3 zeigt an einem einfachen Beispiel die integrierte Datenhaltung. Das Element **C** hat drei mögliche Ausprägungen **c***, **c**** und **c***** . Weiterhin besitzt das Element **Y** zwei mögliche Ausprägungen **y*** und **y****.

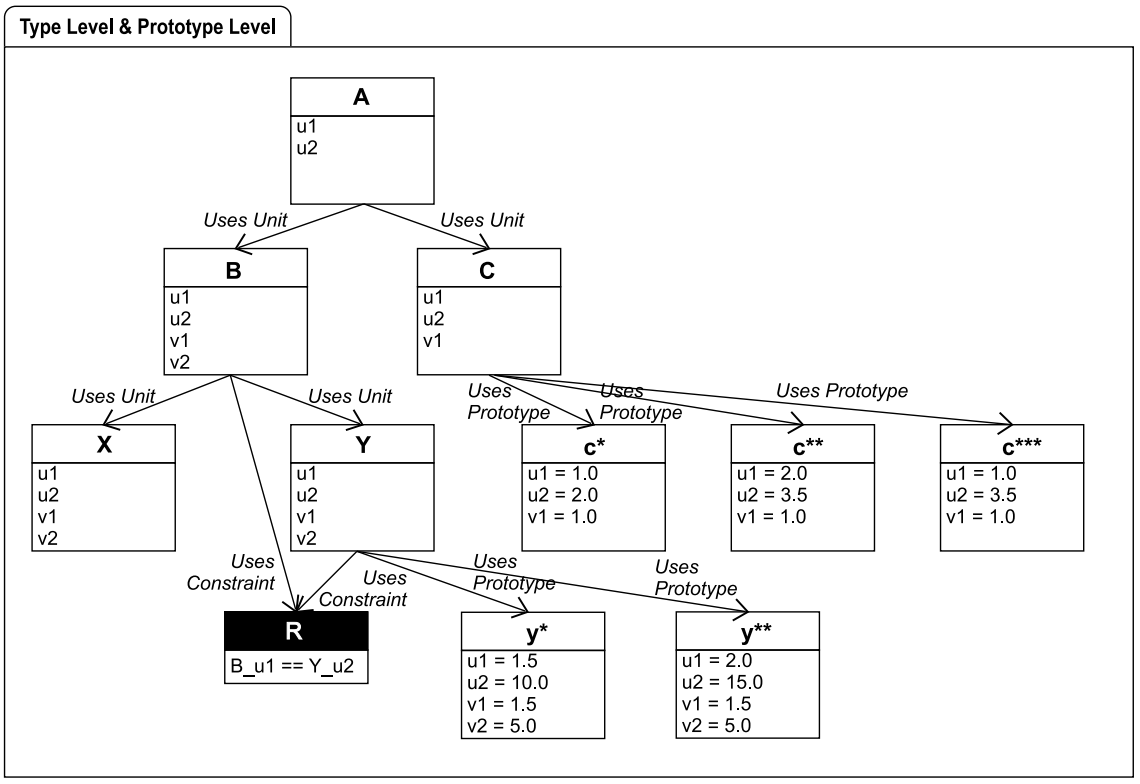

Abbildung 8.3: Integration der Elemente des Prototype Level in den Type Level

Zur Integration der Elemente des Prototype Level in den Type Level existieren mehrere Möglichkeiten.

- Erweiterung der Units um den Verknüpfungstyp *Uses Prototype* zusätzlich zu den *Uses Unit*- und *Uses Constraint*-Links. Hierbei zeigen die neuen Links auf Units.
- Ableitung von Prototype Units von den Units durch Vererbung und Verknüpfung dieser Prototype Units mit der zugehörigen Units über *Uses Unit*-Links

Die Erweiterung der Units um Uses Prototype-Links bietet folgenden Vorteil.

- Zur Auswahl einer bestimmten physischen Ausprägung muss das Planungssystem alle zur Verfügung stehenden Ausprägungen ermitteln und anzeigen können. Diese Elemente können duch die Auswertung der Uses Prototype-Links schnell ermittelt werden.

Gewählte Umsetzung

Im Rahmen dieser Arbeit wurde eine Integration der Elemente des Prototype Level in den Type Level gewählt. Dies wurde durch die Erweiterung der Units um Uses Prototype-Links realisiert. Die Prinzipien des Konfigurationsmanagements und die grundlegenden Methoden erfahren hierdurch keine wesentliche Veränderung.

8.2.3 Configuration Level

Im Configuration Level entsteht im Verlauf einer Produktkonfiguration ein kunden- beziehungsweise auftragsbezogenes Produkt. Grundlage sind das Produktmodell des Type Level und die Zusammenstellungen der möglichen Ausprägungen der Elemente des Prototype Level. Zur Unterstützung der Versionierung wird auch für den Configuration Level eine Elementsammlung gemäss Abschnitt 7.5 verwendet.

Die wesentlichen Operationen zur Erstellung einer Produktkonfiguration sind

- das Einfügen von Elementen mit Berücksichtigung von möglichen Ausprägungen,
- das automatische Einfügen aller relevanten Gültigkeitsregeln und
- das programmautomatische Prüfen der Gültigkeitsregeln der Konfiguration.

Das programmautomatische Prüfen aller in einer Konfiguration enthaltenen Regeln ist in Algorithmus 7.9 beschrieben. Die Algorithmen zum Einfügen von Elementen und zum automatischen Einfügen der Regeln werden in den folgenden Abschnitten beschrieben.

8.3 Abbildung der Elemente des Produktmodells

Durch die im Rahmen der Arbeit gewählte Integration der Elemente des Type Level und der Elemente des Prototype Level ergibt sich, dass zwei Datenhaltungen zur Abbildung aller Daten erforderlich sind.

Eine erste Datenhaltung verwaltet alle Elemente aus den Bereichen Type Level und Prototype Level, eine zweite Datenhaltung verwaltet die Elemente des Bereichs Configuration Level. Die Abbildung der Elemente zur Repräsentation einer Produktzusammensetzung zeigt Abbildung 8.4 am Beispiel der integrierten Datenhaltung für die Bereiche Type Level und Prototype Level. Die Abbildung zeigt alle wesentlichen Elemente und wie diese in einer Elementsammlung nach Abschnitt 7.5 abgebildet werden können.

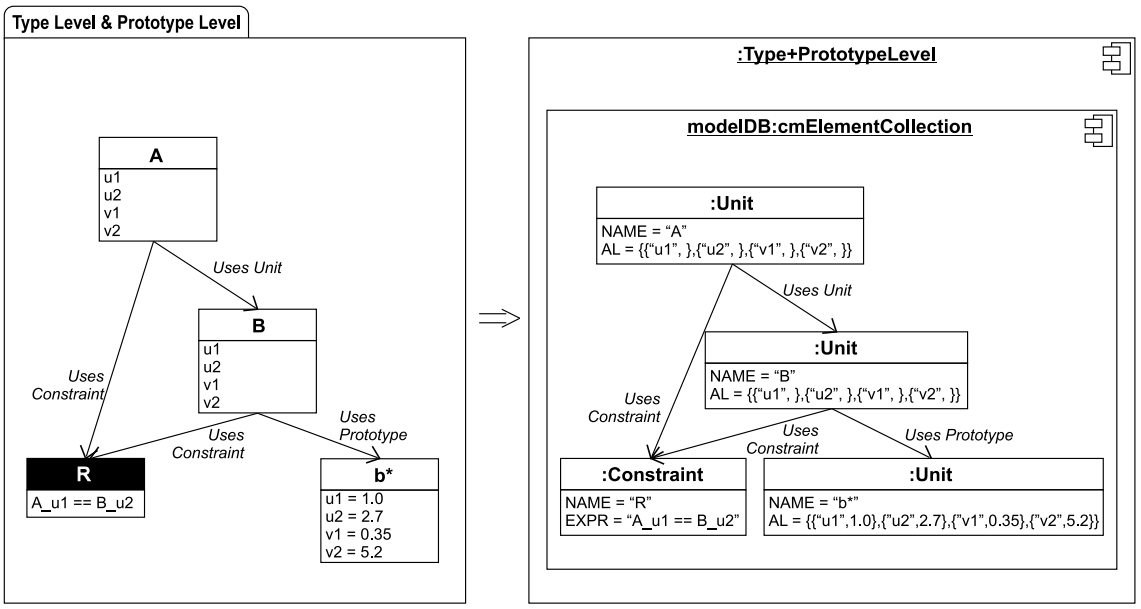

Abbildung 8.4: Abbildung der Elemente des Type Level

8.4 Einfügen eines Elements

Die Modellierung einer Produktzusammensetzung im Type Level entspricht dem Grundsatz "nodes are models" aus [19]. Das bedeutet, dass die Anzahl erlaubter Instanzen im Configuration Level nicht aus der Anzahl an Elementen des Type Level folgt. Die korrekte Anzahl an Instanzen eines Elements im Configuration Level kann über Gültigkeitsregeln kontrolliert werden.

Algorithmus 8.1 zeigt, wie auf der Grundlage des Produktmodells des Type Level ein Element in eine Produktkonfiguration im Configuration Level eingefügt wird. Hierbei werden dem Algorithmus die folgenden Variablen übergeben.

configurationLevel: Die Datenhaltung, in der die auftrags- oder kundenspezifische Konfiguration erstellt werden soll.

typeLevel: Die Datenhaltung, die das Produktmodell mit den dazugehörigen Regeln enthält.

unitInTypeLevel: Das Element im Type Level, von dem eine Instanz in den Configuration Level eingefügt werden soll.

unitInPrototypeLevel: Das Element im Prototype Level, von dem die Attributwerte übernommen werden sollen.

rootInConfigurationLevel: Wurzelelement im Configuration Level. Die Datenhaltung des Configuration Level kann mehrere Produktkonfigurationen enthalten. Dies ergibt sich beispielsweise aus Snapshot- und Revise- Operationen. Das Wurzelelement der Konfiguration, an der aktuell gearbeitet werden soll, kann zum Beispiel durch den Bearbeiter festgelegt werden.

rootInTypeLevel: Wurzelelement im Type Level. Die Datenhaltung des Type Level kann mehrere Konfigurationen enthalten. Das Wurzelelement der Konfiguration, die als relevantes Produktmodell benutzt werden soll, kann beispielsweise durch den Bearbeiter festgelegt werden.

Algorithmus 8.1 bestimmt zunächst das Vater-Element für das einzufügende Element. Existiert dieses nicht, ist das einzufügende Element ein Wurzelelement und wird entsprechend in den Configuration Level eingefügt. Existiert ein Vater-Element, so werden auf der Grundlage des Namens dieses Vater-Element alle Elemente des Configuration Level geprüft. Ist darunter ein Element,

- dessen Namen mit dem des Vater-Elements übereinstimmt,
- das von der aktuellen Wurzel des Configuration Level aus erreichbar ist und
- bei dem alle Werte der Schlüsselvariablen des einzufügenden Elements mit denen des Elements übereinstimmen,

so wird das neue Element unterhalb dieses Elements eingefügt.

Abbildung 8.5 veranschaulicht dieses Vorgehen. Die Schlüsselfelder des Elements **X** sind durch das führende Zeichen "$" gekennzeichnet.

8.5 Einfügen der Regeln

Das Einfügen der Regeln und die Verknüpfung mit den zugehörigen Elementen im Configuration Level erfolgt nach jedem Einfügen eines Elements, wie Algorithmus 8.1 zeigt.

Das Vorgehen zum Einfügen der Regeln stellt Abbildung 8.6 dar. In die Konfiguration des Configuration Level mit der Wurzel **a** wurde das Element **y** eingefügt. Dabei ist das Element **y** eine Instanz des Elements **Y** des Type Level. Aus dem Produktmodell im Type Level ist die Information verfügbar, dass mit Elementen des Typs **Y** eine Regel **R** verknüpft ist. Weiterhin ist diese Regel mit einem Element **B** verknüpft.

Im Configuration Level wird eine Instanz **r** der Regel **R** mit **y** verknüpft. Weiterhin werden alle Instanzen des Elements **B** überprüft. Die Instanz **r** der Regel **R** wird mit derjenigen Instanz des Elements **B** verknüpft, von der aus das eingefügte **y** erreichbar ist.

Der Algorithmus für das Einfügen der Regeln ist als Algorithmus 8.4 dargestellt.

Algorithmus 8.1 Add($configurationLevel$: cmElementCollection, $typeLevel$: cmElementCollection, $unitInTypeLevel$: unit, $unitInPrototypeLevel$: unit, $rootInConfigurationLevel$: unit, $rootInTypeLevel$: unit):cmElementCollection

```
1: fatherInTypeLevel := GetFather(unitInTypeLevel, rootInTypeLevel,
   typeLevel);
2: if fatherInTypeLevel = NULL then
3:    instanceUnit := newUnit();
4:    SetAttributes(instanceUnit, GetAttributes(unitInPrototypeLevel));
5:    instanceUnit.NAME := unitInTypeLevel.NAME;
6:    ConnectConstraints(unitInTypeLevel, instanceUnit, rootInTypeLevel,
      rootInConfigurationLevel, typeLevel, configurationLevel);
7:    RETURN (configurationLevel ∪ {instanceUnit});
8: else
9:    fatherName := fatherInTypeLevel.NAME;
10:   for all units x ∈ configurationLevel do
11:      if (x ∈ XS(rootInConfigurationLevel)) ∧ (x.NAME = fatherName) ∧
         (CompareRelevantAttributes(x, unitInPrototypeLevel)) then
12:         instanceUnit = newUnit();
13:         SetAttributes(instanceUnit, GetAttributes(unitInPrototypeLevel));
14:         instanceUnit.NAME := unitInTypeLevel.NAME;
15:         x.NUU := x.NUU ∪ {instanceUnit};
16:         ConnectConstraints(unitInTypeLevel, instanceUnit, rootInTypeLevel,
            rootInConfigurationLevel, typeLevel, configurationLevel);
17:         RETURN (configurationLevel ∪ {instanceUnit});
18:      end if
19:   end for
20: end if
```

Algorithmus 8.2 GetFather(u: unit, $root$: unit, $cmElementCollection$: cmElementCollection): unit

```
1: unit fatherUnit;
2: fatherUnit := NULL;
3: for all units x ∈ cmElementCollection do
4:    if (x ∈ XS(root)) ∧ (u ∈ x.NUU) then
5:       fatherUnit := x;
6:    end if
7: end for
8: RETURN(fatherUnit);
```

Algorithmus 8.3 CompareRelevantAttributes($fatherUnit$: unit, $childUnit$: unit): BOOLEAN

```
1: if all values of key-attributes ∈ childUnit.AL are equal to the values of the
   corresponding attributes ∈ fatherUnit.AL then
2:    RETURN(TRUE);
3: else
4:    RETURN(FALSE);
5: end if
```

Algorithmus 8.4 ConnectConstraints($modelUnit$: unit, $instanceUnit$: unit, $rootInTypeLevel$: unit, $rootInConfigurationLevel$:unit, $typeLevel$: cmElement-Collection, $configurationLevel$: cmElementCollection)

```
1: for all constraints c ∈ modelUnit.NUC do
2:    SET of units conn_set := {};
3:    for all units u ∈ typeLevel do
4:       if (u ∈ XS(rootInTypeLevel)) ∧ (c ∈ u.NUC) ∧ (u ≠ modelUnit) then
5:          conn_set := conn_set ∪ {u};
6:       end if
7:    end for
8:    if conn_set = {} then
9:       instanceConstraint := newConstraint();
10:      instanceConstraint.EXPR := c.EXPR;
11:      instanceUnit.NUC := instanceUnit.NUC ∪ {instanceConstraint};
12:   else
13:      SET of units units2treat := {};
14:      for all units cu ∈ conn_set do
15:         for all units iu ∈ configurationLevel do
16:            if (iu ∈ XS(rootInConfigurationLevel)) ∧ (iu.NAME =
               cu.NAME) ∧ (instanceUnit ∈ XS(iu)) then
17:               SET of units units2treat := units2treat ∪ {iu};
18:            end if
19:         end for
20:      end for
21:      instanceConstraint := newConstraint();
22:      instanceConstraint.EXPR := c.EXPR;
23:      instanceUnit.NUC := instanceUnit.NUC ∪ {instanceConstraint};
24:      for all units u ∈ units2treat do
25:         u.NUC := u.NUC ∪ {instanceConstraint};
26:      end for
27:   end if
28: end for
```

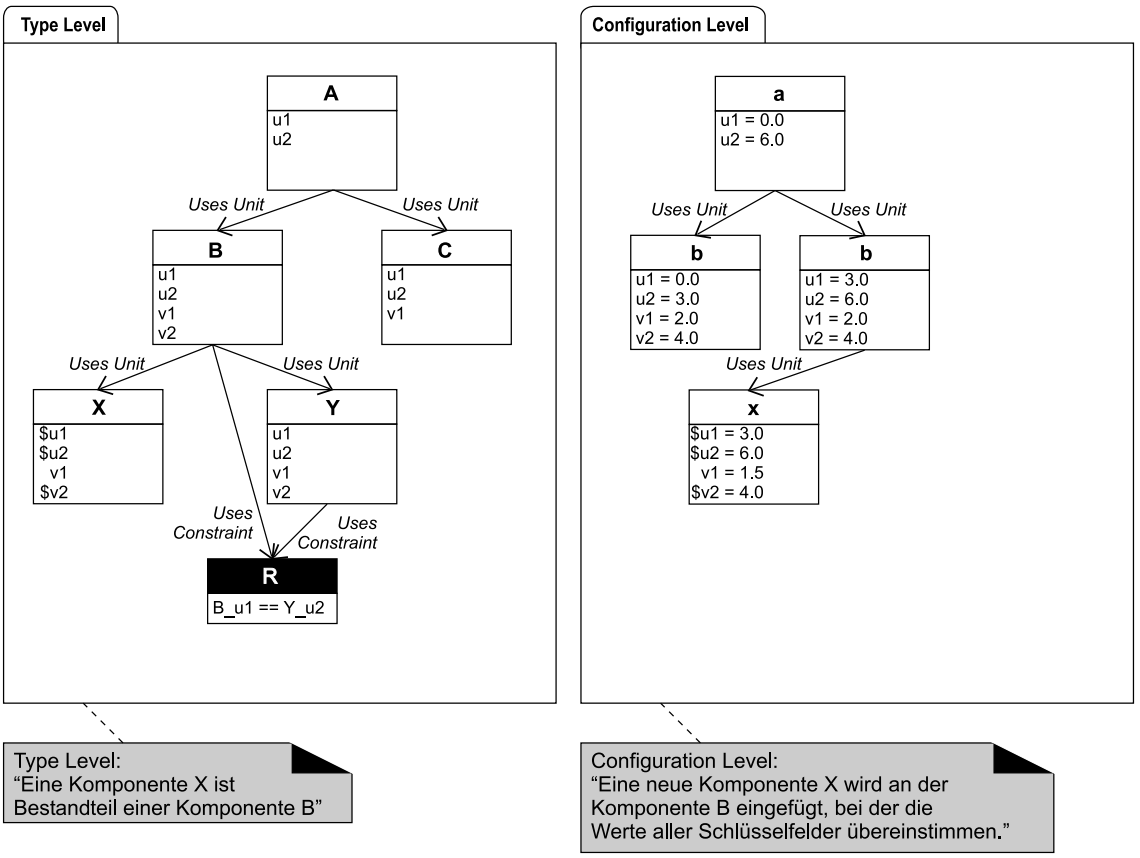

Abbildung 8.5: Einfügen einer Instanz von **X**

8.6 Vererbung

Das Einfügen neuer Elemente wird, wie in Abschnitt 8.4 beschrieben, über zwei Mechanismen gesteuert.

1. Die Auswertung des Graphen des Produktmodells.
2. Der Vergleich der Schlüsselattribute.

Da die Übereinstimmung der Werte der Schlüsselattribute ein Kriterium beim Einfügen eines Elements darstellen, muss auch das nachträgliche Ändern von Attributwerten berücksichtigt werden. Wird ein Attributwert eines Elements **a** verändert, so ist zu prüfen, ob dieses Attribut in Elementen, die von **a** aus erreichbar sind, ein Schlüsselattribut ist. Falls ja, dann ist in diesen Elementen der Wert des entsprechenden Attributs ebenfalls zu ändern. Den Vorgang stellen die Abbildungen 8.7 bis 8.9 dar. Abbildung 8.7 zeigt eine Produktkonfiguration im Configuration Level. Vereinfachend sind hierbei nur die Werte der Attribute von Elementen **b** und von **y** dargestellt. Die Attribute $v1$ und $v2$ des Elements **y** sind Schlüsselattribute. Daraus folgt, dass das Element an der Instanz **b** von **B** angehängt wurde, da bei dieser die Werte der entsprechenden Attribute gleich waren.

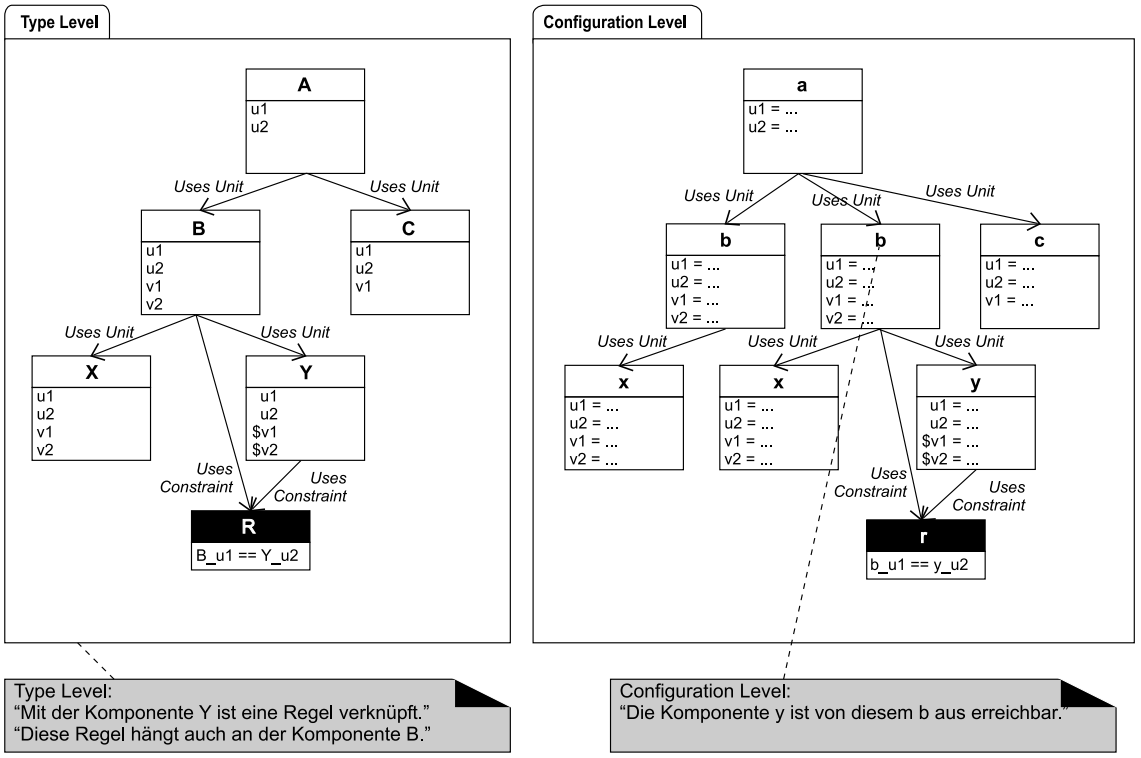

Abbildung 8.6: Einfügen einer Regel **R**

In der Konfiguration in Abbildung 8.8 ist der gekennzeichnete Wert des Attributs $v1$ eines Elements **b** geändert. Die kann beispielsweise manuell durch eine Benutzereingabe geschehen. Weiterhin ist ersichtlich, dass das genannte Attribut ein Schlüsselattribut für das referenzierte Element **y** ist.

Der Wert des Attributs $v1$ des Elements **y** wird automatisch geändert. Abbildung 8.9 zeigt die Produktkonfiguration nach der Änderung der Werte.

8.7 Klassifikation der Regeln

Die Regeln zur Anwendung im Produktkonfigurator lassen sich in drei Klassen einteilen.

Regeln für Attributwerte: Diese Regeln greifen auf Attributwerte sie referenzierender Elemente zu. Damit werten diese Regeln den Zustand eines Elements aus. Sie werden im Type Level definiert und prüfen Attributwerte, die sich beim Aufbau einer Konfiguration im Configuration Level beispielsweise durch Wahl einer Ausprägung aus dem Prototype Level ergeben.

Regeln für Elemente: Diese Regeln beziehen sich auf eine Menge von Elementen. Zu diesen Regeln gehören beispielsweise Regeln mit Termen `[Unitname]_COUNT`. Diese Regeln werden im Type Level definiert und prüfen,

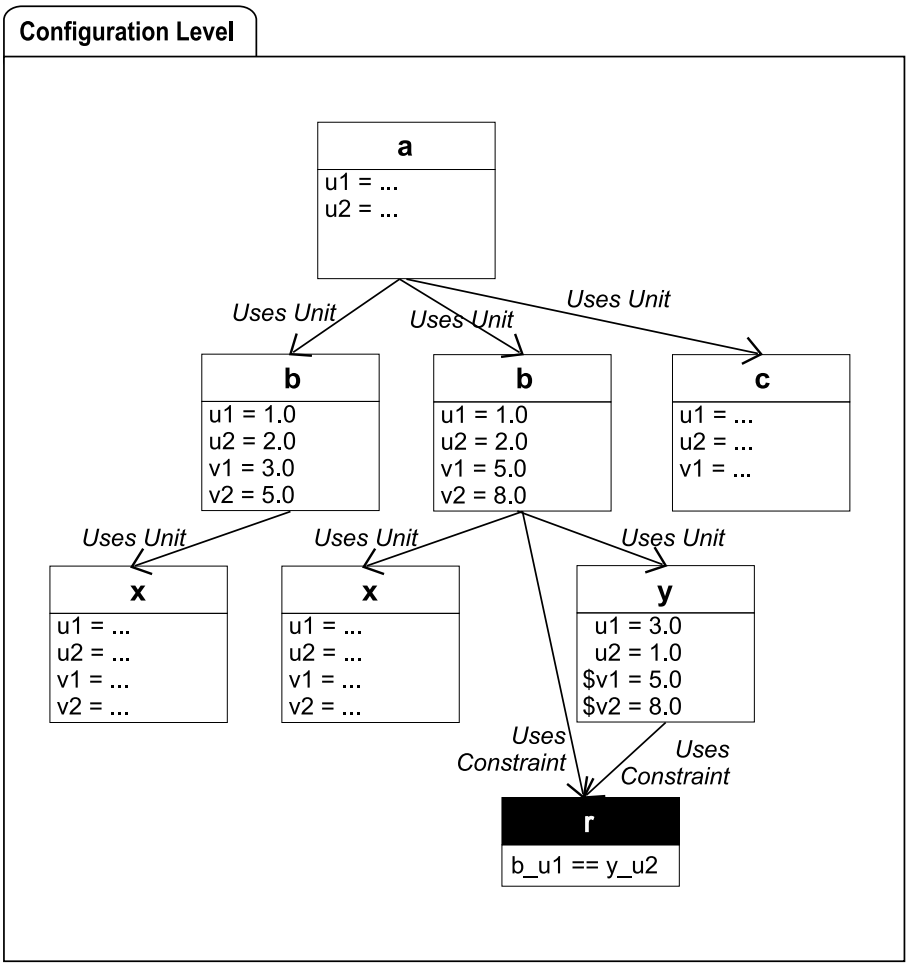

Abbildung 8.7: Produktkonfiguration im Configuration Level

ob die im Configuration Level generierten Elementmengen definierten Eigenschaften genügen. Dies sind beispielsweise Regeln, die die notwendige Mindestanzahl oder die maximale Anzahl an Elementen prüfen.

Benutzerspezifische Regeln: Regeln, die den beiden beschriebenen Klassen zugeordnet werden können und durch einen Benutzer in einer bestimmten Konfiguration des Configuration Level eingefügt werden. Abbildung 8.12 zeigt eine benutzerspezifische Regel. Dabei ist es auch möglich, eine benutzerspezifische Regel mit mehreren Elementen **b** zu verknüpfen, wenn diese im Configuration Level umbenannt werden und damit eindeutige Namen zur Referenzierung der Attributwerte erhalten.

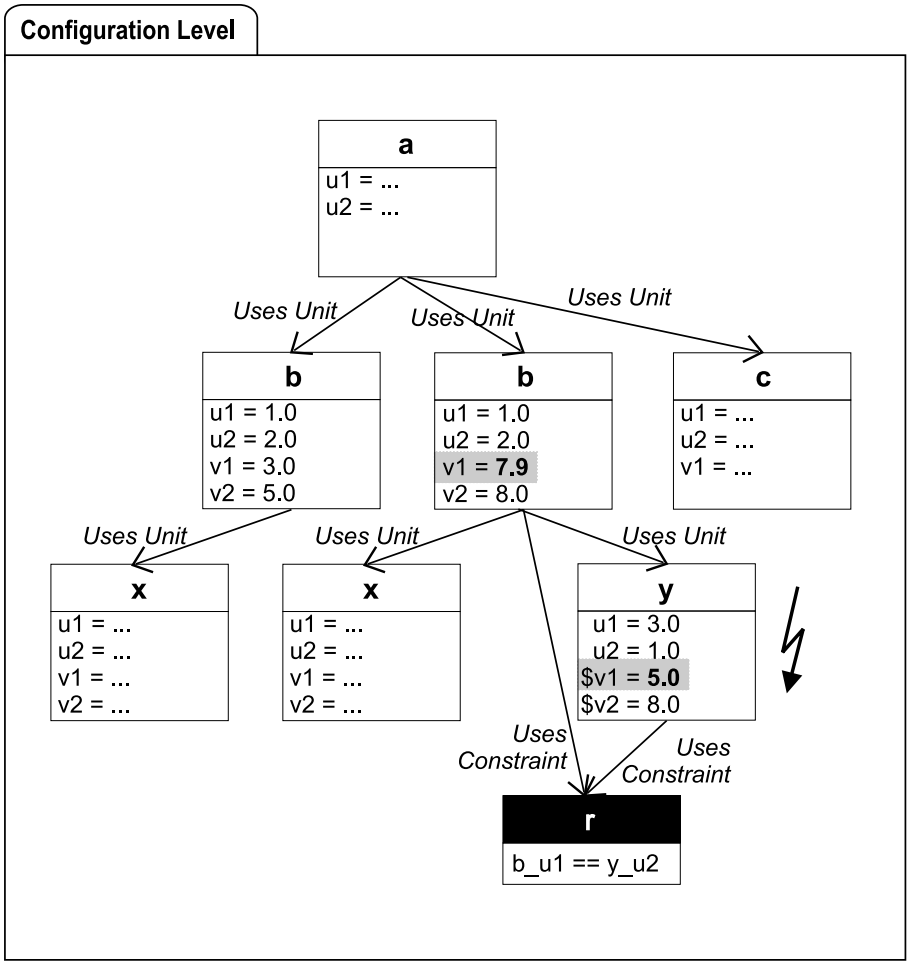

Abbildung 8.8: Produktkonfiguration mit geändertem Wert des Attributs $v1$ in **b**.

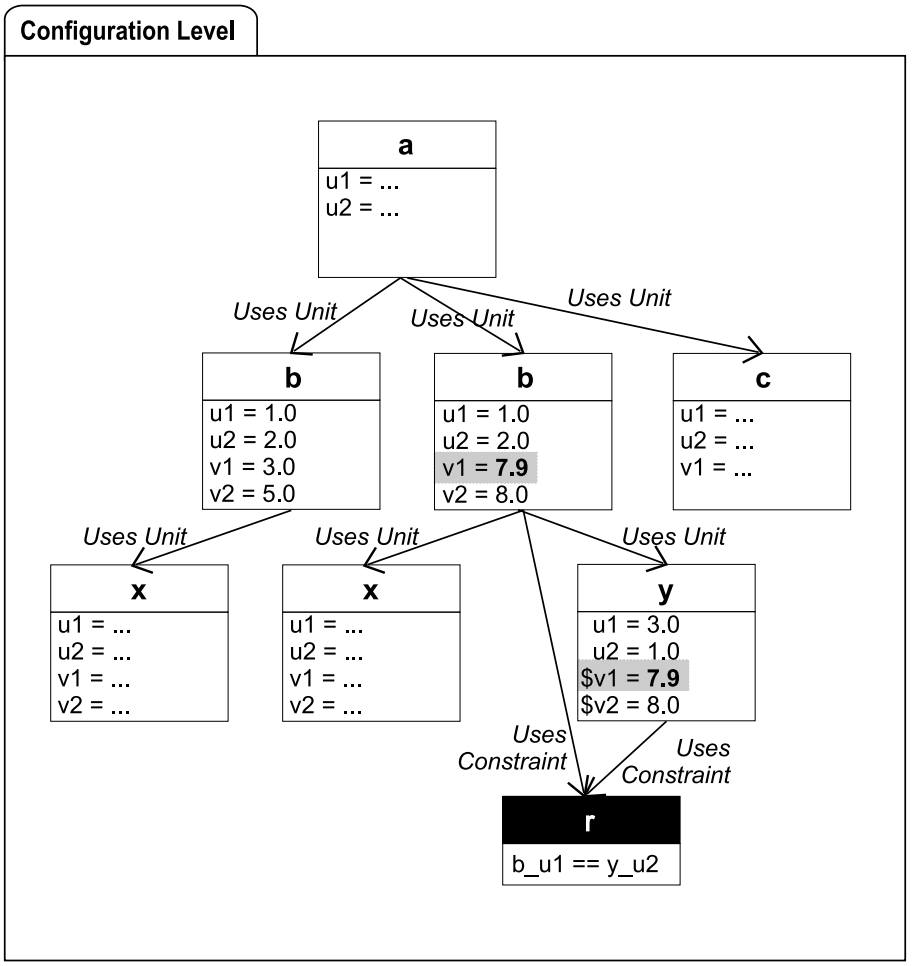

Abbildung 8.9: Der Wert des Attributs $v1$ in Element **y** wurde automatisch verändert.

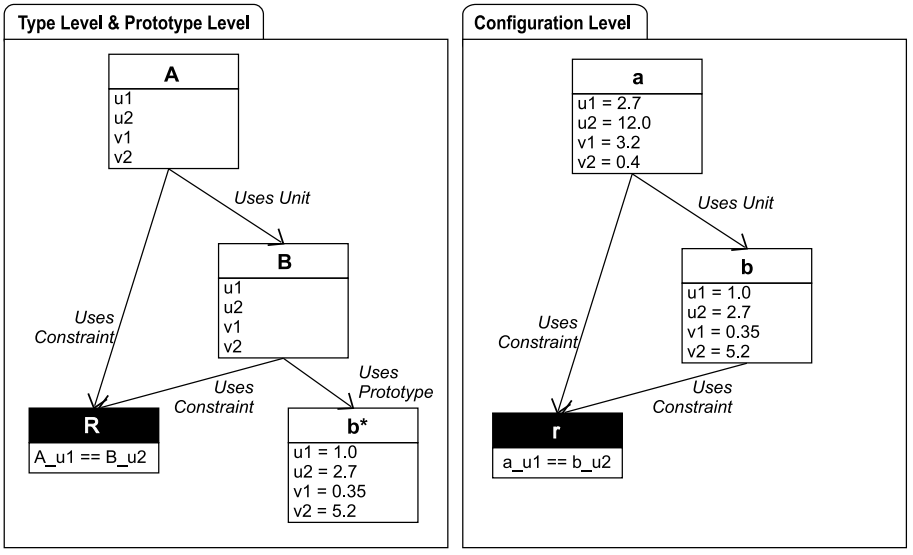

Abbildung 8.10: Regeln für Attributwerte

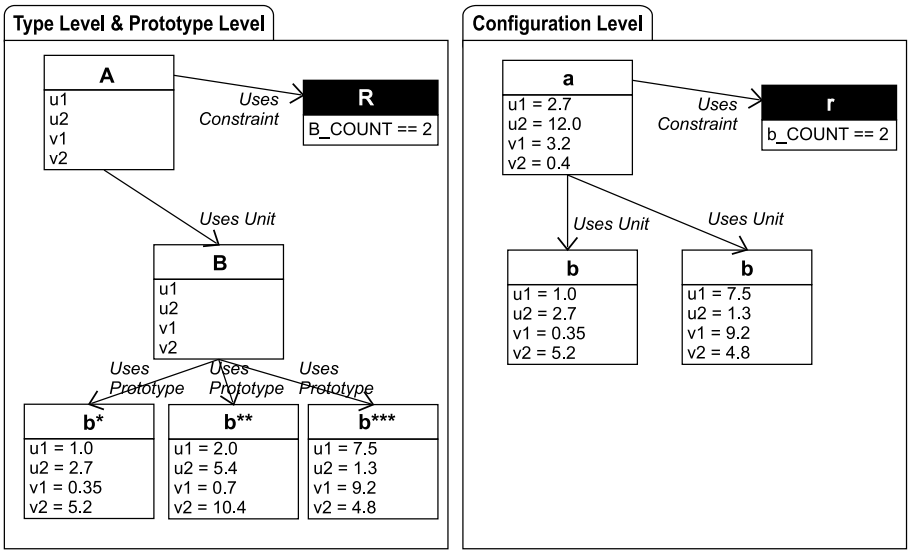

Abbildung 8.11: Regeln für Elemente

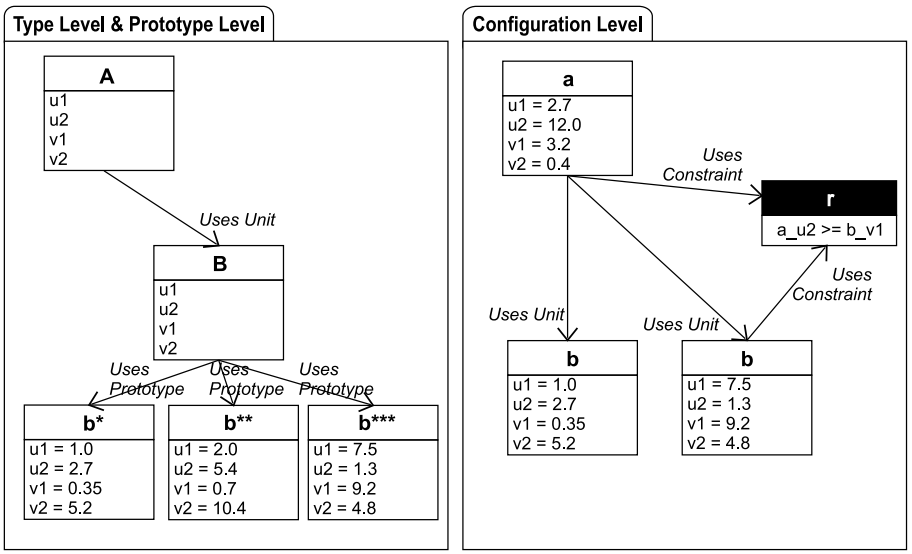

Abbildung 8.12: Benutzerspezifische Regel

Kapitel 9

Abbildung auf relationale Datenbanken

9.1 Allgemeines

Zur Speicherung großer Datenmengen stehen relationale Datenbanksysteme zur Verfügung. Derartige relationale Datenbanksysteme werden in Unternehmen seit Jahren eingesetzt und bilden dort beispielsweise die Grundlage von Client-Server-Geschäftsanwendungen [53]. Schöttner führt in [53] weiter aus, dass die relationalen Datenbanksysteme nach Jahrzehnten von Forschungs- und Entwicklungsarbeit einen hohen technischen Stand erreicht haben.

Die theoretische Grundlage für die relationalen Datenbanksysteme bildet die relationale Algebra. Die Erläuterungen in [25] können als Einführung in die relationale Algebra dienen. Die Standardabfragesprache für relationale Datenbanksysteme ist SQL[1]. Für eine Einführung in SQL, Grundlagen des Datenbank-Designs und des Entity Relationship Modells wird auf [51] verwiesen. Das Entity Relationship Modell wurde von Chen in [9] dargestellt.

In relationalen Datenbanksystemen werden Daten in Relationen gespeichert. Für den Begriff der Relation kann auch der Begriff der Tabelle verwendet werden.

Das beschriebene Planungssystem wurde in einer objektorientierten Programmiersprache umgesetzt. Die Elemente zur Repräsentation von Produktstrukturen und Gültigkeitsregeln sind im Arbeitsspeicher als Objekte realisiert, welche in Sammlungen zusammengefasst sind. Ziel der Abbildung ist es, die objektorientierte Struktur des Planungssystems auf ein relationales Datenbankschema abzubilden.

Zum Entwurf eines Datenbankschemas ist es sinnvoll, zunächst die Hauptgruppen zu den sogenannten *Entities* zusammenzufassen. In nächsten Schritt werden dann die *Beziehungen (Relationship)* der Entities zueinander festgelegt [51]. Die Darstellung der Entities und der Beziehungen erfolgt in sogenannten ER-Diagrammen.

[1] Structured Query Language

Bei der Ermittlung der Beziehungen der Entities untereinander können die in Abschnitt 7.3 von Kapitel 7 dokumentierten objektorientierten Assoziationen als eine Grundlage herangezogen werden. Dabei ist zu beachten, dass objektorientierte Assoziationen ein andere Semantik als relationale Beziehungen des Entitity Relationship Modells besitzen [50]. Die Beziehungen im Entitiy Relationship Modell werden bidirektional betrachtet während die objektorientierten Assoziationen nach [50] gerichtete Assoziationen sind. Insbesondere sind die in Abschnitt 7.3 dargelegten Beziehungen gerichtet.

9.2 Entities des Planungssystems

Wesentliche Entities des Planungssystems sind die grundlegenden Elemente

- Unit und
- Constraint,

welche in Abschnitt 7.2 dokumentiert sind. Die Units beinhalten wie in Tabelle 7.1 in Kapitel 7 dokumentiert eine Liste von Attributen, in der die produktmodellspezifischen Daten gespeichert werden. Die

- Attribute

stellen ebenfalls wesentliche Elemente dar, weshalb diese als Entity betrachtet werden.

Wie in Abschnitt 8.2 beschrieben, unterscheidet das Planungssystem die Bereiche Type Level, Prototype Level und Configuration Level. Diese Bereiche

- Type Level,
- Prototype Level und
- Configuration Level

werden als Entities berücksichtigt.

9.3 Beziehungen der Entities zueinander

9.3.1 Beziehungen zwischen Komponenten des Planungssystems und den Bereichen

Jede Unit gehört zu genau zu einem der drei möglichen Bereiche Type Level, Prototype Level oder Configuration Level. Jeder Constraint gehört zu genau einem der

Bereiche Type Level oder Configuration Level. Aus der Sicht der Bereiche sind jedem dieser Bereiche beliebig viele Units und Constraints zugeordnet. Die Beziehung der Bereiche zu den Units beziehungsweise den Constraints sind $1 : n$-Beziehungen. Abbildung 9.1 stellt diese Beziehungen dar.

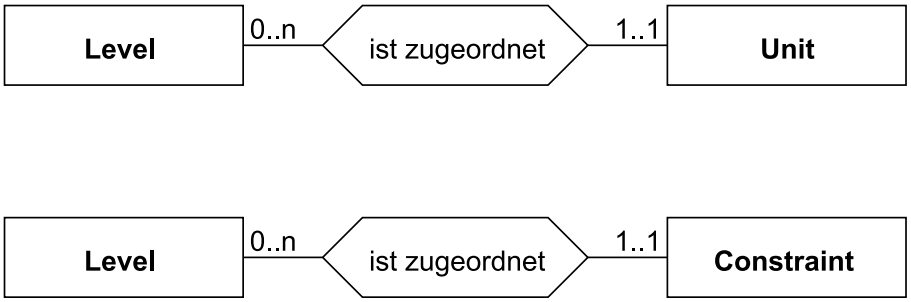

Abbildung 9.1: Beziehungen von Units und Constraints zu Bereichen

9.3.2 Beziehungen zwischen Units und Units

Die Units bilden im beschriebenen Planungssystem die Produktzusammensetzung ab. Ausgehend von genau einem Wurzelknoten ist diese Produktstruktur ein Baum. Eine Unit kann auf keine, eine oder beliebig viele weitere Units zeigen. Weiterhin können im beschriebenen Planungssystem Units von mehreren Wurzelknoten aus erreichbar sein. So können beispielsweise Units in mehreren Produktfamilien verwendet werden. Diese Situation zeigt Abbildung 7.7 in Kapitel 7. Eine Unit kann auch dann von mehreren Units referenziert werden, wenn diese nach Snapshot- und Revise-Operationen Element mehrerer Konfigurationen ist.

Zusammenfassend ergeben sich die folgenden Beziehungen von Units zueinander.

- Eine Unit zeigt auf keine, eine oder beliebig viele Units.
- Eine Unit wird von keiner, einer oder beliebig vielen Units referenziert.

Die Beziehung der Units zueinander ist eine $m : n$-Beziehung, welche in Abbildung 9.2 dargestellt ist.

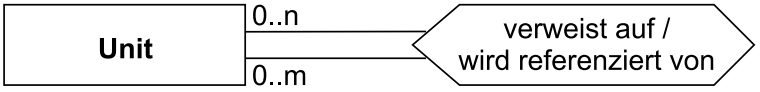

Abbildung 9.2: Beziehungen der Units zueinander

9.3.3 Beziehungen zwischen Units und Constraints

Eine Unit kann kann auf keine, eine oder beliebig viele Constraints verweisen. Für die Constraints gilt, dass diese immer von mindestens einer Unit oder beliebig vielen Units referenziert werden.

Die Beziehungen der Units zu den Constraints lassen sich wie folgt zusammenfassen.

- Innerhalb einer Konfiguration zeigt eine Unit auf keine, eine oder beliebig viele Constraints.
- Ein Constraint kann von mehreren Units referenziert werden.
- Ein Constraint wird von mindestens einer Unit referenziert.

Die Beziehung von Units zu Constraints ist vom Typ $m : n$. Existiert ein Constraint, so muss mindestens eine Unit existieren, die auf diesen Constraint verweist. Die $m : n$-Beziehung ist in Abbildung 9.3 dargestellt.

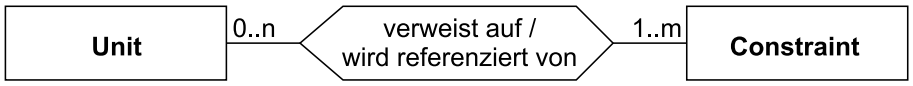

Abbildung 9.3: Beziehungen der Units zu den Constraints

9.3.4 Beziehungen zwischen Units und ihren Attributen

Mit dem beschriebenen Planungssystem sollen eine Vielzahl von Produktstrukturen abgebildet werden können. Dies wird auch dadurch erreicht, dass Attribute einer Unit zur Laufzeit des Programms deklariert und initialisiert werden können. Hierfür werden die einer Unit zugehörigen Attribute zur Repräsentation von Produkteigenschaften zur Laufzeit des Planungssystems in einer Sammlung zusammengefasst. Diese Sammlung ist Element des Unit-Objekts. Für die Beziehung zwischen Units und Attributen ergeben sich die folgenden Aussagen.

- Eine Unit besitzt keine, eine oder beliebig viele Attribute.
- Ein Attribut gehört zu genau einer Unit. Existiert ein Attribut, so existiert auch genau eine Unit, die dieses Attribut referenziert.

Die in Abbildung 9.4 dargestellte Beziehung der Unit zu den Attributen ist vom Typ $1 : n$.

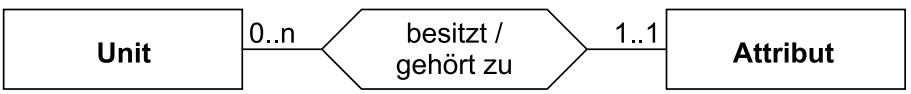

Abbildung 9.4: Beziehungen der Units zu den Attributen

9.3.5 Beziehungen zwischen Units und ihren Prototypen

Falls die Elemente des Prototype Level in den Type Level integriert werden, muss auch die Beziehung zwischen den entsprechenden Elementen berücksichtigt werden. Abbildung 8.3 in Kapitel 8 zeigt, wie die Elemente des Prototype Level in den Type Level integriert werden können. Dabei existieren zwei Möglichkeiten zur Integration.

1. Erweiterung der Units um den Verknüpfungstyp *UsesPrototype* zusätzlich zu den *UsesUnit*- und *UsesConstraint*-Links. Hierbei zeigen die neuen Links auf Units.

2. Ableitung der Prototype Units von den Units durch Vererbung und Verknüpfung dieser Prototype Units mit der zugehörigen Units über *UsesUnit*-Links

Wie in Kapitel 8 beschrieben, dienen die Elemente des Prototype Level der Zusammenstellung aller möglichen Ausprägungen der im Type Level definierten Produktkomponenten. Diese Produktkomponenten werden im Type Level mit Units repräsentiert. Entsprechend sind die zugeordneten Komponenten des Prototype Level ebenfalls Units.

Die Beziehungen von Units zu Prototypen ist von den beiden Möglichkeiten unabhängig und kann wie folgt zusammengefasst werden.

- Eine Unit des Type Level zeigt auf keine, eine oder beliebig viele Prototypen.
- Innerhalb einer Konfiguration gehört ein Prototyp zu genau einer Unit.
- Konfigurationsübergreifend können beliebig viele Units den selben Prototyp referenzieren.
- Nur Units des Type Level können auf Prototypen verweisen.
- Existiert ein Prototyp, so muss mindestens eine Unit existieren, die diesen Prototyp referenziert.

Die Beziehung von Units zu Prototypen ist vom Typ $m : n$ und ist in Abbildung 9.5 dargestellt. Da die Prototypen durch Units repräsentiert werden, ergibt sich resultierend die in Abbildung 9.5 unten dargestellte Beziehung.

9.3.6 Zusammenfassung

Abbildung 9.6 zeigt eine mögliche Zusammenfassung der Beziehungen der Komponenten zueinander.

Hierbei ist anzumerken, dass folgende Zusatzbedingungen eingehalten werden müssen.

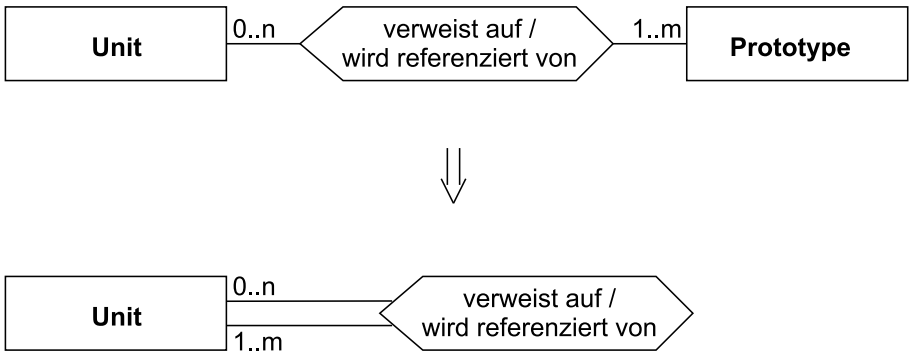

Abbildung 9.5: Beziehungen der Units zu den Prototypen

- Eine Unit im Prototype Level verweist auf keine weiteren Units.
- Eine Unit im Prototype Level verweist nicht auf Constraints.
- Ein Constraint ist nur dem Type Level oder dem Configuration Level zugeordnet.

9.4 Datenbankentwurf

Ausgehend von den ER-Modellen ergibt sich beispielsweise der folgende Entwurf einer relationalen Datenbank.

9.4.1 Bereiche

Die drei Bereiche Type Level, Prototype Level und Configuration Level werden in einer Tabelle *T_Level* entsprechend Tabelle 9.1 gespeichert.

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel, ID der Unit
$LEVELNAME$	Text	Bezeichnung des Levels

Tabelle 9.1: Tabelle T_Level

Die Tabelle T_Level besitzt genau drei Einträge (Tabelle 9.2).

9.4.2 Units

Alle Units können in der Tabelle *T_Unit* gespeichert werden (Tabelle 9.3). Dabei wird zusätzlich zu den Attributen der Unit-Objekte ($ID, NAME, DESCR, VP, FX$) gespeichert, ob die Unit ein Element des Type Level, des Prototype Level oder des

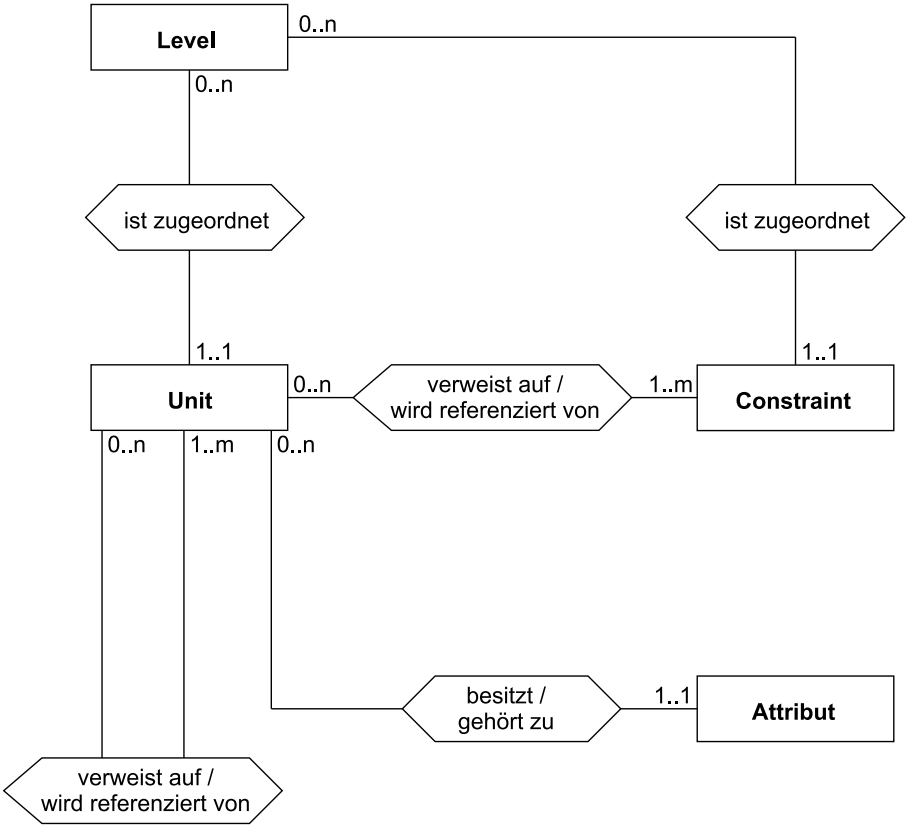

Abbildung 9.6: Übersicht über alle Beziehungen

ID	*LEVELNAME*
1	Type Level
2	Prototype Level
3	Configuration Level

Tabelle 9.2: Datensätze der Tabelle T_Level

Configuration Level ist. Dies erfolgt durch das Attribut $LEVEL$. Die Attribute der Units dokumentiert Tabelle 7.1 in Kapitel 7. Die Attribute (AL, NUU, NUC, NUP) werden durch eigene Tabellen abgebildet.

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel, ID der Unit
$NAME$	Text	Name der Unit
$DESCR$	Text	Beschreibung
VP	Ganzzahl	ID des Versionsvorgängers
FX	Boolean	Fixed flag
$LEVEL$	Ganzzahl	Fremdschlüssel, ID des Levels

Tabelle 9.3: Tabelle T_Unit

9.4.3 Constraints

Die Constraints werden in einer Tabelle *T_Constraint* gemäss Tabelle 9.4 gepeichert. Analog zur Tabelle *T_Unit* erfolgt eine Speicherung der Zugehörigkeit zu Type Level beziehungsweise Configuration Level durch das Attribut $LEVEL$. Die Attribute der Constraint-Objekte sind in Tabelle 7.3 in Kapitel 7 dokumentiert.

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel, ID des Constraint
$DESCR$	Text	Beschreibung
$EXPR$	Text	Ausdruck zur Repräsentation der Regel
VP	Ganzzahl	ID des Versionsvorgängers
FX	Boolean	Fixed flag
$LEVEL$	Ganzzahl	Fremdschlüssel, ID des Levels

Tabelle 9.4: Tabelle T_Constraint

9.4.4 Attribute

Die Unit-Objekte beinhalten eine Liste AL von Attributen zur Speicherung der produktmodellspezifischen Daten. Die Tabelle *T_Attribut*, Tabelle 9.5 speichert die zu den Units gehörenden Attribute. Dabei wird bei der Abbildung der Attribute unterschieden, ob es sich bei dem Attributwert um eine Zahl oder um einen Text handelt.

9.4.5 Speicherung der Objektverweise

Abschliessend sind die Beziehungen

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel, ID des Attributs
UNIT_ID	Ganzzahl	Fremdschlüssel, ID der Unit
NAME	Text	Attributname
VALUE	Gleitkommazahl	Wert des Attributs
STRINGVALUE	Text	Wert des Attributs
ISKEY	Wahr/Falsch	Wahr, falls Schlüsselattribut

Tabelle 9.5: Tabelle T_Attribut

- von Units untereinander,
- von Units zu Constraints und
- von Units zu Prototypen.

zu speichern. Wie beschrieben sind diese Beziehungen vom Typ $m : n$. Die Abbildung von $m : n$-Relationen erfolgt über jeweils eine zusätzliche Tabelle. Tabelle 9.6 dient der Abbildung der Beziehungen der Units untereinander. Die Tabelle 9.7 bildet die Beziehungen zwischen Units und Constraints und die Tabelle 9.8 bildet die Beziehungen zwischen Units und Prototypen ab.

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel
PARENT_ID	Ganzzahl	Fremdschlüssel, ID des Vaterknotens
CHILD_ID	Ganzzahl	Fremdschlüssel, ID des Kindknotens

Tabelle 9.6: Tabelle T_UnitUsesUnit

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel
UNIT_ID	Ganzzahl	Fremdschlüssel, ID der Unit
CONSTRAINT_ID	Ganzzahl	Fremdschlüssel, ID des Constraints

Tabelle 9.7: Tabelle T_UnitUsesConstraint

Feldname	Datentyp	Beschreibung
ID	Ganzzahl	Primärschlüssel
UNIT_ID	Ganzzahl	Fremdschlüssel, ID der Unit
PROTOTYPE_ID	Ganzzahl	Fremdschlüssel, ID des Prototypen

Tabelle 9.8: Tabelle T_UnitUsesPrototype

Kapitel 10

Beispiele für die Anwendung des Planungssystems in der Bauindustrie

10.1 Beispiel – Modellbasierter CAD-Entwurf

Im Rahmen der Planung von Bauvorhaben existieren durch Normen und Regelungen festgelegte Eigenschaften. Hierzu zählen beispielsweise

- Bauteilmindestabmessungen (Mindestdurchmesser von Stützen aus Stahlbeton),
- Materialeigenschaften (Verwendung von Stahl mit einer Mindestzugfestigkeit),
- Anforderungen an die bauliche Durchbildung (Anforderungen an den Wärmeschutz).

Die Einhaltung dieser festgelegten Eigenschaften kann durch Gültigkeitsregeln überprüft werden, die sich auf Attributwerte von Elementen beziehen.

10.1.1 Planungsaufgabe

In diesem Beispiel wird gezeigt, wie Anforderungen an den Wärmeschutz beim Entwurf eines Wohnhauses in einem Entwurfssystem definiert werden können und wie nach einem Entwurfsschritt die Einhaltung der Anforderungen programmautomatisch überprüft werden kann.

Nach [18] sind die maximalen U-Werte U_{max} der Tabelle 10.1 vorgeschrieben.

In einem ersten Schritt wird ein Modell für ein Wohnhaus angelegt. Die U_{max}-Werte haben in Bezug auf das geplante Bauvorhaben den Charakter von globalen Variablen. Sie werden als Eigenschaften des Elements **Haus** festgelegt. Abbildung 10.1

Bauteil	Aussenwände	Fenster	Türen	Dächer	Kellerdecken
$U_{max}[W/m^2K]$	$0,45$	$1,7$	$2,9$	$0,30$	$0,40$

Tabelle 10.1: Anforderungen an den baulichen Wärmeschutz aus [18]

zeigt das Modell für ein Haus im Type Level, Abbildung 10.2 zeigt die definierten U_{max}-Werte.

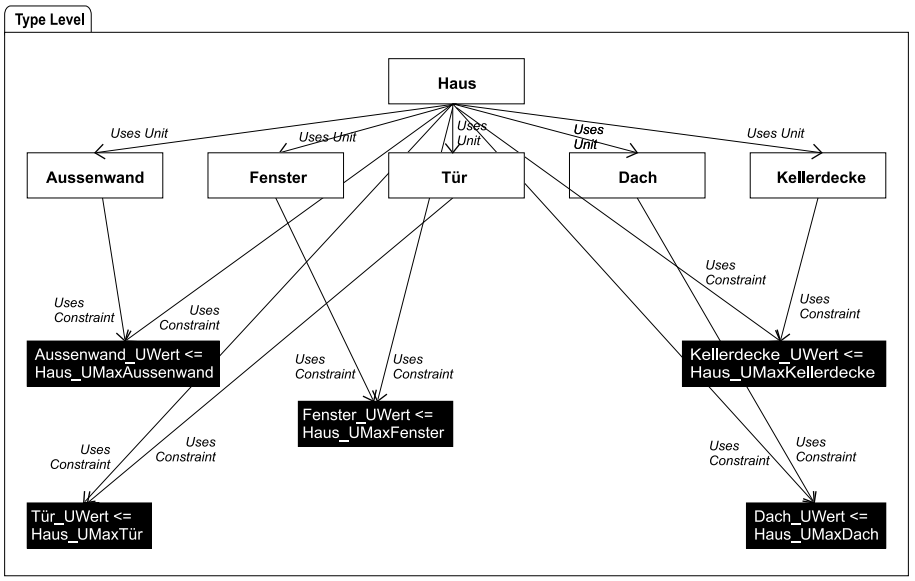

Abbildung 10.1: Haus-Modell im Type Level mit Gültigkeitsregeln

10.1.2 Erster Planungsschritt

In einem ersten Planungsschritt wird im Configuration Level eine Instanz des Elements **Haus** im Configuration Level erzeugt (Abbildung 10.3). Da später dieser Zustand jederzeit verfügbar sein soll, wird die Operation *Snapshot()* auf das Element **Haus, V1.0** angewendet. Anschliessend existiert das Haus als unveränderliche Version **Haus, V1.0** und als bearbeitbare Version **Haus, V1.1**, an der weitergearbeitet werden kann (Abbildung 10.4). Da im Entwurfssystem nur an der bearbeitbaren Version weitergearbeitet werden kann, wird nur diese angezeigt.

10.1.3 Zweiter Planungsschritt

Im nächsten Planungsschritt fügt ein Konstrukteur eine Aussenwand ein. Das Entwurfssystem fügt zum einen das Element **Aussenwand, V1.0** zur Repräsentation einer Aussenwand und zum anderen ein Element zur Repräsentation der Gültigkeitsregel für den $U-$Wert ein.

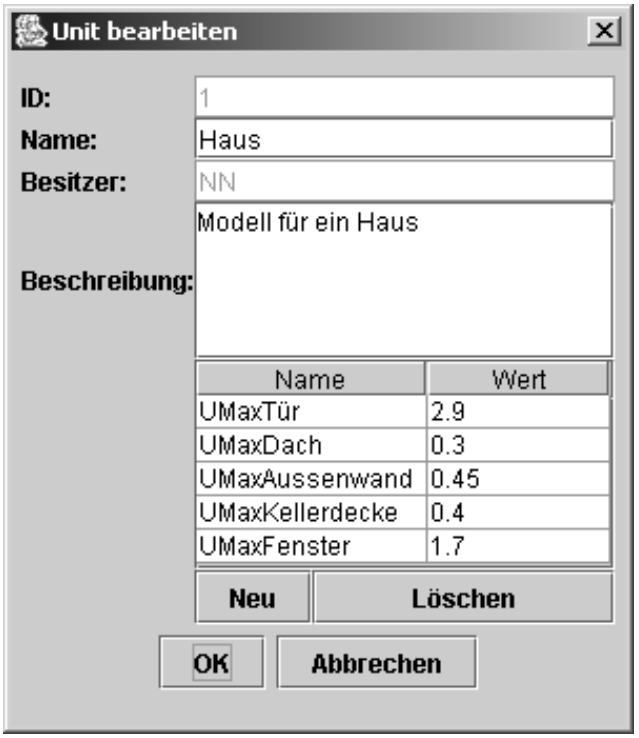

Abbildung 10.2: Definierte Maximalwerte im Element **Haus**

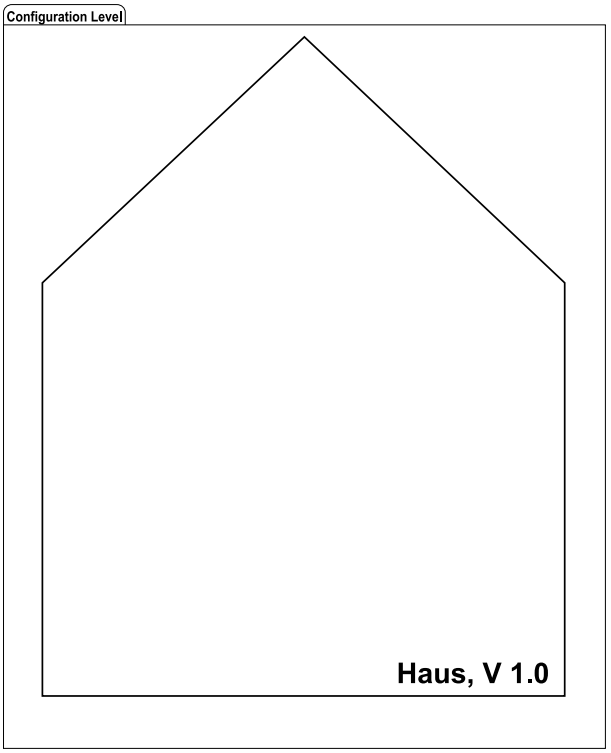

Abbildung 10.3: **Haus, V1.0**

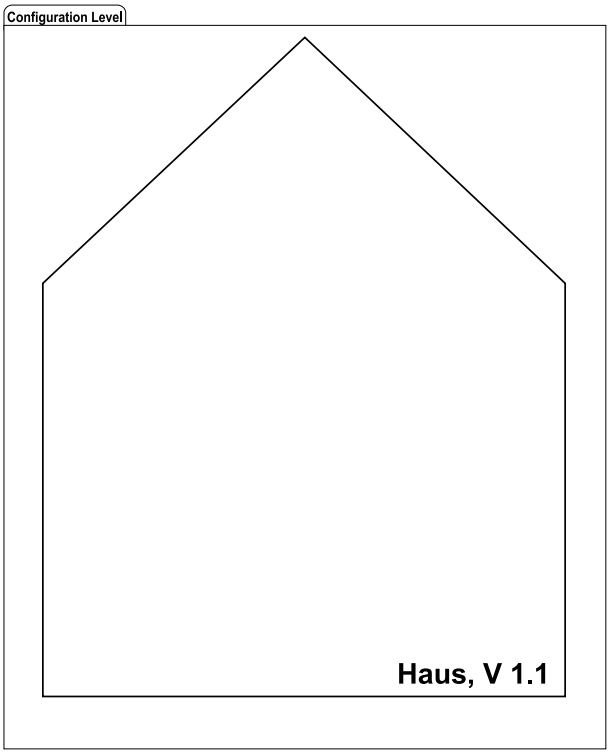

Abbildung 10.4: Die bearbeitbare Komponente **Haus, V1.1**

Weiterhin kann das Entwurfssystem die Verbindung von **Aussenwand, V1.0** zur Gültigkeitsregel und die Verbindung von **Aussenwand, V1.0** mit **Haus, V1.1** anlegen (Abbildung 10.5).

10.1.4 Überprüfung der aktuellen Konfiguration

Der $U-$Wert als Eigenschaft der Wand wird entweder

- vom Entwurfssystem aus den Materialeigenschaften und den Abmessungen ermittelt oder er
- wird vom Benutzer durch Auswahl eines Elements des Prototype Level festgelegt oder er
- wird durch den Benutzer direkt angegeben.

Damit sind alle in der angelegten Gültigkeitsregel enthaltenen Werte bekannt und die Bedingung kann überprüft werden. Abbildung 10.6 zeigt eine Programmausgabe dieser Überprüfung.

Für die Bedingung `Aussenwand_UWert <= Haus_UMaxAussenwand` wurde der Wert Eins ermittelt, es handelt sich um eine wahre Aussage. Die Bedingung ist erfüllt.

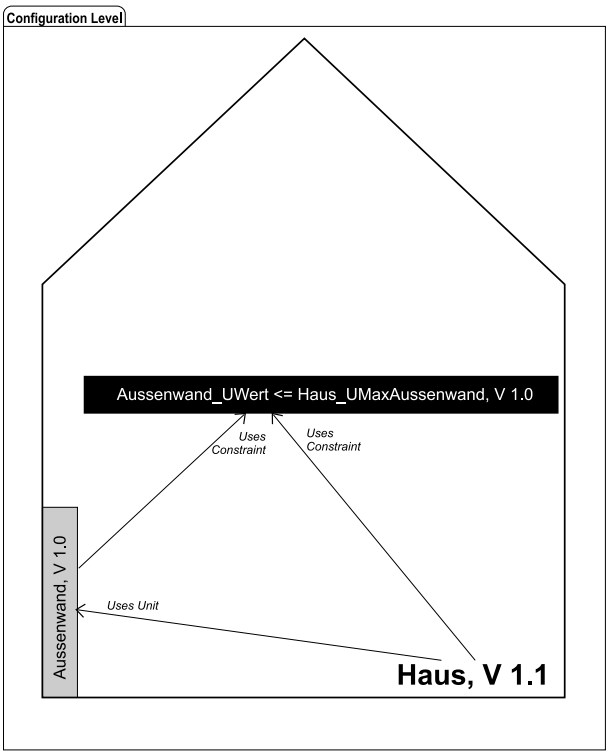

Abbildung 10.5: Hinzugefügte **Aussenwand, V1.0** und die Gültigkeitsregel für den $U-$Wert

```
Checking XS(Haus, Version 1.1)
Initialising parser for Aussenwand_UWert <= Haus_UMaxAussenwand
Aussenwand_UWert=0.45
Aussenwand_d=24.0
Haus_UMaxAussenwand=0.45
Haus_UMaxKellerdecke=0.40
Haus_UMaxDach=0.30
Haus_UMaxFenster=1.7
Haus_UMaxTür=2.9
Result for Aussenwand_UWert <= Haus_UMaxAussenwand: 1.0
```

Abbildung 10.6: Programmausgabe bei der Überprüfung der Constraints

10.2 Beispiel – CAD-Entwurf ohne Modell

Die Anwendung des Planungssystems zur Unterstützung bei der Entwurfsphase ist auch ohne ein vorab vorhandenes Modell im Type Level möglich. Dabei werden entsprechend einer gewählten Konstruktion die relevanten Gültigkeitsregeln im Rahmen der Bearbeitung eingegeben.

Damit zeigt das Beispiel die Verwendung von benutzerspezifischen Regeln, die während des Entwurfsprozesses im Configuration Level eingegeben werden.

10.2.1 Planungsaufgabe

Im Rahmen des Beispiels soll ein Rahmen entwickelt werden. Um das Beispiel einfach zu halten, werden nur die folgenden Randbedingungen festgelegt.

	Minimum	Maximum
Rahmenhöhe	$hrMin = 3,0m$	$hrMax = 4,0m$
Rahmenbreite	$brMin = 7,0m$	$brMax = 8,0m$

Tabelle 10.2: Geometrische Randbedingungen

In einem ersten Planungsschritt wird der Rahmen als Unit-Element im System angelegt und die minimalen und maximalen Abmessungen werden als Attribute des Elements eingegeben (Bild 10.7).

Zu diesem Zeitpunkt gibt es nur dieses Element im Configuration Level. Da später dieser Zustand jederzeit wieder verfügbar sein soll, wird die Operation *Snapshot()* auf das Element angewendet. Anschliessend existiert der Rahmen als unveränderliche Version **Rahmen, V1.0** und als aktive Version **Rahmen, V1.1**, an der weitergearbeitet werden kann (Bild 10.8).

10.2.2 Designentscheidung

In einem Planungsschritt entscheidet sich ein Bearbeiter für die Realisierung des Rahmens durch eine Konstruktion mit einem Riegel und zwei Stielen (Abbildung 10.9).

Der Riegel und die Stiele werden als Unit-Elemente mit einer Länge l und einer Profilhöhe h abgebildet. Tabelle 10.3 zeigt die Elemente, ihre eindeutigen Bezeichner, mit dem sie im System eingegeben werden und die zugehörigen Abmessungen, die als Attribute eingegeben werden. Anschliessend werden diese Elemente mit dem **Rahmen, V1.1** verknüpft, womit sich der in Bild 10.10 dargestellte Zustand ergibt. Die Graphendarstellung in Bild 10.10 zeigt den **Rahmen, V1.1** und seine Elemente, aus denen er besteht. Weiterhin dargestellt ist der Versionsvorgänger **Rahmen, V1.0**.

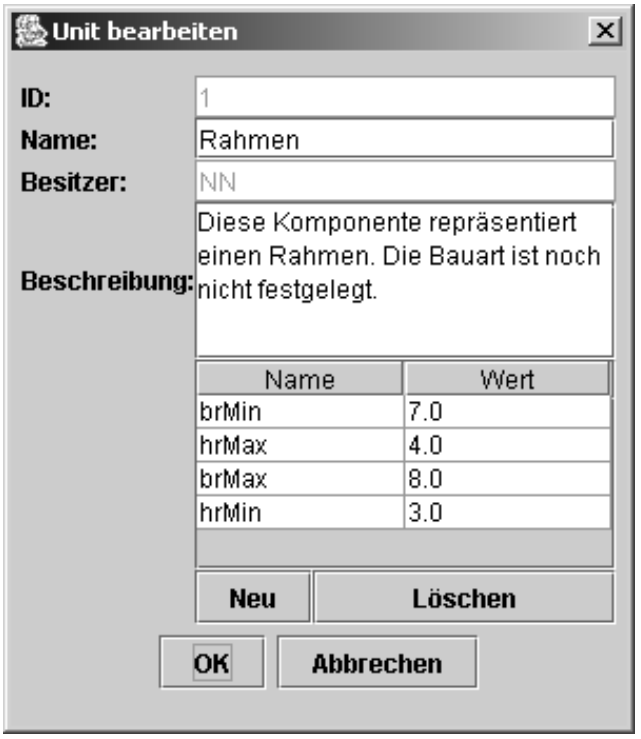

Abbildung 10.7: Eingabe der Elementattribute

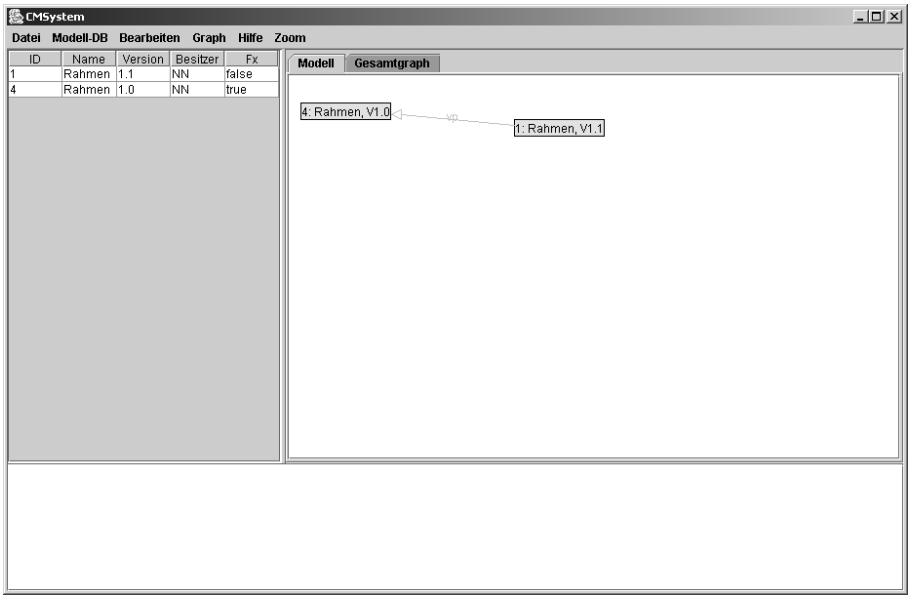

Abbildung 10.8: Bearbeitbarer Rahmen und sein Versionsvorgänger

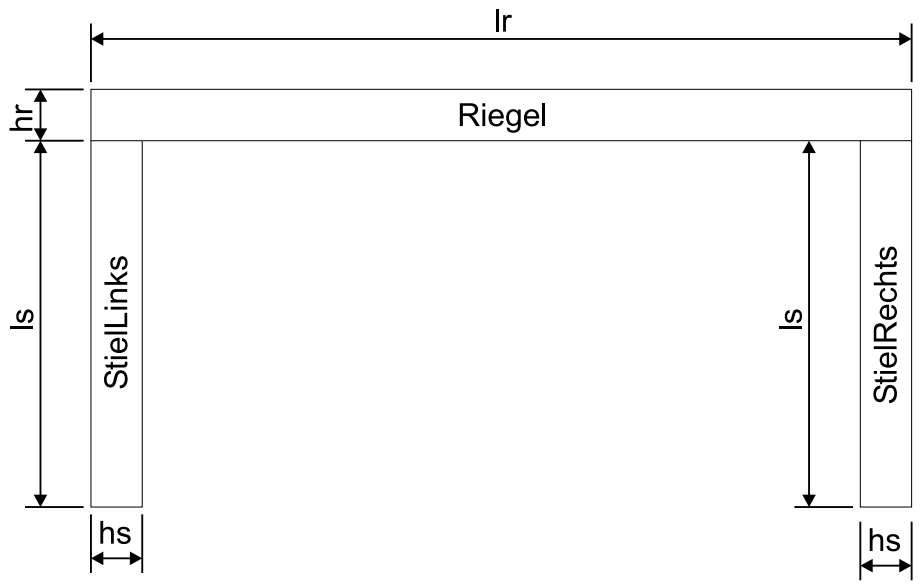

Abbildung 10.9: Rahmen aus einem Riegel und zwei Stielen

Komponente	Riegel	Stiel links	Stiel rechts
Eindeutiger Bezeichner	Riegel	StielLinks	StielRechts
Länge	$lr = 7,5m$	$ls = 3,5m$	$ls = 3,5m$
Höhe	$hr = 0,3m$	$hs = 0,2m$	$hs = 0,15m$

Tabelle 10.3: Abmessungen des Riegels und der Stiele

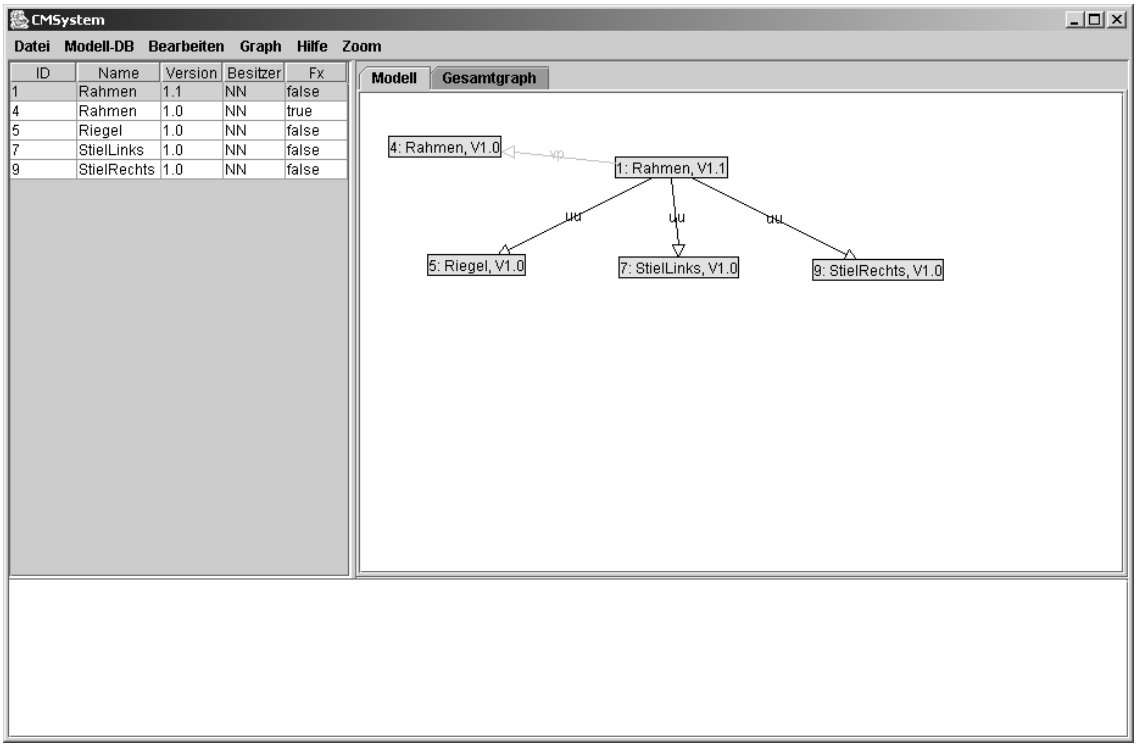

Abbildung 10.10: Graph des Configuration Level nach Designentscheidung

10.2.3 Formulierung der geometrischen Zusammenhänge

Die Länge des Riegels ist gleichzeitig die Breite des Rahmens. Damit folgen die Bedingungen

$$Riegel\_lr \geq Rahmen\_brMin \tag{10.1}$$

und

$$Riegel\_lr \leq Rahmen\_brMax\,. \tag{10.2}$$

Die Rahmenhöhe setzt sich aus der Länge des Stiels und der Höhe des Riegels zusammen. Man erhält die Bedingungen

$$\begin{aligned} Riegel\_hr + StielLinks\_ls &\geq Rahmen\_hrMin\,, && (10.3)\\ Riegel\_hr + StielLinks\_ls &\leq Rahmen\_hrMax\,, && (10.4)\\ Riegel\_hr + StielRechts\_ls &\geq Rahmen\_hrMin\,, && (10.5)\\ Riegel\_hr + StielRechts\_ls &\leq Rahmen\_hrMax\,. && (10.6) \end{aligned}$$

Zur Wahrung der Übersichtlichkeit werden nur die Bedingungen 10.1 bis 10.4 als Constraint-Elemente in das System eingegeben. Die Eingabe der Bedingungen zeigen die Bilder 10.11 bis 10.14.

Nach Eingabe der Bedingungen werden diese mit den entsprechenden Unit-Elementen verknüpft.

Abbildung 10.11: Bedingung 10.1

Abbildung 10.12: Bedingung 10.2

Abbildung 10.13: Bedingung 10.3

Abbildung 10.14: Bedingung 10.4

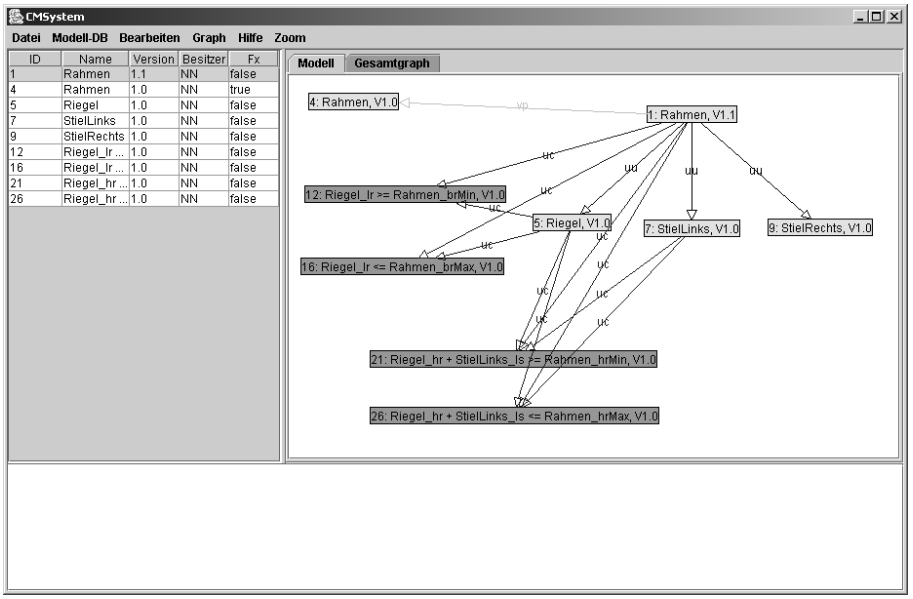

Abbildung 10.15: Vollständige Darstellung aller Elemente

10.2.4 Darstellung des Gesamtsystems

Bild 10.15 zeigt alle im System vorhandenen Elemente.

Die Konfiguration von **Rahmen, V1.1** besteht aus den Unit-Elementen **Riegel, V1.0**, **StielLinks, V1.0** und **StielRechts, V1.0**. Diese *Besteht-aus-Beziehungen* werden mit Pfeilen mit der Beschriftung *uu* für *Uses Unit* gezeichnet. Weiterhin beinhaltet die Konfiguration von **Rahmen, V1.1** die Versionen 1.0 der genannten Constraint-Elemente. Die *Benutzt-Constraint-Beziehungen* werden mit Pfeilen mit der Beschriftung *uc* für *Uses Constraint* gezeichnet. gezeichnet. Ebenfalls dargestellt ist **Rahmen, V1.0**, der Versionsvorgänger von **Rahmen, V1.1**. Die *Versionsvorgänger-Beziehungen* werden mit Pfeilen mit der Beschriftung *vp* für *Version Predecessor* gezeichnet.

10.2.5 Überprüfung der aktuellen Konfiguration des Rahmens

Die Constraint-Elemente definieren die geometrischen Abhängigkeiten zwischen den Bauteilen. Durch Auswertung der Constraint-Elemente wird überprüft, ob die gewählten Abmessungen für Riegel und StielLinks die Anforderungen an den Rahmen, dessen Bestandteil sie sind, erfüllen. Die Auswertung jedes Constraint-Elements muss dann eine wahre Aussage ergeben. Abbildung 10.16 zeigt die Programmausgabe der Überprüfung.

Mit den als Elementeigenschaften eingegebenen Abmessungen werden alle formulierten Abhängigkeiten erfüllt.

```
Checking XS(Rahmen, Version 1.1)

Initialising parser for Riegel_lr >= Rahmen_brMin
Riegel_lr=7.5
Riegel_hr=0.3
Rahmen_brMax=8.0
Rahmen_hrMax=4.0
Rahmen_brMin=7.0
Rahmen_hrMin=3.0
Result for Riegel_lr >= Rahmen_brMin: 1.0

Initialising parser for Riegel_lr <= Rahmen_brMax
Riegel_lr=7.5
Riegel_hr=0.3
Rahmen_brMax=8.0
Rahmen_hrMax=4.0
Rahmen_brMin=7.0
Rahmen_hrMin=3.0
Result for Riegel_lr <= Rahmen_brMax: 1.0

Initialising parser for Riegel_hr + StielLinks_ls >= Rahmen_hrMin
Riegel_lr=7.5
Riegel_hr=0.3
StielLinks_ls=3.5
StielLinks_hs=0.2
Rahmen_brMax=8.0
Rahmen_hrMax=4.0
Rahmen_brMin=7.0
Rahmen_hrMin=3.0
Result for Riegel_hr + StielLinks_ls >= Rahmen_hrMin: 1.0

Initialising parser for Riegel_hr + StielLinks_ls <= Rahmen_hrMax
Riegel_lr=7.5
Riegel_hr=0.3
StielLinks_ls=3.5
StielLinks_hs=0.2
Rahmen_brMax=8.0
Rahmen_hrMax=4.0
Rahmen_brMin=7.0
Rahmen_hrMin=3.0
Result for Riegel_hr + StielLinks_ls <= Rahmen_hrMax: 1.0
```

Abbildung 10.16: Programmausgabe bei der Überprüfung der Constraints

10.3 Beispiel – Kooperation im Bauplanungsprozess

10.3.1 Planungsmodell für Stahlhallen

Abbildung 10.17 zeigt ein Modell für die Planung der Stahlhalle. Da im Modell nur eine Konfiguration enthalten ist, werden keine Versionsnummern dargestellt. Das Modell besteht aus den folgenden Planungselementen.

HS: Hauptanforderungen an Stahlhallen.

NBH: Nachweis und Bemessung der Hauptkonstruktion.

NBA: Nachweis und Bemessung der Anschlüsse.

ZG: Zeichnungen von Übersichten und Details.

Der Nachweis und die Bemessung der Hauptkonstruktion **NBH** schließt sich der Definition der Hauptanforderungen **HS** an. Typisierte Nachweise vorausgesetzt existiert eine Menge von Gültigkeitsregeln $\mathbf{R}_i$ zwischen den Elementen **HS** und **NBH**. Die Notatation *(Uses Constraint)** repräsentiert die Menge an Uses Constraint-Links zu den Gültigkeitsregeln der Regelmengen.

Dem Nachweis und der Bemessung der Hauptkonstruktion folgen der Nachweis und die Bemessung der Anschlüsse **NBA**. Auch hier sind die Nachweise typisiert und es existiert eine Menge von Gültigkeitsregeln $\mathbf{R}_k$ zwischen den Elementen.

Den Abschluss der Planung bildet die Zeichnung von Übersichten und Details **ZG** . Übersichten können gegebenenfalls bereits vor dem Nachweis der Anschlüsse **NBA** erfolgen. Daher existiert eine Uses Unit-Verbindung zwischen **NBH** und **ZG**. Detailzeichnungen erfordern die Bemessung der Anschlüsse, weshalb zusätzlich eine Uses Unit-Verbindung zwischen **NBA** und **ZG** existiert. Die Mengen an Gültigkeitsregeln $\mathbf{R}_l$ und $\mathbf{R}_m$ prüfen die Abhängigkeiten zwischen den Elementen.

Methoden zum Aufbau des Graphen im Configuration Level

Das Modell für den Planungsprozess unterscheidet sich von den Modellen zur Abbildung von Produktzusammensetzungen.

- Der Subgraph der Planungselemente ohne Berücksichtigung der Regeln ist selbst ein gerichteter azyklischer Graph.
- Jedes Element des Type Level soll nach Abschluss der Planung genau ein Mal im Configuration Level enthalten sein.

Der Aufbau der Konfiguration im Configuration Level kann durch modifizierte Versionen der Algorithmen 8.1 und 8.2 erfolgen. Diese sind als Algorithmen 10.1 und 10.2 dargestellt.

Algorithmus 10.1 Add(*configurationLevel*: cmElementCollection, *typeLevel*: cmElementCollection, *unitInTypeLevel*: unit, *rootInConfigurationLevel*: unit, *rootInTypeLevel*: unit):cmElementCollection

```
1: SET of units fathersInTypeLevel := GetFather(unitInTypeLevel,
   rootInTypeLevel, typeLevel);
2: if fathersInTypeLevel = NULL then
3:    instanceUnit := newUnit();
4:    SetAttributes(instanceUnit, GetAttributes(unitInTypeLevel));
5:    instanceUnit.NAME := unitInTypeLevel.NAME;
6:    ConnectConstraints(unitInTypeLevel, instanceUnit, rootInTypeLevel,
      rootInConfigurationLevel, typeLevel, configurationLevel);
7:    RETURN (configurationLevel ∪ {instanceUnit});
8: else
9:    for all units father ∈ fathersInTypeLevel do
10:      fatherName := father.NAME;
11:      for all units x ∈ configurationLevel do
12:         if (x ∈ XS(rootInConfigurationLevel)) ∧ (x.NAME = fatherName)
            then
13:            instanceUnit = newUnit();
14:            SetAttributes(instanceUnit, GetAttributes(unitInTypeLevel));
15:            instanceUnit.NAME := unitInTypeLevel.NAME;
16:            x.NUU := x.NUU ∪ {instanceUnit};
17:            ConnectConstraints(unitInTypeLevel, instanceUnit,
               rootInTypeLevel, rootInConfigurationLevel, typeLevel,
               configurationLevel);
18:            RETURN (configurationLevel ∪ {instanceUnit});
19:         end if
20:      end for
21:   end for
22: end if
```

Algorithmus 10.2 GetFather(*u*: unit, *root*: unit, *cmElementCollection*: cmElementCollection): SET of units

```
1: SET of units fatherUnits;
2: for all units x ∈ cmElementCollection do
3:    if (x ∈ XS(root)) ∧ (u ∈ x.NUU) then
4:       fatherUnits := fatherUnits ∪ {x};
5:    end if
6: end for
7: if fatherUnits = {} then
8:    RETURN NULL;
9: else
10:   RETURN(fatherUnits);
11: end if
```

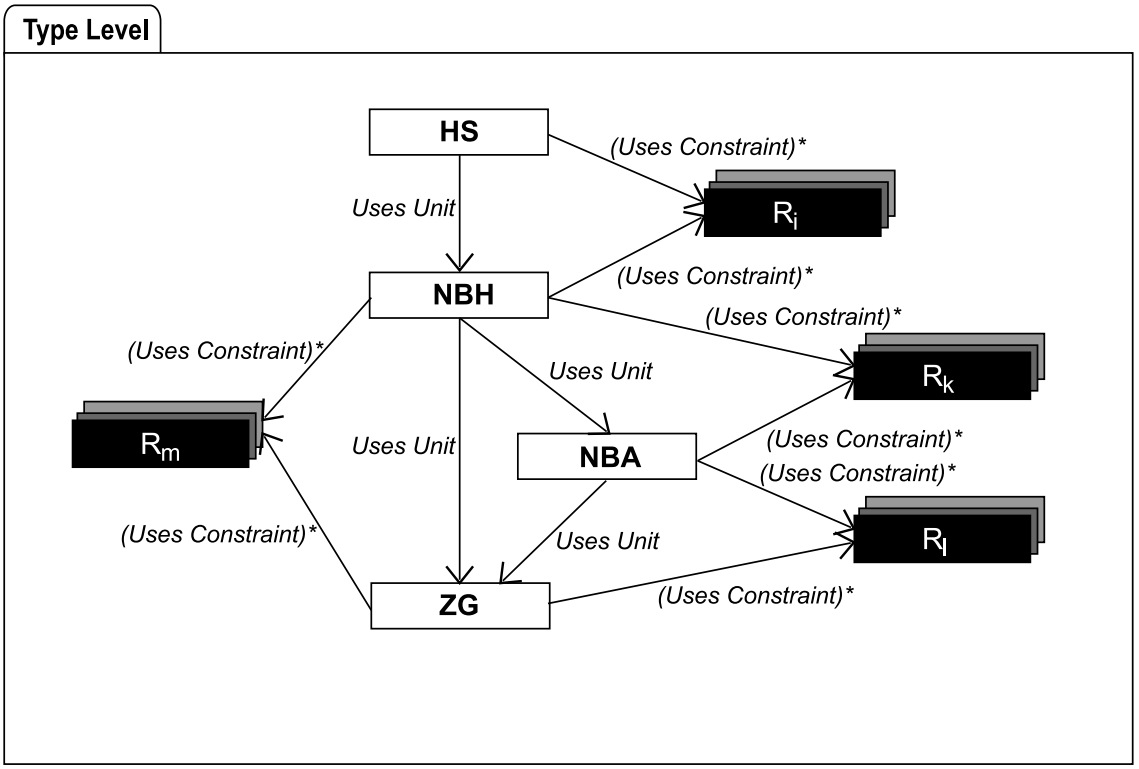

Abbildung 10.17: Modell für den Planungsprozess einer Stahlhalle

10.3.2 Festlegung der Nutzungsanforderungen

Im ersten Planungsschritt klärt der Architekt in einem Gespräch mit dem Bauherr die Nutzungsanforderungen an eine geplante Stahlhalle. Bauhöhe, Gesamtlänge und Gesamtbreite werden festgelegt und es erfolgt die Festlegung auf ein Rahmensystem. Weiterhin werden von dem Architekten die Achsen des Systems definiert. Der Architekt greift auf das Planungsmodell für die Planung derartiger Hallen zu und speichert die bisher festgelegten Eigenschaften als Element **hs, V1.0** im Configuration Level ab (Abbildung 10.18).

Der Architekt will diesen Planungszustand jederzeit wieder aufrufen können. Er führt die Operation *Snapshot()* aus und erzeugt ein unveränderliches Element **hs, V1.0** als Versionsvorgänger vom bearbeitbaren Element **hs, V1.1**.

Den Graph im Configuration Level zeigt Abbildung 10.19. Hierbei werden in dieser und den folgenden Abbildungen dieses Beispiels

- Uses Unit-Links mit *UU*,
- Uses Constraint-Links mit *UC* und
- Version Predecessor-Links mit *VP* abgekürzt.

Abbildung 10.18: Architekt speichert erstes Element

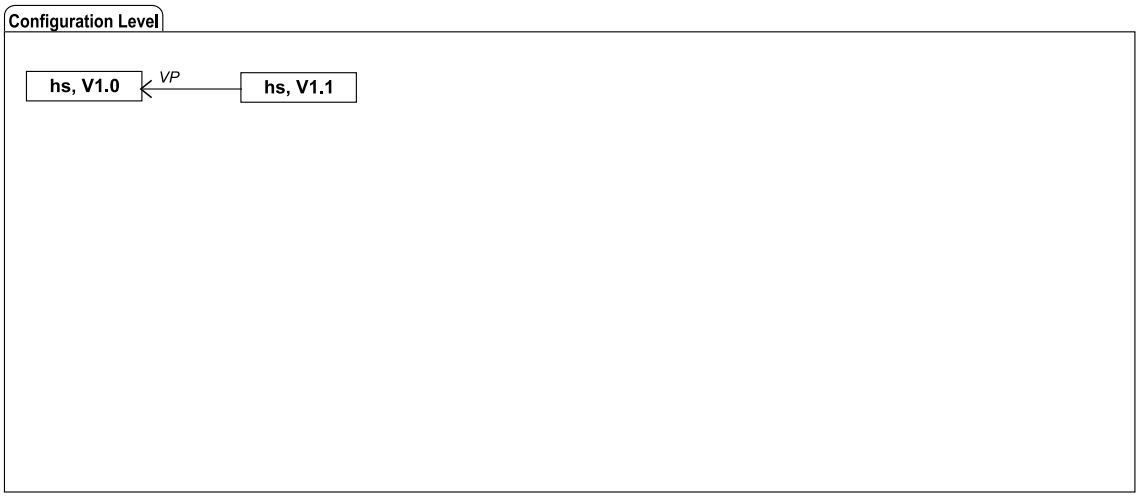

Abbildung 10.19: Archivierung des ersten Entwurfs

10.3.3 Nachweis und Bemessung der Hauptkonstruktion

Ein Statiker fertigt eine statische Berechnung an und wählt auf dieser Grundlage die Stahlprofile für Riegel und Stiele der Hallenrahmen aus. Basis für die statische Berechnung sind die im Element **hs, V1.1** gespeicherten Attribute. Er speichert seine Ergebnisse im Element **nbh, V1.0** ab. Da die erforderlichen Nachweise in einer typisierten Form vorliegen, kann anhand der automatisch eingefügten Gültigkeitsregeln festgestellt werden, dass der Zustand konsistent ist. Den Graph der Elemente im Configuration Level stellt Abbildung 10.20 dar.

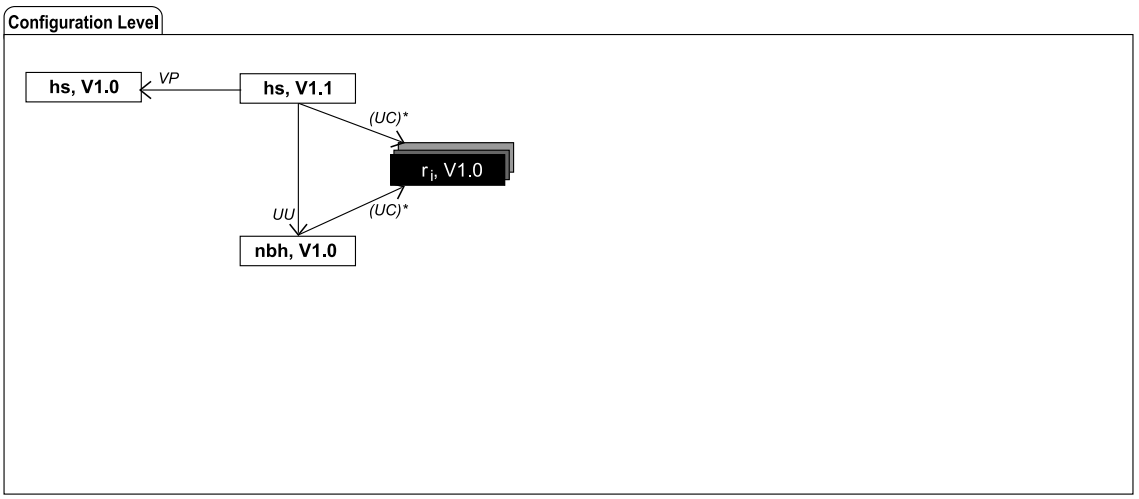

Abbildung 10.20: Fertiggestellte Bemessung der Hauptkonstruktion

10.3.4 Änderung der Nutzungsanforderungen

Nachdem bereits die Hauptkonstruktion nachgewiesen ist, ändert der Bauherr die Anforderungen. Hierfür benachrichtigt er den Architekten. Der Architekt will zum Nachweis der bisher geleisteten Arbeit den bisherigen konsistenten Zustand der Planung archivieren und führt daher die Operation *Snapshot()* auf das Element **hs, V1.1** aus. Als Ergebnis der Operation wird die bisherige Konfiguration archiviert und er erhält eine neue bearbeitbare Konfiguration mit dem Wurzelelement **hs, V1.2** (Abbildung 10.21.

Die notwendigen Änderungen trägt er in **hs, V1.2** ein und er erhält als Ergebnis einer Überprüfung der Gültigkeitsregeln, dass die Bemessung der Hauptkonstruktion ebenfalls geändert werden muss. Der vom Architekten informierte Statiker bearbeitet die Bemessung der Hauptkonstruktion **nbh, V1.1**. Ergebnis ist wieder eine konsistente Konfiguration mit Wurzelelement **hs, V1.2**.

Auch dieser Zustand wird archiviert. Das Resultat ist in Abbildung 10.22 dargestellt. Hier ist ersichtlich, dass alle archivierten Konfigurationen auf die Regeln $\mathbf{R}_i$,**V 1.0** verweisen können, da diese bisher nicht verändert wurden. Die neue, veränderbare Konfiguration besitzt das Wurzelelement **hs, V1.3**.

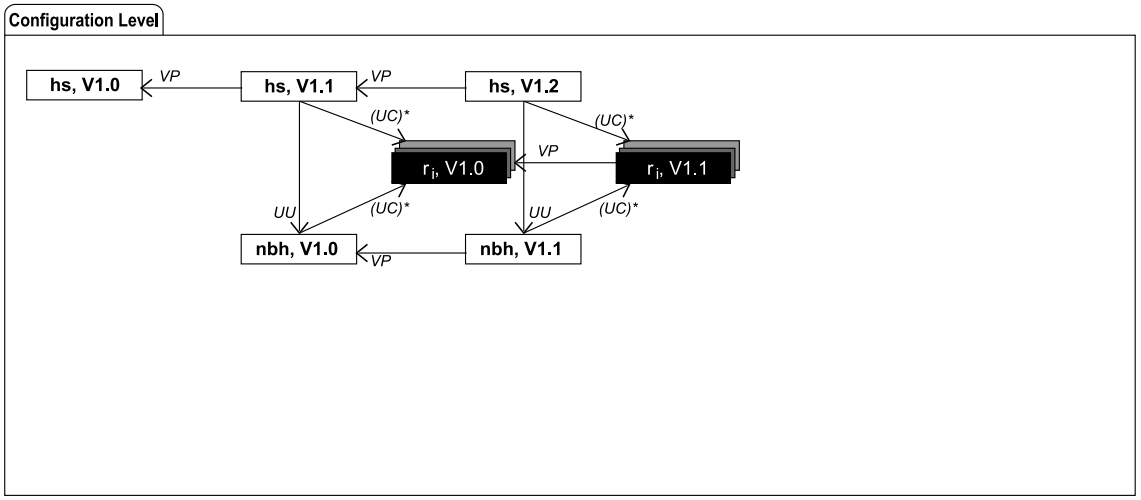

Abbildung 10.21: Archivierung des Planungszustands **hs, V1.1**

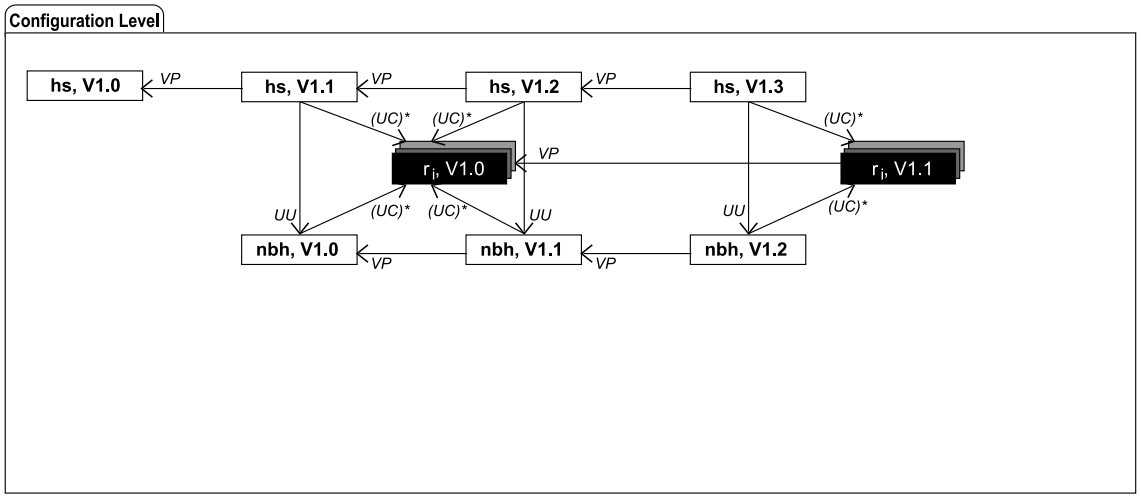

Abbildung 10.22: Archivierung des Planungszustands **hs, V1.2**

10.3.5 Nachweis und Bemessung der Anschlüsse

Ein weiterer Statiker führt die Bemessung und den Entwurf der Anschlüsse durch. Die Ergebnisse seiner Arbeit speichert er in **nba, V1.0** ab. Die relevanten Gültigkeitsregeln $\mathbf{R}_k$,**V 1.0** werden programmautomatisch auf der Grundlage der Definitionen im Type Level eingefügt. Das Ergebnis zeigt Abbildung 10.23.

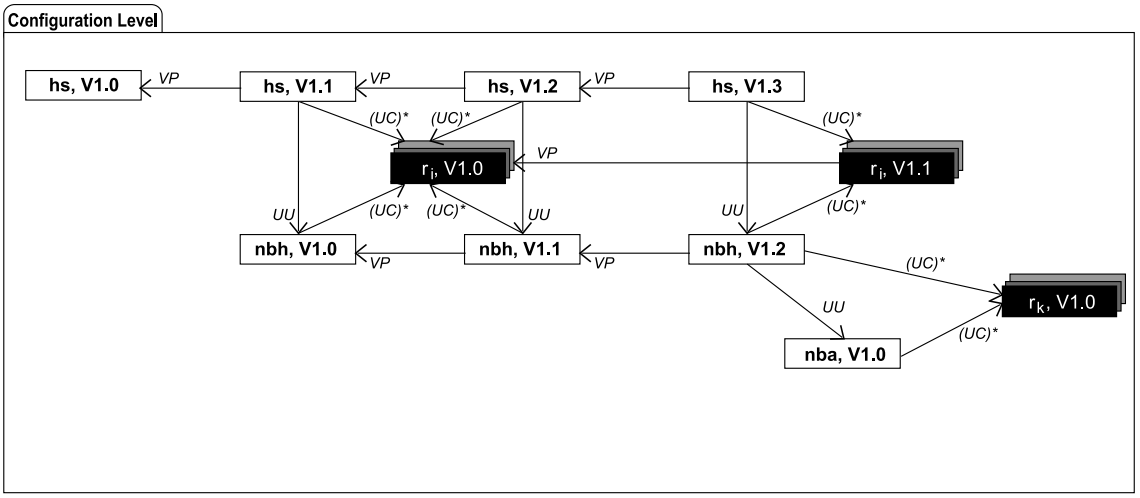

Abbildung 10.23: Nachweis und Bemessung der Anschlüsse

10.3.6 Zeichnung von Übersichten und Details

Abschliessend fertigt der Zeichner die Darstellungen von Übersichten und Details an. Dadurch erweitert sich der Graph des Configuration Level um das Element **zg, V1.0** und um die Regelmengen $\mathbf{R}_l$,**V1.0** und $\mathbf{R}_m$,**V1.0**. Der resultierende Graph des Configuration Level zeigt Abbildung 10.24.

10.3.7 Darstellung aller unveränderlichen Konfigurationen

Zur Archivierung des Endzustands der Planung wird die Operation *Snapshot()* auf das Element **hs, V1.3** ausgeführt. Abbildung 10.25 zeigt den Graph im Configuration Level, der von den unveränderlichen Elementen erzeugt wird. Nicht berücksichtigt ist die zusätzlich zur Verfügung stehende veränderbare Konfiguration mit Wurzelelement **hs, V1.4**.

10.3.8 Vereinfachtes Vorgehen

Der Aufbau einer Konfiguration im Configuration kann dadurch vereinfacht werden, dass Instanzen aller Elemente des Type Level in genau einem Bearbeitungsschritt in den Configuration Level zusammen mit den Verknüpfungen zueinander eingefügt

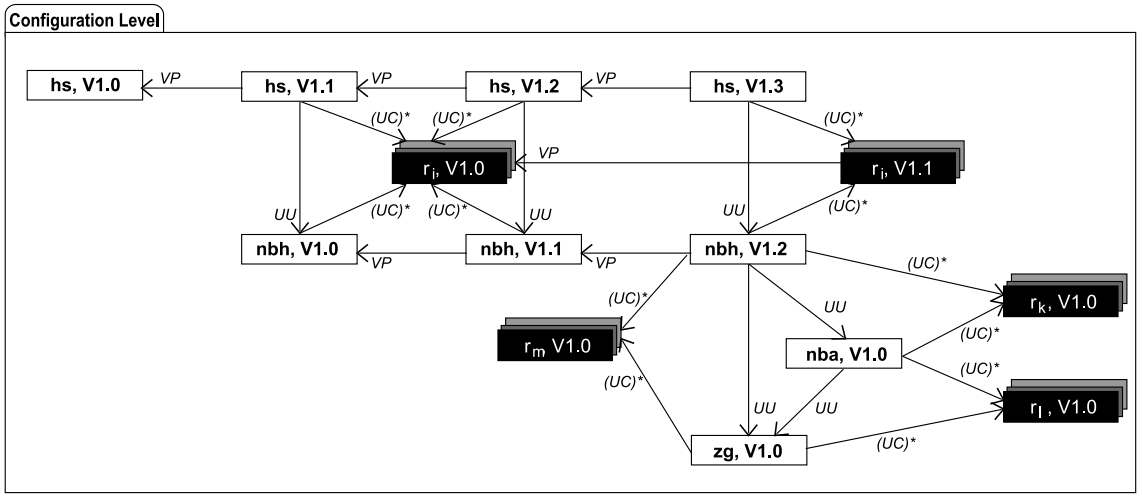

Abbildung 10.24: Zeichungen und Übersichten werden gespeichert

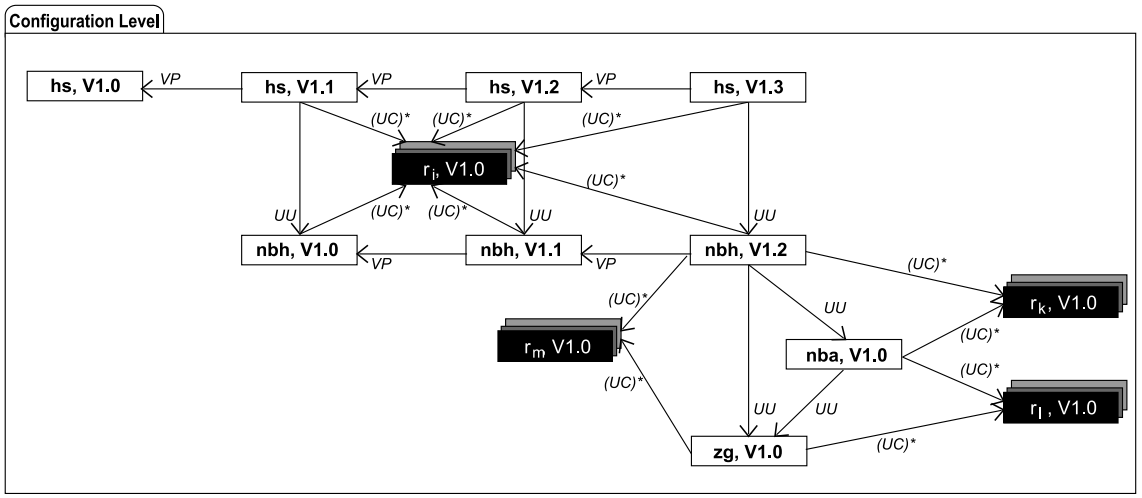

Abbildung 10.25: Darstellung aller erzeugten unveränderlichen Konfigurationen

werden. Damit wird der Graph im Type Level als Kopie in den Configuration Level eingefügt.

Weiterhin ergibt sich eine einfache Möglichkeit der programmautomatischen Kontrolle der Konsistenz einer Planung, wenn für jedes Element der Zeitpunkt der letzen Bearbeitung als Attributwert gespeichert wird.

Abbildung 10.26 zeigt einen konsistenten Zustand. Abhängige Elemente haben ein jüngeres Speicherdatum als die Elemente, von denen sie abhängen. Alle vorhandenen Gültigkeitsregel werden erfüllt.

Wird **hs, V1.0** nachträglich verändert, so ist die Gültigkeitsregel auf die **hs, V1.0** verweist, nicht mehr erfüllt. Der Zustand ist inkonsistent. Weiterhin ist ersichtlich, dass in einem nächsten Schritt **nbh, V1.0** aktualisiert werden muss. Durch die Auswertung der Gültigkeitsregeln können die jeweils zu aktualisierenden Elemente ausfindig gemacht werden. Schrittweise kann ein konsistenter Zustand hergestellt werden.

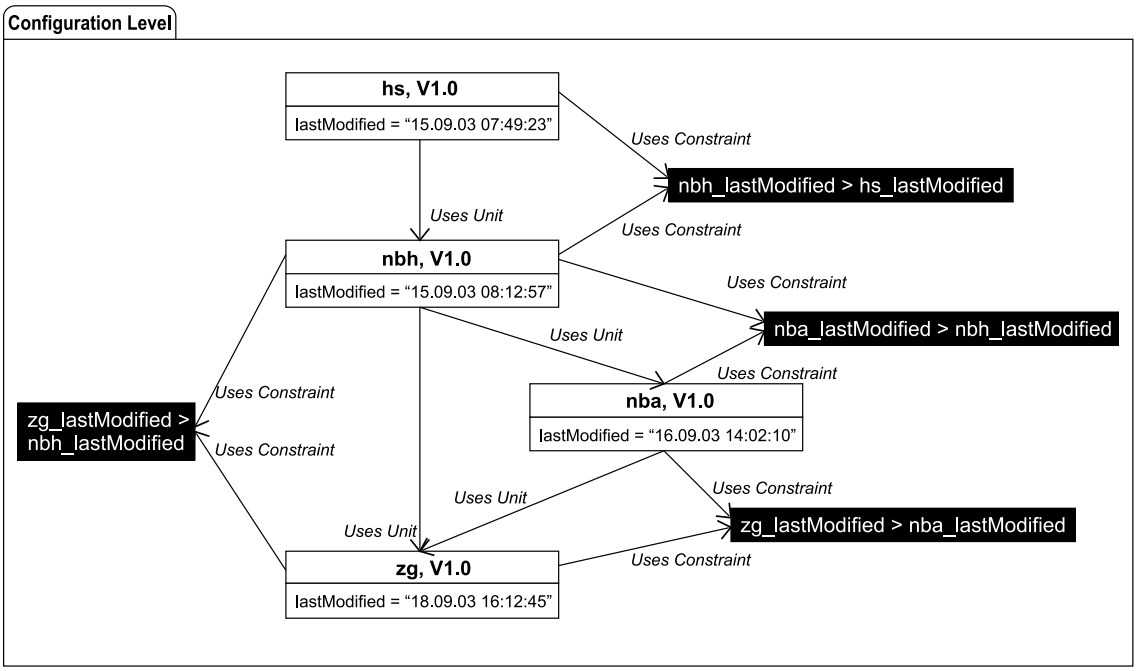

Abbildung 10.26: Kontrolle der Konsistenz über den Zeitpunkt der letzten Bearbeitung

Kapitel 11

Anwendung des Planungssystems am Beispiel Gerüstbau

11.1 Allgemeines

In diesem Kapitel wird die Anwendung des Planungssystems zur Unterstützung bei der Angebotserstellung in einem Produktionsbetrieb zur Herstellung von Systemgerüsten vorgestellt.

Dabei wird das Planungssystem nicht auf die Unterstützung des Herstellungsprozesses von Gerüstbauteilen angewendet. Vielmehr wird das System zur projektbezogenen Planung von Fassadengerüsten angewendet.

Kennzeichnend für ein Fassadengerüst sind die folgenden Merkmale.

- Ein Fassadengerüst ist ein Unikatprodukt. Die einzelnen geplanten Fassadengerüste unterscheiden sich untereinander durch eine Vielzahl von Merkmalen.
- Die Komponenten eines Fassadengerüsts sind in Serie hergestellte Bauteile.
- Fassadengerüste müssen definierten Regeln entsprechen.

Wie im Verlauf des Beispiels deutlich wird, existieren viele Regelungen, die sicherstellen sollen, dass bestimmte Gerüstbauteile in der erforderlichen Anzahl in ein Fassadengerüst eingebaut werden. Daher zeigt dieses Beispiel die Anwendung von Regeln, die sich auf eine Menge von Elementen beziehen.

11.2 Umsetzung des Planungssystems

Der im Rahmen der vorliegenden Arbeit erstellte Prototyp wurde mit der objektorientierten Programmiersprache *Java* entwickelt. Eine Darstellung der Programmiersprache und wichtiger Bibliotheken ist in [33] und [34] enthalten.

Die Vorteile von Java sind die

- Unterstützung der Objektorientierten Programmierung,
- freie Verfügbarkeit der Laufzeitumgebung für mehrere Rechnerplattformen,
- Verfügbarkeit leistungsfähiger Komponenten, wie beispielsweise Sammlungen (Tabellen, Verkettete Listen, Bäume) und die
- freie Verfügbarkeit integrierter Entwicklungsumgebungen zur Java-Programmentwicklung.

Für die programmautomatische Überprüfung der Constraints wurde der *Java Math Expression Parser* [23] mit Erweiterungen durch die Verwendung von Regulären Ausdrücken verwendet. *Reguläre Ausdrücke* und ihre Anwendung sind in [52] beschrieben.

Die Darstellung der Graphen erfolgte mit der Swing-Komponente *JGraph* [2].

11.3 Situation

11.3.1 Normen und Regelungen

In Deutschland ist die Herstellung und der Aufbau von Gerüsten durch Normen und durch Allgemeine bauaufsichtliche Zulassungen geregelt. Hierbei regelt DIN 4420, Teil 1 [16] grundlegende Punkte wie

- Gerüstbauarten,
- Anforderungen an Bauteile und ihre Herstellung,
- Lastannahmen, Gerüstgruppen
- erforderliche Mindestabmessungen von Bauteilen und
- erforderliche Nachweise.

Kennzeichnend ist weiterhin, dass die spezifischen Regelungen für ein Gerüstsystem eines Herstellers in einer Allgemeinen bauaufsichtlichen Zulassung zusammengefasst werden, beispielsweise [5].

Hierin enthalten sind

- Bestimmungen für die Gerüstbauteile,
- Bestimmungen für Entwurf und Bemessung und
- Bestimmungen für die Ausführung.

Darüber hinaus enthalten diese Allgemeinen bauaufsichtlichen Zulassungen zahlreiche Aufbauvarianten, die *Regelausführungen* genannt werden. Werden die Gerüste entsprechend einer Regelausführung aufgebaut, so ist gemäss [16] der Nachweis der Standsicherheit bereits erbracht.

11.3.2 Produktentwicklung

Bestehende Gerüstsysteme werden laufend durch Neuentwicklungen erweitert. Dabei handelt es sich beispielsweise um zusätzliche Serienteile, die das vorhandene Gerüstsystem ergänzen. Im Rahmen dieser Neuentwicklungen werden von den Mitarbeitern der Entwicklungsabteilungen der Gerüstsystemhersteller auch einzuhaltende Aufbauregeln festgelegt. Weiterhin werden bereits existierende Aufbauregeln auch an sich verändernde gesetzliche Regelungen angepasst. Existierende Bauteile unterliegen ebenfalls einer stetigen Weiterentwicklung.

Um die Anzahl von Anfragen an die Mitarbeiter der Entwicklungsabteilung bezüglich der Aubauregeln gering zu halten, ist eine Dokumentation der Regeln erforderlich.

Die Situation in der Produktentwicklung stellt sich damit wie folgt dar.

- Ständige Erweiterung der Gerüstsysteme um neue Teile.
- Anpassung der Aufbauregeln aufgrund veränderter gesetzlicher Regelungen oder neuer Teile.
- Notwendige schnelle Weitervermittlung neuer Möglichkeiten an Mitarbeiter von Vertrieb und Technischer Angebotsbearbeitung.

11.3.3 Vertrieb und Technische Angebotsbearbeitung

Die Mitarbeiter der Abteilungen Vertrieb und Technische Angebotsbearbeitung erstellen kundenspezifische Angebote. Wenn diese Angebote einen oder mehrere Aufbauvorschläge enthalten, so sind diese entsprechend der geltenden Aufbauregeln auszuführen. In der überwiegenden Anzahl der anzufertigenden Angebote sind mehrere Varianten zu planen.

Die Angebote müssen aus zwei Gründen rückverfolgbar verwaltet werden.

1. Zur Verringerung des Zeitaufwands bei der Angebotsbearbeitung wird versucht, auf bestehende Planungen aufzubauen und diese zu modifizeren.
2. Bis zu einer Auftragsvergabe und auch danach müssen Nachfragen zu den gemachten Planungen beantwortet werden können.

Neue Aufbaumöglichkeiten müssen von den Mitarbeitern der Vertriebsabteilungen und den Mitarbeitern der Technischen Angebotsbearbeitung möglichst schnell umgesetzt werden können. Hierfür brauchen diese

- Kenntnis über die gültigen Aufbauregeln und
- Entscheidungshilfen, ob eine Ausarbeitung den Aufbauregeln entspricht.

Zusammenfassend ergibt sich folgende Situation.

- Planungen für Angebote sollen den jeweils geltenden Aufbauregeln entsprechen und erweiterte Möglichkeiten durch neu entwickelte Komponenten berücksichtigen.
- Die bearbeiteten Angebote müssen rückverfolgbar verwaltet werden können.

11.4 Existierende Lösungsansätze zur Gerüstplanung

Zur Unterstützung der Planung einfacher Fassadengerüste wurde vom Autor das Programm *Der Layher Gerüstplaner* entwickelt (Abbildung 11.1) [58].

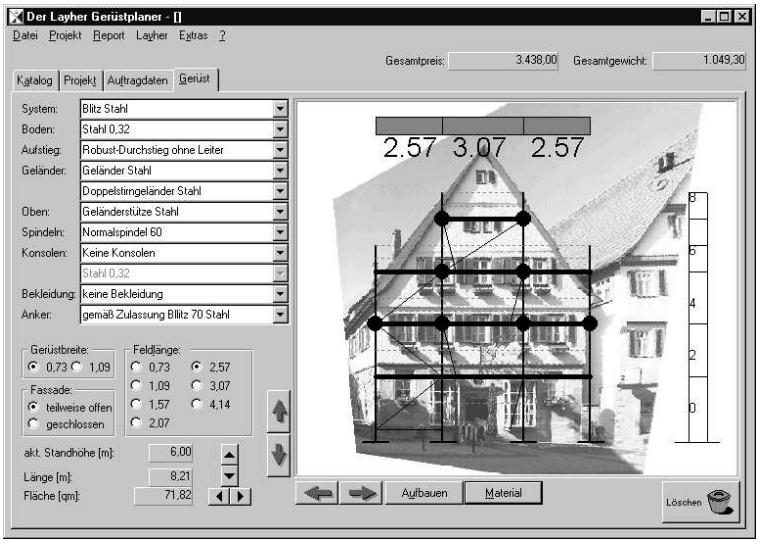

Abbildung 11.1: Der Layher Gerüstplaner

Diese Softwarelösung zeichnet sich durch folgende Eigenschaften aus.

- Erstellung von Fassadengerüsten nach den deutschen Aufbauregeln.
- Ermittlung der Materialliste.
- Export der Materialliste zur Weiterverarbeitung im Programm zur Angebotserstellung.

Die Aufbauregeln, auf deren Basis das Programm einen Aufbauvorschlag generiert sind ausschliesslich im Quellcode programmiert. Daraus ergeben sich die wesentlichen Nachteile dieser Lösung.

- Der Programmierer muss zusätzlich zu seinem umfangreichen Fachwissen alle gerüstbauspezifischen Regeln kennen.

- Bei Veränderungen an den Regeln oder der Produktzusammensetzung muss der Quellcode des Programms durch einen Programmierer angepasst und neu übersetzt werden.
- Die Aufbauregeln für Fassadengerüste weichen länderspezifisch voneinander ab. Die länderspezifischen Regelungen können derzeit nur im Programmcode berücksichtigt werden.

11.5 Anforderungen an eine neue Lösung

11.5.1 Anforderungen im Bereich Produktentwicklung

Die neue Softwarelösung soll es den Entwicklungsingenieuren des Fachbereichs ermöglichen, das Produktmodell und die zugehörigen Regeln in einem Computerprogramm zu definieren. Hierzu sollen keine Programmierkenntnisse erforderlich sein. Das Ändern von Regeln oder der Produktstruktur soll ohne anschliessendes Übersetzen des Programms durchführbar sein.

Durch Neuentwicklungen verändern sich Produktzusammensetzung und zugehörige Regeln im zeitlichen Verlauf. Weiterhin existieren mehrere Gerüstsysteme gleichzeitig. Daraus folgt, dass das neue System sowohl mehrere unterschiedliche Konfigurationen in zeitlicher Abfolge als auch mehrere Konfigurationen zeitgleich nebeneinander verwalten muss.

11.5.2 Vertrieb und Technische Angebotsbearbeitung

Die Mitarbeiter von Vertriebsabteilungen und der Technischen Angebotsbearbeitung planen ihre Gerüste auf der Basis der Definitionen, die zuvor von den Entwicklungsingenieuren festgelegt wurden. Die Einhaltung der Regeln soll programmautomatisch geprüft werden können.

Auch in diesem Bereich ist es notwendig, dass zahlreiche Konfigurationen zeitgleich und in zeitlicher Abfolge verwaltet werden müssen.

11.6 Komponenten eines einfachen Fassadengerüsts

Die Abbildung 11.2 zeigt die notwendigen Komponenten eines einfachen Fassadengerüsts.

Spindel: Die Spindeln sind das Anfangsbauteil für ein Fassadengerüst. Mit ihnen kann ein Ausgleich bei Geländeunebenheiten erfolgen, da diese höhenverstellbar sind.

Stellrahmen: Bauteil zum Vertikalausbau des Fassadengerüsts.

Boden/Durchstieg: Bauteil zum horizontalen Ausbau des Fassadengerüsts. Weiterhin dienen die Böden als Arbeitsfläche und die Durchstiege darüber hinaus zum Erreichen der durch die Böden entstehenden Ebenen.

Diagonale: Element zur Aussteifung des Fassadengerüsts.

Geländer: Schutz der auf einem Fassadengerüst befindlichen Personen gegen Absturz.

Zur Gruppierung der Elemente eines Fassadengerüsts sind weiterhin die folgenden Definitionen hilfreich.

Ebene: Wie in Abbildung 11.2 dargestellt, bilden alle Böden und Durchstiege einer Höhe eine Ebene. Dieser Ebene können auch die direkt an diesen Böden und Durchstiegen beginnenden Stellrahmen, Geländer und Diagonalen zugeordnet werden. Damit wird ein Fassadengerüst in mehrere Ebenen unterteilt.

Feld: Ein Feld dient der weiteren Unterteilung der Ebenen. Er bezeichnet den Bereich einer Ebene zwischen zwei Stellrahmen. Diesem Bereich können Böden, Durchstiege, Geländer und Diagonale zugeordnet werden.

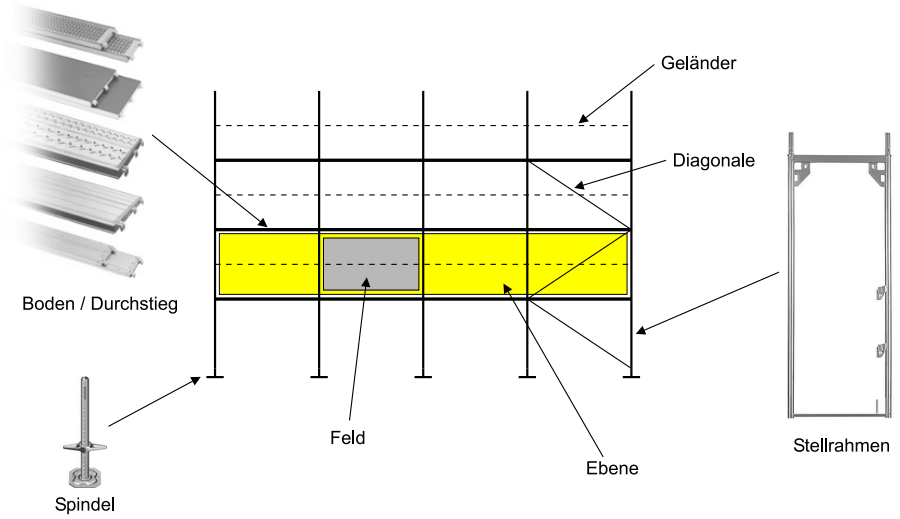

Abbildung 11.2: Komponenten eines vereinfachten Fassadengerüsts

11.7 Type Level

Das Modell für ein Fassadengerüst ergibt sich direkt aus den Definitionen des Abschnitts 11.6. Die Abbildung 11.3 zeigt die Produktstruktur für ein Fassadengerüst. Diese Darstellung enthält zunächst noch keine Regeln.

Die Beziehungen zwischen den einzelnen Elementen sind überwiegend "Besteht-aus-Beziehungen". Im Rahmen der Abbildung der Spindeln wurde bewusst davon abgewichen. Die Bindung des Elements **Spindel** an das Element **Stellrahmen** orientiert sich daran, dass Spindeln ausschliesslich zusammen mit Stellrahmen in einem Fassadengerüst vorkommen können.

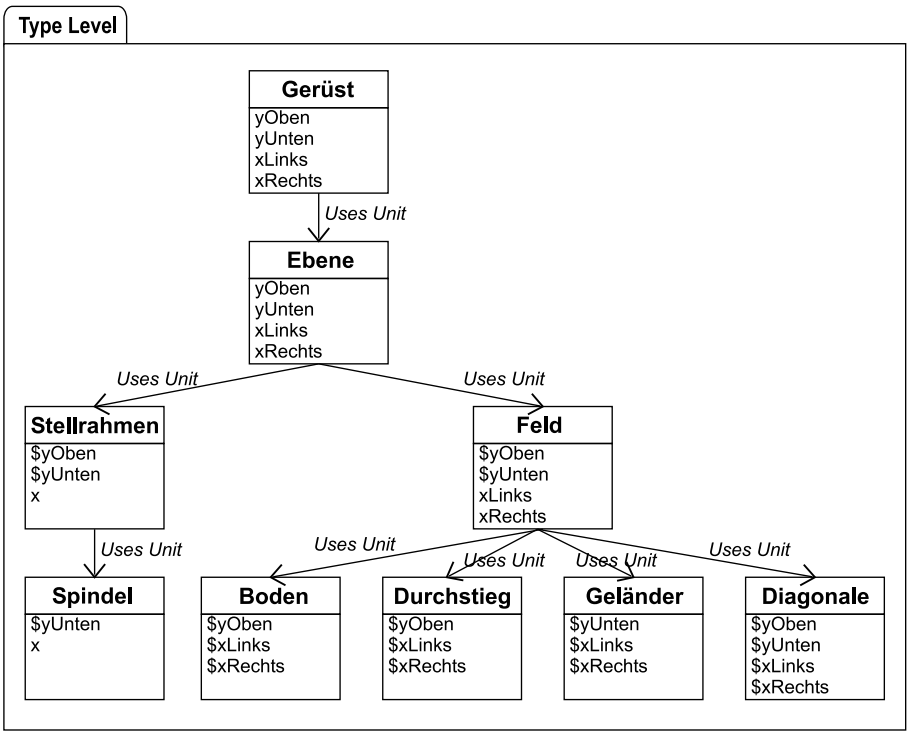

Abbildung 11.3: Modell des Fassadengerüsts ohne Regeln im Type Level

Die aufgeführten Attribute sind die Koordinaten in einem globalen x-y-Koordinatensystem.

11.7.1 Umgesetzte Regeln

Die folgenden Abschnitte zeigen die Umsetzung von Regeln aus dem Gerüstbau. Zur Wahrung der Übersichtlichkeit wird eine Darstellung des Modells ohne die zu den Elementen gehörigen Attribute gewählt. Weiterhin wird jeweils nur diejenigen Gültigkeitsregeln dargestellt, die zur Umsetzung der Gerüstbauregel erforderlich sind.

Regel 1: Anzahl der Spindeln

Die Regel soll sicherstellen, dass jeder Stellrahmen der untersten Ebene auf jeweils zwei Spindeln steht. Die Umsetzung dieser Regel zeigt die Abbildung 11.4.

Im Configuration Level nehmen alle Attribute der Elemente konkrete Werte an. Entsprechend nehmen alle Attribute *yUnten* der Elemente des Typs *Ebene* im Verlauf des Konfigurationsprozesses einen Wert an. Die Regel zeigt, wie die Anzahl an

Instanzen eines Elements des Type Level im Configuration Level geprüft werden kann.

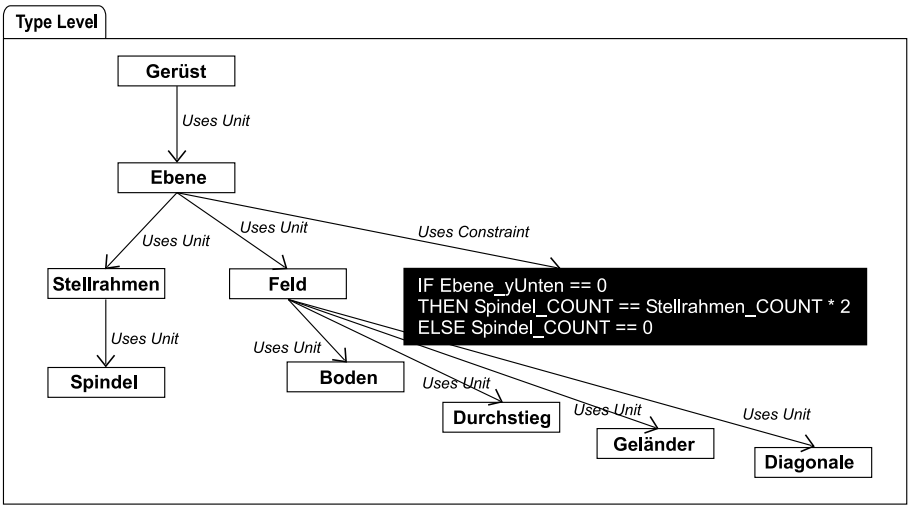

Abbildung 11.4: Anzahl der Spindeln

Regel 2: Anzahl der Diagonalen

Zur Sicherstellung einer ausreichenden Aussteifung muss jedes fünfte Gerüstfeld mit einer Diagonale ausgesteift werden. In Abbildung 11.5 ist dargestellt, wie diese Regel umgesetzt werden kann. Die Regel ermittelt anhand der Anzahl der Felder einer Ebene die Gesamtanzahl notwendiger Diagonalen in der Ebene. Die Verteilung der Diagonalen in der Ebene wird nicht weiter festgelegt.

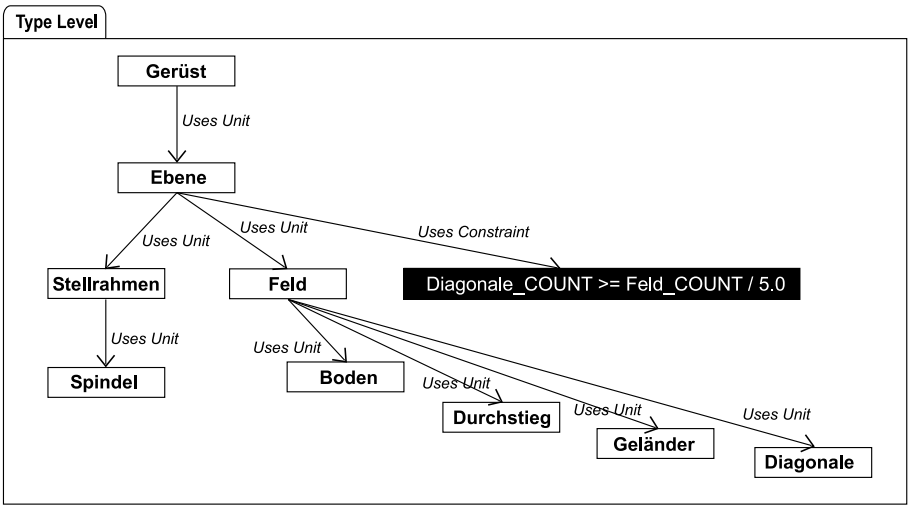

Abbildung 11.5: Anzahl der Diagonalen

Regel 3: Anzahl der Aufstiegsfelder

Um jeden Bereich eines grossen Fassadengerüsts ausreichend schnell erreichen zu können, sind eine ausreichende Anzahl von Durchstiegen einzubauen. Als Regel ist definiert, dass alle 50m ein Durchstieg vorzusehen ist. Die Umsetzung in den Type Level zeigt Abbildung 11.6. In der obersten Ebene ist kein Aufstieg erforderlich. Dies kann in der Formulierung der Gültigkeitsregel beispielsweise dadurch berücksichtigt werden, dass die Anzahl der Durchstiege nur überprüft wird, wenn auch Böden vorhanden sind. Die Felder der obersten Ebene beinhalten keine Böden.

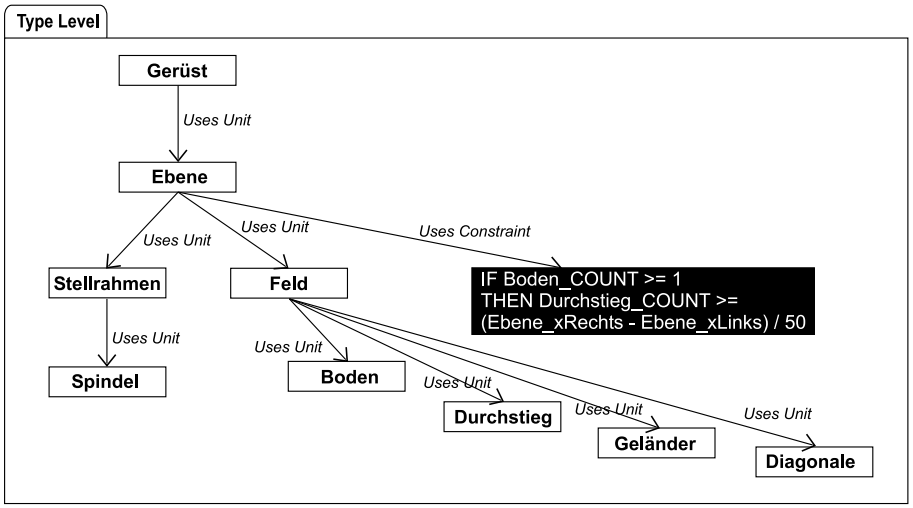

Abbildung 11.6: Anzahl der Aufstiegsfelder

Regel 4: Schutz vor Absturz

Bei Absturzhöhen grösser oder gleich 2,0m ist gemäss der geltenden Regelungen ein Geländer einzubauen. Da die Elemente als Attribute ihre Lage in einem globalen Koordinatensystem enthalten, kann die Regel wie in Abbildung 11.7 durch Verwendung zweier Regeln im Type Level umgesetzt werden. Eine Regel ist mit dem Element **Ebene** verknüpft und sagt aus, dass die Anzahl der Geländer einer Ebene gleich der Anzahl der Felder sein muss, wenn die Ebene oberhalb von $2.0m$ endet. Zur Ergänzung ist mit dem **Feld** eine Regel verknüpft, die sicherstellt, dass ein Feld höchstens ein Geländer besitzen darf.

Regeln zur geometrischen Verträglichkeit

Abbildung 11.8 zeigt an weiteren Beispielen die Umsetzung von Regeln zur geometrischen Verträglichkeit. Die Regeln garantieren, dass das geplante Fassadengerüst physikalisch sinnvoll ist. So garantieren beispielsweise die Regeln, die jeweils mit den Elementen **Feld** und **Boden** verknüpft sind, dass die x-Koordinaten des Felds mit den x-Koordinaten des Bodens übereinstimmen. Die Regel mit der Verknüpfung

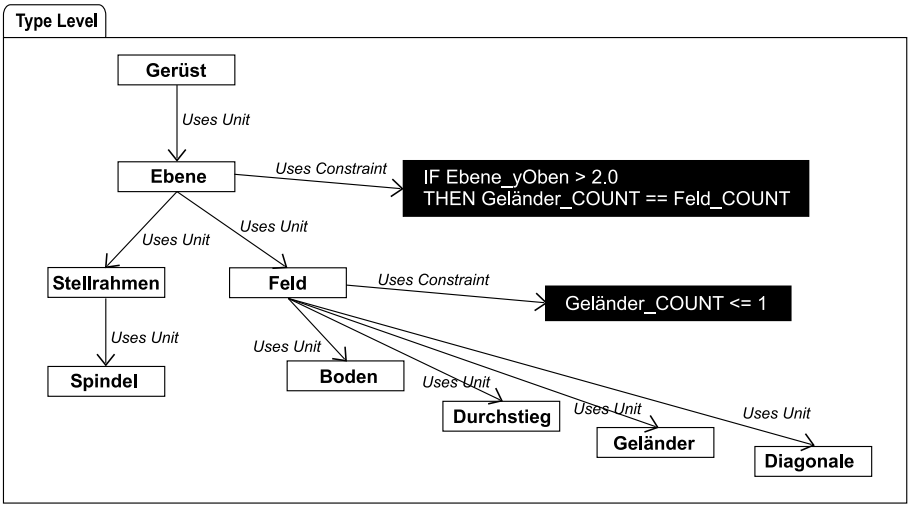

Abbildung 11.7: Anzahl der Geländer

zwischen **Feld** und **Diagonale** überprüft, ob eine Diagonale mit korrekter Länge in das Fassadengerüst eingebaut wurde.

11.7.2 Regeln zur Kontrolle der Anzahl von Elementen

Ein Modell für Fassadengerüste im Type Level soll Grundlage für beliebig grosse benutzerspezifische Fassadengerüste im Configuration Level sein. Dies ist nicht möglich, wenn das Modell des Type Level die Anzahl an Elementen vorab festlegt.

Das in dieser Arbeit beschriebene Planungssystem ermöglicht daher die Erzeugung beliebig vieler Instanzen eines Elements, was Abschnitt 8.4 dokumentiert. Die Kontrolle der Anzahl wird durch Verwendung entsprechender Regeln ermöglicht. Diese erläutert Abschnitt 7.6. Dabei handelt es sich um Regeln für Elemente gemäss Abschnitt 8.7.

Die Umsetzung der Gerüstbauregeln 1 bis 4 in Abschnitt 11.7.1 zeigt die Anwendung dieser Gültigkeitsregeln. Die Arbeitsweise dieser Regeln kann am Beispiel der Umsetzung der Gerüstbauregel 4 in Abbildung 11.7 gezeigt werden.

Die erste Regel lautet

`IF Ebene_yOben > 2.0 THEN Geländer_COUNT == Feld_COUNT`.

Diese Regel ist mit Elementen des Typs **Ebene** verknüpft. Zur Auswertung der Regel in einer benutzerspezifischen Konfiguration im Configuration Level werden die Elemente der Typen **Geländer** und **Feld** gezählt, die über Verknüpfungen von einem Element **Ebene** aus erreichbar sind.

Die zweite Regel,

`Geländer_COUNT <= 1`,

ist im Configuration Level jeweils mit einem Element des Typs **Feld** verknüpft. Zur

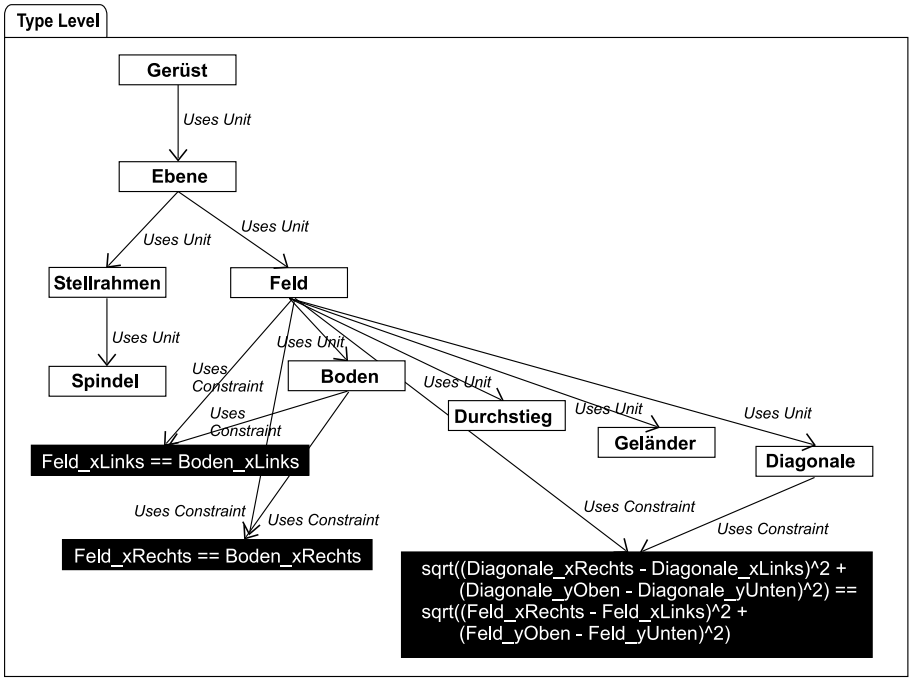

Abbildung 11.8: Regeln zur geometrischen Verträglichkeit

Auswertung der Regel wird die Anzahl der Elemente **Geländer** ermittelt, die über Verknüpfungen von **Feld** aus erreichbar sind.

11.8 Programmablauf

11.8.1 Admin-UI

Abbildung 11.9 zeigt die Benutzerschnittstelle zur Eingabe eines Produktmodells und der zugehörigen Gültigkeitsregeln. Der abgebildete Graph zeigt einen Ausschnitt aus dem Modell für Fassadengerüste. Zusätzlich werden alle Komponenten des Fassadengerüsts in einer Tabelle angezeigt.

11.8.2 Client-UI

Erstellen einer benutzer- beziehungsweise auftragsbezogenen Konfiguration

In Abbildung 11.10 ist eine einfache fachspezifische Benutzeroberfläche dargestellt. Diese Benuzeroberfläche beinhaltet eine Menge an Befehlsschaltflächen zum Einfügen entsprechender Gerüstkomponenten. Weiterhin kann über eine Befehlsschaltfläche eine Überprüfung aller enthaltenen Gültigkeitsregeln gestartet werden. Das Ergebnis der Überprüfung wird als Text im unteren Fensterabschnitt angezeigt. Der

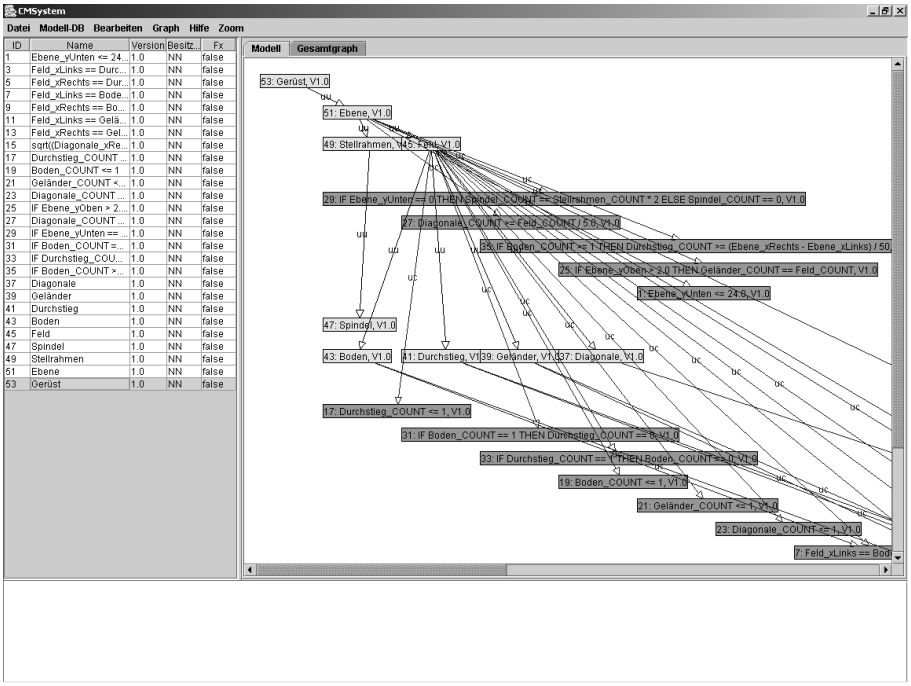

Abbildung 11.9: Darstellung des Admin-UI mit einem Ausschnitt des Produktmodells

gesamte verbleibende Bereich der Benutzeroberfläche dient der Darstellung des in der Planung befindlichen Fassadengerüsts.

Das Erstellen einer spezifischen Konfiguration erfolgt ein mehrerern aufeinanderfolgenden Schritten. Hierbei werden jeweils Instanzen von Elementen des Type Level der Datenhaltung des Configuration Level als Elemente hinzugefügt.

- Im ersten Schritt wird eine Instanz von **Gerüst** hinzugefügt. Abbildung 11.11 zeigt, mit welchen Attributwerten die Instanz zum Configuration Level addiert wird.
- Im Anschluss wird eine Instanz von **Ebene** hinzugefügt (Abbildung 11.12). Abbildung 11.13 zeigt den Graph im Configuration Level nach dem Einfügen der Ebene. Im Type Level sind Regeln mit dem Element **Ebene** verknüpft. Instanzen dieser Regeln können ebenfalls in den Configuration Level eingefügt werden.
- Danach wird eine Instanz von **Feld** addiert (Abbildung 11.14). Den Graphen des Configuration Level nach diesen Operationen zeigt Abbildung 11.15. Wie in Abbildung 11.7 und 11.8 dargestellt, sind Regeln mit dem Element **Feld** im Type Level verknüpft. Die mit **Feld** verknüpfte Regel aus Abbildung 11.7 kann bereits programmautomatisch eingefügt werden. Die mit **Feld** gemäss Abbildung 11.8 verknüpften Regeln können noch nicht eingefügt werden. Es existiert im Configuration Level noch keine Instanz des Elements Boden, die

Abbildung 11.10: Darstellung der fachspezifischen Benutzerschnittstelle

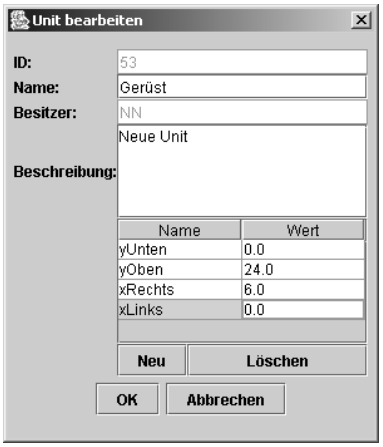

Abbildung 11.11: Hinzufügen einer Instanz von **Gerüst**

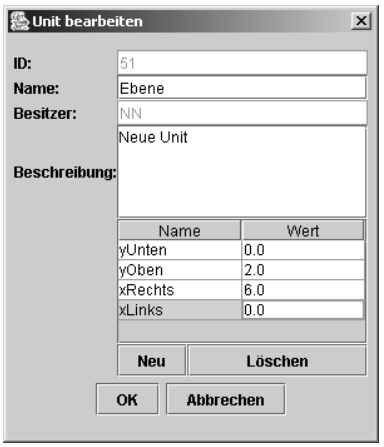

Abbildung 11.12: Hinzufügen einer Instanz von **Ebene**

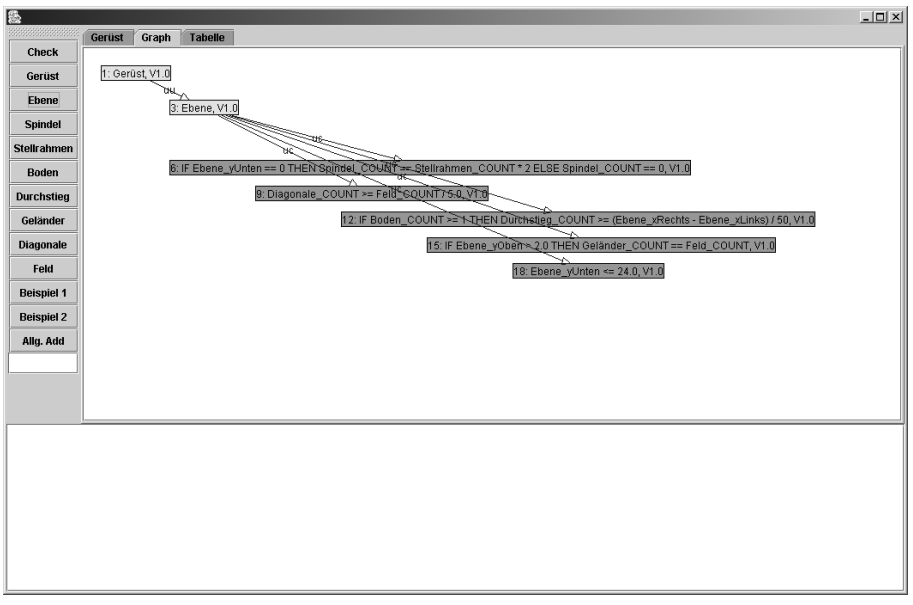

Abbildung 11.13: Graph des Configuration Level nach dem Einfügen einer Instanz von **Ebene**

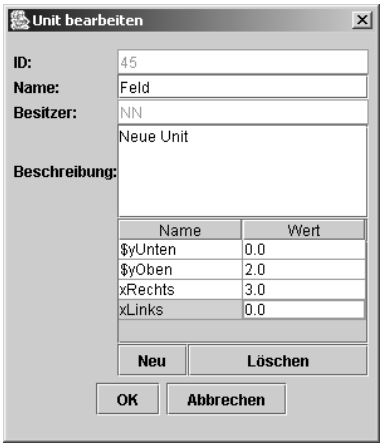

Abbildung 11.14: Hinzufügen einer Instanz von **Feld**

von Feld aus über einen Uses Unit-Link verfügbar ist. Diese ist erforderlich, da die Regeln auch mit dieser Instanz von **Boden** verknüpft werden muss.

- In einem weiteren Schritt wird eine Instanz der Komponente **Boden** eingefügt (Abbildung 11.16). Instanzen der Regeln aus Abbildung 11.8 können jetzt eingefügt werden, da Instanzen aller diese Regeln referenzierenden Elemente im Configuration Level existieren. Anschliessend ergibt sich der in Abbildung 11.17 dargestellte Graph des Configuration Level.

- Analog hierzu werden zwei Elemente **Stellrahmen** und insgesamt vier **Spindel**-Elemente eingefügt. Den Graph des Configuration Level nach dem Einfügen zeigt 11.18. Das bisher entstandene Gerüst nach dem Einfügen der Bauteile zeigt Abbildung 11.19.

Eine erste Prüfung der Gültigkeitregeln dokumentiert Abbildung 11.20. Erwartungsgemäss ist die Gerüstbauregel 2 für die Anzahl der Diagonalen nicht erfüllt. Ebenso ist die Gerüstbauregel 3 für die Anzahl der Aufstiegsfelder nicht erfüllt. Da die Stellrahmen und Spindeln Regelkonform hinzugefügt wurden, sind alle hierfür relevanten Gültigkeitsregeln erfüllt.

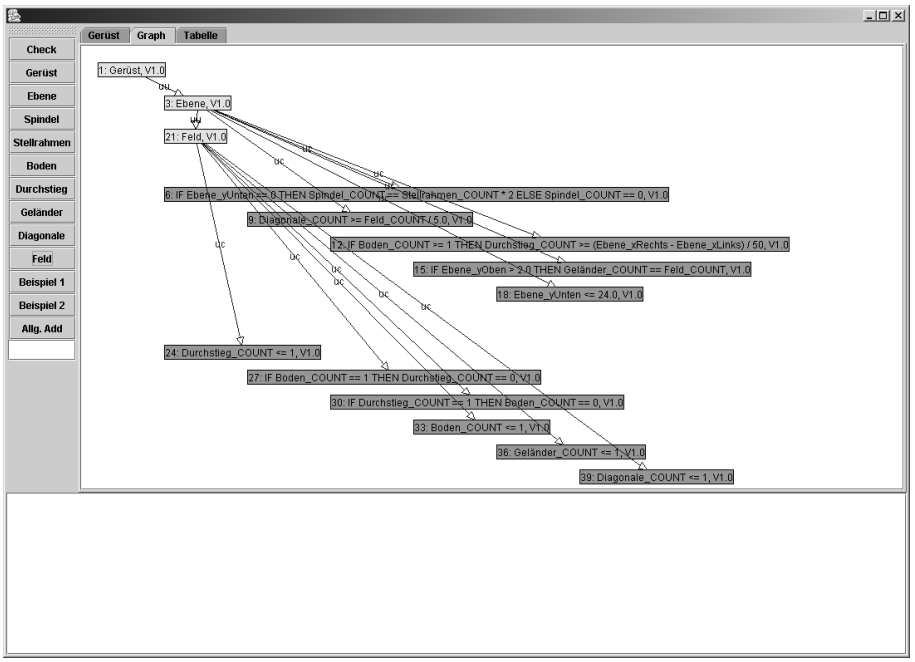

Abbildung 11.15: Graph des Configuration Level nach dem Einfügen einer Instanz von **Feld**

Abbildung 11.16: Hinzufügen einer Instanz von **Boden**

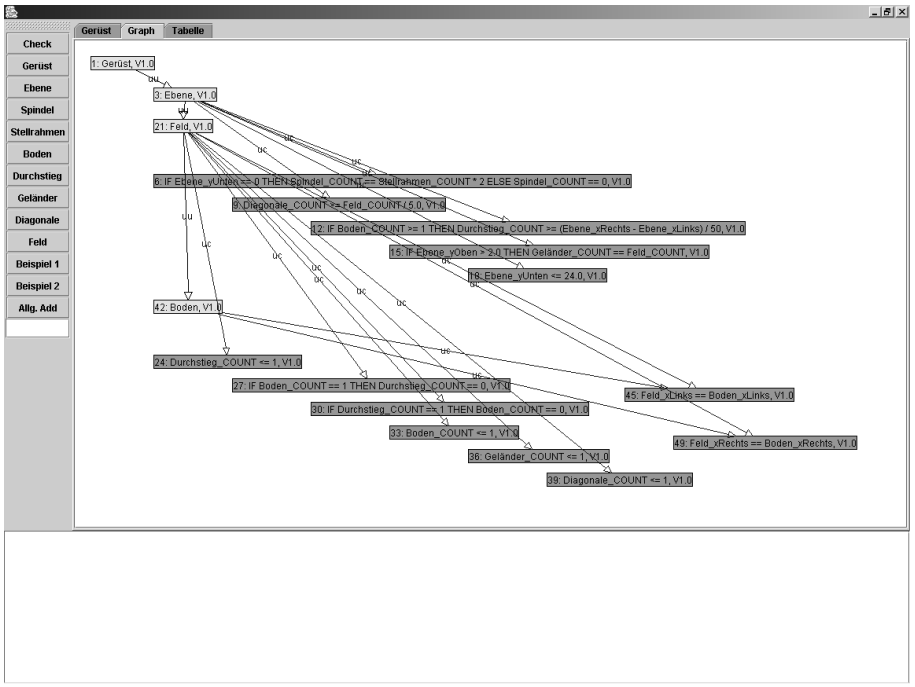

Abbildung 11.17: Graph des Configuration Level nach dem Einfügen einer Instanz von **Boden**

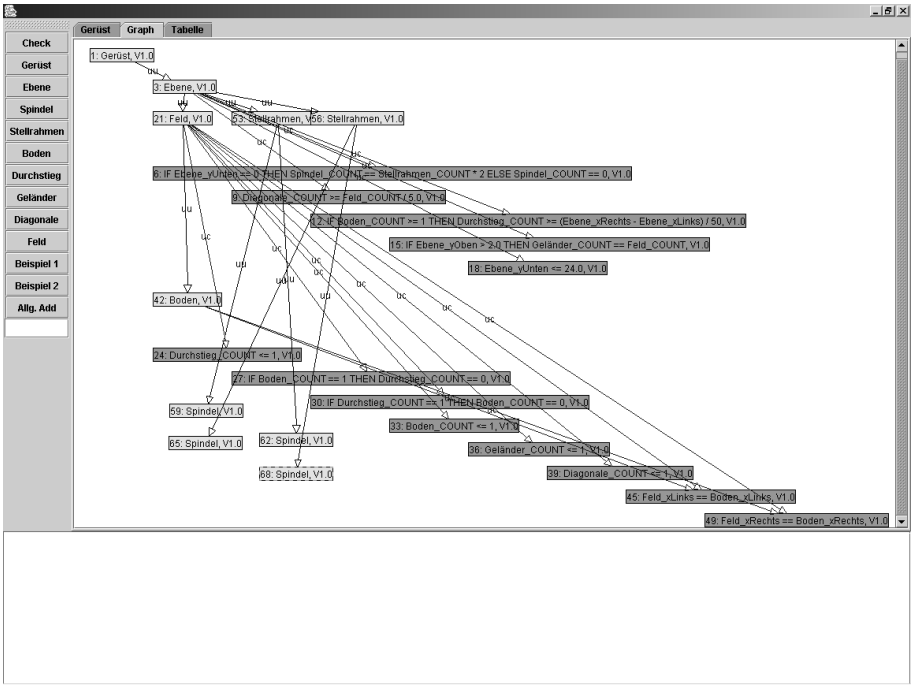

Abbildung 11.18: Graph des Configuration Level nach dem Einfügen der Bauteile

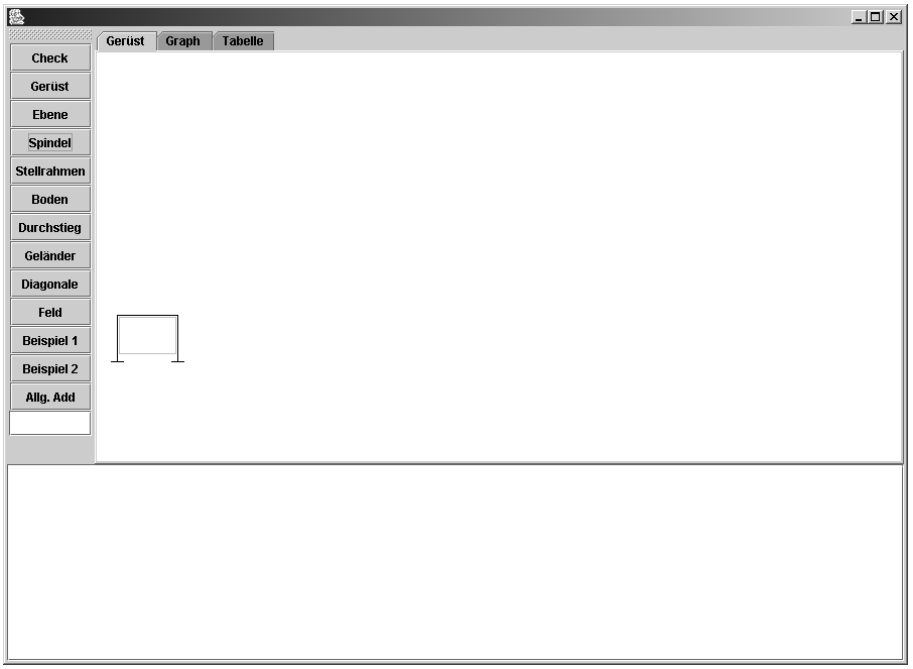

Abbildung 11.19: Darstellung des Gerüsts nach dem Einfügen der Bauteile

Programmautomatische Prüfung größerer Fassadengerüste

Das erste Beispiel ist das Fassadengerüst der Abbildung 11.21. Dieses entspricht allen definierten Aufbauregeln. Einen Ausschnitt der Programmausgabe zeigt Abbildung 11.22.

Als zweites Beispiel dient das Fassadengerüst aus Abbildung 11.23. Dieses Fassadengerüst verletzt die Regel für die Anzahl an Diagonalen in einer Ebene. Die mittlere Ebene des Gerüsts ist nicht ausgesteift.

Abbildung 11.24 zeigt den relevanten Ausschnitt der Programmausgabe im Verlauf der programmautomatischen Prüfung. Im Verlauf der Prüfung der Gültigkeitsregeln wird festgestellt, dass die betreffende Gültigkeitsregel verletzt wird.

```
Prüfe XS(1)
        Boden; ID = 42
        Feld; ID = 21
Erfüllt: Feld_xLinks == Boden_xLinks
        Boden; ID = 42
        Feld; ID = 21
Erfüllt: Feld_xRechts == Boden_xRechts
        Feld; ID = 21
Erfüllt: Durchstieg_COUNT <= 1
        Feld; ID = 21
Erfüllt: IF Boden_COUNT == 1 THEN Durchstieg_COUNT == 0
        Feld; ID = 21
Erfüllt: IF Durchstieg_COUNT == 1 THEN Boden_COUNT == 0
        Feld; ID = 21
Erfüllt: Boden_COUNT <= 1
        Feld; ID = 21
Erfüllt: Geländer_COUNT <= 1
        Feld; ID = 21
Erfüllt: Diagonale_COUNT <= 1
        Ebene; ID = 3
Erfüllt: IF Ebene_yUnten == 0 THEN Spindel_COUNT == Stellrahmen_COUNT * 2
         ELSE Spindel_COUNT == 0
        Ebene; ID = 3
*** Nicht erfüllt: Diagonale_COUNT >= Feld_COUNT / 5.0 ***
        Ebene; ID = 3
*** Nicht erfüllt: IF Boden_COUNT >= 1
         THEN Durchstieg_COUNT >= (Ebene_xRechts - Ebene_xLinks) / 50 ***
        Ebene; ID = 3
Erfüllt: IF Ebene_yOben > 2.0 THEN Geländer_COUNT == Feld_COUNT
        Ebene; ID = 3
Erfüllt: Ebene_yUnten <= 24.0
```

Abbildung 11.20: Auszug aus dem Ergebnis einer ersten programmautomatischen Prüfung

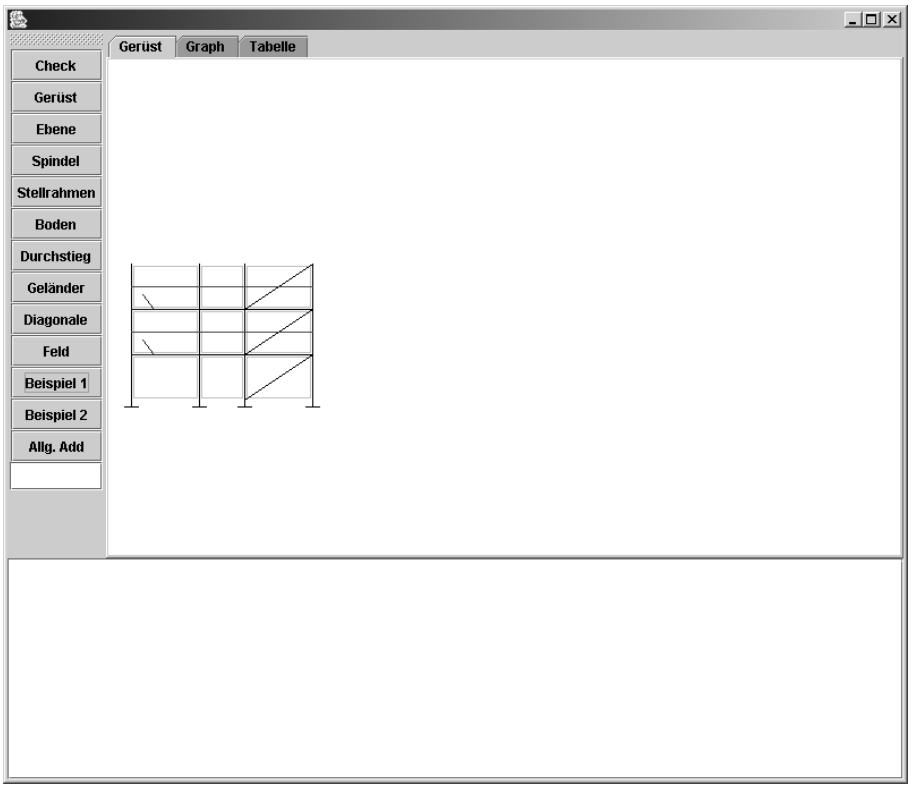

Abbildung 11.21: Beispiel einer gültigen Konfiguration

```
Prüfe XS(1)
.
.
.
        Boden; ID = 304
        Feld; ID = 78
Erfüllt: Feld_xLinks == Boden_xLinks
        Boden; ID = 304
        Feld; ID = 78
Erfüllt: Feld_xRechts == Boden_xRechts
        Feld; ID = 78
Erfüllt: Durchstieg_COUNT <= 1
        Feld; ID = 78
Erfüllt: IF Boden_COUNT == 1 THEN Durchstieg_COUNT == 0
        Feld; ID = 78
Erfüllt: IF Durchstieg_COUNT == 1 THEN Boden_COUNT == 0
        Feld; ID = 78
Erfüllt: Boden_COUNT <= 1
        Feld; ID = 78
Erfüllt: Geländer_COUNT <= 1
        Feld; ID = 78
Erfüllt: Diagonale_COUNT <= 1
        Boden; ID = 315
        Feld; ID = 99
Erfüllt: Feld_xLinks == Boden_xLinks
        Boden; ID = 315
        Feld; ID = 99
Erfüllt: Feld_xRechts == Boden_xRechts
        Diagonale; ID = 414
        Feld; ID = 99
Erfüllt: sqrt((Diagonale_xRechts - Diagonale_xLinks)^2
            +(Diagonale_yOben - Diagonale_yUnten)^2) ==
         sqrt((Feld_xRechts - Feld_xLinks)^2+(Feld_yOben - Feld_yUnten)^2)
        Feld; ID = 99
.
.
.
```

Abbildung 11.22: Auszug aus dem Ergebnis der programmautomatischen Prüfung der gültigen Konfiguration

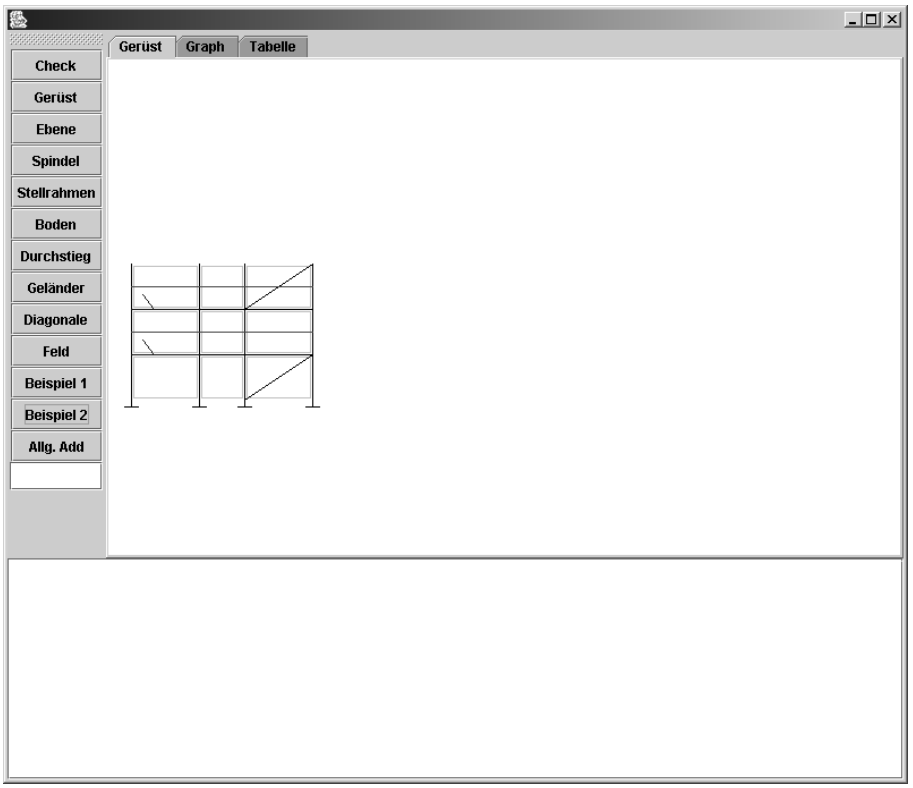

Abbildung 11.23: Beispiel einer nicht gültigen Konfiguration

```
Prüfe XS(1)
.
.
.
        Feld; ID = 162
Erfüllt: Geländer_COUNT <= 1
        Feld; ID = 162
Erfüllt: Diagonale_COUNT <= 1
        Ebene; ID = 21
Erfüllt: IF Ebene_yUnten == 0 THEN Spindel_COUNT ==
         Stellrahmen_COUNT * 2 ELSE Spindel_COUNT == 0
        Ebene; ID = 21
*** Nicht erfüllt: Diagonale_COUNT >= Feld_COUNT / 5.0 ***
        Ebene; ID = 21
Erfüllt: IF Boden_COUNT >= 1 THEN Durchstieg_COUNT >=
         (Ebene_xRechts - Ebene_xLinks) / 50
        Ebene; ID = 21
Erfüllt: IF Ebene_yOben > 2.0 THEN Geländer_COUNT == Feld_COUNT
        Ebene; ID = 21
Erfüllt: Ebene_yUnten <= 24.0
        Geländer; ID = 381
        Feld; ID = 183
Erfüllt: Feld_xLinks == Geländer_xLinks
        Geländer; ID = 381
        Feld; ID = 183
Erfüllt: Feld_xRechts == Geländer_xRechts
.
.
.
```

Abbildung 11.24: Auszug aus dem Ergebnis der programmautomatischen Prüfung der nicht gültigen Konfiguration

Kapitel 12

Zusammenfassung und Ausblick

Ein neues Konzept für eine vereinfachte Integration von Fachwissen in Computerprogramme wird vorgestellt. Ergebnis ist ein Planungssystem zur Unterstützung von Planungsprozessen. Das System ermöglicht die Abbildung von Produktzusammensetzungen einschliesslich ihrer Abhängigkeiten untereinander. Die Produktzusammensetzung kann durch Gültigkeitsregeln vervollständigt werden. Damit ist eine computerunterstützte Planung mit der Möglichkeit programmautomatischer Überprüfung der Gültigkeitsregeln möglich. Weiterhin ermöglicht das System die Verwaltung zahlreicher Produktfamilien und Produktversionen. Damit wird zum einen berücksichtigt, dass unterschiedliche Produktfamilien zur Planung zur Verfügung stehen und dass zahlreiche benutzer- beziehungsweise auftragsspezifische Konfigurationen geplant und verwaltet werden müssen. Zum anderen wird berücksichtigt, dass Produktkomponenten ständig weiterentwickelt werden und somit in neuen Versionen zur Verfügung stehen.

Die praktische Anwendbarkeit wird durch die Orientierung bei der Eingabe der Produktzusammensetzung an bekannten Strukturen erreicht.

Da die Regeln der Produktzusammensetzung von den Experten des Anwendungsgebiets in das System eingegeben werden, entfällt eine aufwändige Weitergabe dieses umfangreichen Fachwissens an die Experten der Softwareentwicklung. Fachspezifische Anwenderprogramme können auf der Grundlage dieses Systems schneller erstellt werden.

Ausblick

Für eine Weiterentwicklung des vorgestellten Planungssystems können folgende Bereiche genannt werden.

In Tiefenrichtung

- Erweiterung des Sprachumfangs der Gültigkeitsregeln.

- Regelbasierte Anwenderführung.
- Erweiterung zur Lösung von Optimierungsaufgaben.

In Breitenrichtung

- Anwendung auf verschiedenste Bereiche im Bauwesen und außerhalb.

Anhang A

Computer Supported Cooperative Work und Groupware

In diesem Kapitel werden die grundlegenden Begriffe der Computer Supported Cooperative Work (CSCW) erklärt. Anschliessend werden die unterschiedlichen CSCW-Systeme Kategorisiert und die Anforderungen an CSCW-Systeme werden formuliert.

A.1 Allgemeines

Stein führt in [54] aus, dass Computer Supported Cooperative Work, kurz CSCW, ein interdisziplinäres Forschungsgebiet der Bereiche Informatik, Soziologie, Psychologie, Arbeits- und Organisationswissenschaften, Anthropologie, Ethnograhie und der Wirtschaftswissenschaften ist. CSCW beschäftigt sich mit Gruppenarbeit und mit Informations- beziehungsweise Kommunikationstechnologie, die diese Gruppenarbeit unterstützt. Groupware bezeichnet eine Kategorie von Softwareprodukten, die die im Forschungsgebiet der CSCW gewonnenen Erkenntnisse in ein Informations- und Kommunikationssystem umsetzen [47, 54].

A.2 Begriffe

A.2.1 Gruppe

Mayer beschreibt in [46] die Unterscheidung in Gruppen, Arbeitsgruppen und Teams. Eine *Gruppe* besteht aus zwei oder mehreren Personen, die sich kennen, miteinander agieren und sich gegenseitig beeinflussen. Eine *Arbeitsgruppe* ist eine Gruppe aus zwei oder mehreren Personen, die zwecks einer gemeinsamen Aufgabe interagieren. Ein *Team* ist eine Arbeitsgruppe, deren Mitglieder ein gemeinsames Ziel verfolgen.

A.2.2 Gruppenarbeit

Die *Gruppenarbeit* setzt sich aus

- Gruppentätigkeit,
- Gruppenaufgabe und
- Gruppenziel

zusammen. Die Gruppenarbeit bedient sich *Gruppenprozessen*, die sich in die Bereiche

- Kommunikation,
- Koordination und
- Kooperation

einteilen lassen. Die Prozesse der einzelnen Bereiche bauen hierbei aufeinander auf. Die Kooperation setzt Koordination voraus, die Koordination benötigt Kommunikation.

Kommunikation ist der Austausch von Informationen zwischen mehreren Mitgliedern einer Gruppe. Diese Kommunikation mit Sender und Empfänger benötigt weder ein gemeinsames Material nach ein gemeinsames Ziel.

Koordination basiert auf Kommunikation. Ziel der Koordination ist es, neben Informationen auch andere Resourcen, beispielsweise das gemeinsame Material bereitzustellen. Konkurrierende Tätigkeiten am gemeinsamen Material werden durch Koordination aufeinander abgestimmt. Koordination setzt kein gemeinsames Ziel voraus.

Kooperation ist eine spezielle Form der Kommunikation, für die Koordinationsprozesse erforderlich sind. Sie basiert auf der Existenz eines gemeinsamen Materials und eines gemeinsamen Ziels. Bei der Kooperation wird unterschieden in

- Implizite Kooperation, die sich durch Arbeit am gemeinsamen Material auszeichnet und
- Explizite Kooperation, die durch bewusstes Austauschen von Informationen gekennzeichnet ist.

	Zur selben Zeit	Zu unterschiedlichen Zeitpunkten
Am selben Ort	Entscheidungsraum Gemeinsames Editieren	Zeit- und Aufgabenmanagement
An verschiedenen Orten	Workflow-Management Verteiltes gemeinsames Editieren	Asynchrone Konferenzsysteme E-Mail

Tabelle A.1: Klassifikation von CSCW-Systemen

A.3 Klassifikation von CSCW-Systemen

Eine im Forschungsgebiet der CSCW-Systeme verbreitete Kategorisierung ist die *Raum-Zeit-Klassifikation*, wie [6] beschreibt. Hierbei werden die Systeme in einer Raum-Zeit-Matrix angeordnet (vgl. Tabelle A.1).

Eine weitere Möglichkeit der Klassifikation ist die *Klassifikation nach Unterstützungsfunktionen*. Unterschiedliche CSCW-Systeme unterstützen die drei grundlegenden Gruppenprozesse Kommunikation, Koordination und Kooperation in unterschiedlicher Art. Demnach können die CSCW-Applikationen an verschiedenen Positionen in einem Dreieck angeordnet werden, dessen Ecken die Unterstützung der drei Gruppenprozesse darstellt (Abbildung A.1).

A.4 Anforderungen an CSCW-Systeme

CSCW-Systeme können als Werkzeug in den drei verschiedenen Bereichen der Gruppenprozesse eingesetzt werden.

A.4.1 Kommunikationsunterstützung

Wie Müller in [47] beschreibt, wird der Rechner hierbei zur Unterstützung bei der Übermittlung, Verwaltung und Speicherung von Informationen eingesetzt. Programme zur Kommunikationsunterstützung müssen sowohl asynchrone als auch synchrone Zugriffe auf Informationen unterstützen. Beispiele für Systeme zur Kommunikationsunterstützung sind E-Mail-Systeme für den asynchronen Austausch von Nachrichten oder EDI-Systeme zum Austausch und der gemeinsamen Nutzung von Papierdokumenten nach Umwandlung in elektronische Dokumente.

A.4.2 Koordinationsunterstützung

Systeme zur Koordinationsunterstützung dienen der Strukturierung von Informationen und der Verwaltung von Ressourcen. Beispiele hierfür sind Systeme, die Vor-

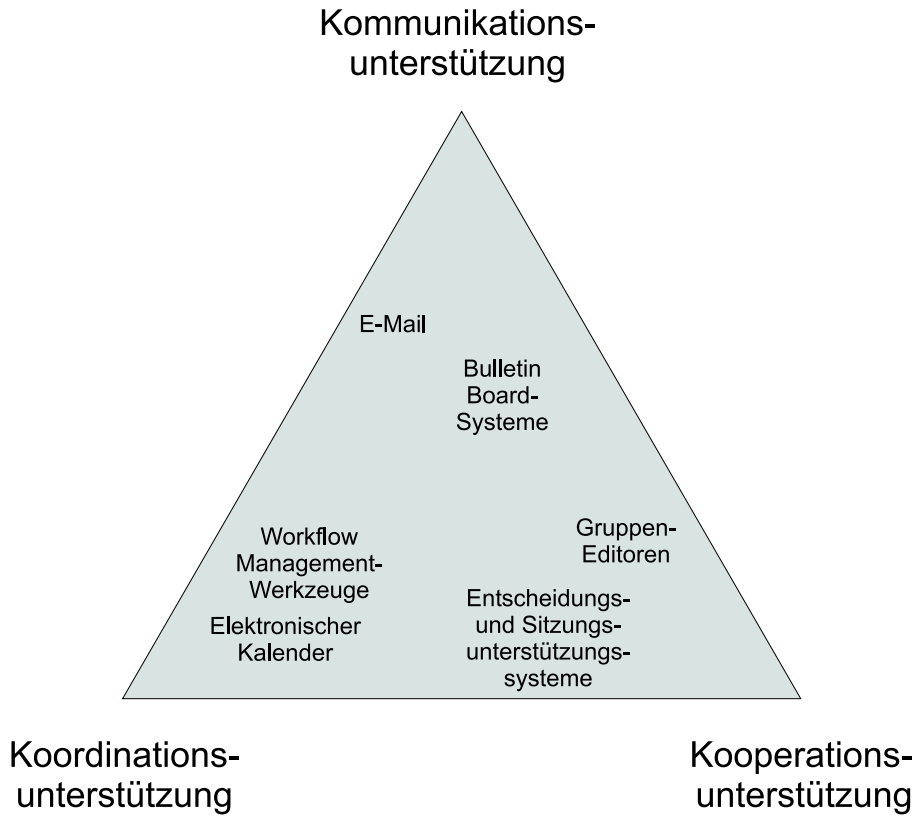

Abbildung A.1: Klassifikationsschema nach Unterstützungsfunktionen

gänge steuern und das automatische Routing von Dokumenten an Bearbeiter unterstützen. Gruppen-Terminkalender, die einen elektronischen Terminkalender zur Verfügung stellen und Projektmanagement-Systeme, die bei der Planung, der Koordination und der Verfolgung von Aufgaben helfen, sind weitere Vertreter der Koordinationsunterstützenden Systeme.

A.4.3 Kooperationsunterstützung

Bei Systemen zur Kooperationsunterstützung wird der Rechner zur Auflösung von Konflikten eingesetzt [47]. Programme zur Kooperationsunterstützung können in Gruppeneditoren und Systeme zur Entscheidungsunterstützung eingeteilt werden. Mehrautorensysteme müssen synchrone und asynchrone Bearbeitung des gemeinsamen Materials unterstützen.

Anhang B

Objektorientierte Analyse

B.1 Allgemeines

Die Objektorientierte Analyse nach [10], kurz OOA, baut auf Konzepten auf, die dem Menschen vertraut sind: Objekte und Eigenschaften, Ganze und Teile, Klassen und Elemente. Das Ziel einer Objektorientierten Analyse ist es, ein Anwendungsgebiet zu verstehen. Nach Abschluss der Analyse liegt das Wissen über alle relevanten Aspekte eines Anwendungsgebiets im Hinblick auf alle benötigten Systemaufgaben in einer Sammlung von *Modellschichten* vor. Bei diesen Modellschichten handelt es sich um

- Subjekte,
- Klassen und Objekte,
- Strukturen,
- Attribute und
- Services.

B.2 Klassen und Objekte

Die Begriffe Klassen und Objekte beziehen sich in der OOA auf das Anwendungsgebiet und die Systemaufgaben. Coad und Yourdon definieren hierzu in [10]:

Das Anwendungsgebiet ist der Tätigkeitsbereich, der untersucht wird.

Die Systemaufgaben sind eine Gruppe von Dingen, für die Verantwortlichkeit besteht und die als Einheit/Ganzes zu sehen sind.

Bei der OOA wird das Anwendungsgebiet untersucht und das Wissen über dieses Anwendungsgebiet so lange gefiltert, bis nur noch die zu modellierenden Aufgaben des Systems übrig bleiben. Damit ergibt sich nach [10] für die Begriffe *Objekt* und *Klasse*:

Ein Objekt ist die Abstraktion eines Etwas innerhalb des betreffenden Anwendungsgebiets, welches die Fähigkeit des Systems beschreibt, Informationen über dieses Etwas zu speichern und/oder mit dem Etwas zu kommunizieren. Weiterhin ist ein Objekt die Kapselung von Attributwerten und ihren exklusiven Services.

Eine Klasse ist die Beschreibung von einem oder mehreren Objekten, die gemeinsame Attribute und Services aufweisen. Hierzu gehört auch eine Definition, wie neue Objekte dieser Klasse erzeugt werden können.

Eine Darstellung mit Klassen und Objekten ist dadurch begründet, dass die technische Darstellung eines Systems so nah wie möglich an der vom Anwender empfundenen Realität bleiben soll. Ein entsprechendes OOA-Modell ist dann von allen Beteiligten leicht zu verstehen. Es dient dazu, Wissen über das Anwendungsgebiet zu gewinnen und dieses Wissen anderen zu vermitteln. Dies gilt auch im Hinblick auf eine Wiederverwendbarkeit von Teilen bei ähnlichen Systemen. Weiterhin ist diese Darstellung mit Klassen und Objekten über längere Zeiträume stabil, auch wenn sich Attribute und Services ändern.

Um mögliche Klassen und Objekte zu finden, sollten folgende Bereiche geprüft werden:

Strukturen: Bei den Strukturen, die für die Identifizierung von Klassen und Objekten zu untersuchen sind, handelt es sich um *Generalisierungs-/Spezialisierungs-* Strukturen und *Ganzes-/Teile-* Strukturen.

Andere Systeme: Systeme, mit denen das System zusammenarbeiten soll, können als eine Klasse im System beschrieben werden.

Geräte: Geräte für den Datenaustausch oder den Austausch von Kontrollinformationen können ebenfalls als Klassen und Objekte abgebildet werden. Hierbei ist zu beachten, dass keine implementierungsspezifischen Computerkomponenten modelliert werden. Diese werden in der Phase des *Objektorienterten Design* betrachtet.

Dinge oder Ereignisse, die erinnert werden sollen: Hier wird überprüft, ob im Anwendungsgebiet und der vorgesehenen Systemaufgabe Ereignisse existieren, die festgehalten werden sollen.

Gespielte Rollen: Innerhalb dieses Bereichs werden die Rollen betrachtet, die Menschen im System spielen. Hierbei sind Menschen entweder Benutzer des Systems oder Personen, über die das System Informationen speichert.

Arbeitsanleitungen: Weitere Quelle für mögliche Klassen und Objekte liefert die Fragestellung, ob das System Arbeitsanleitungen speichern soll. Dies ist relevant, sobald Arbeitsabläufe notwendig sind, die nicht durch normale Gesetzmässigkeiten des Anwendungsgebiets vorgegeben sind.

Orte: Muss das System geographische Orte kennen? Diese können als Klassen und Objekte im System modelliert werden.

Organisationseinheiten: Hier ist zu untersuchen, zu welchen Organisationseinheiten die Benutzer des Systems gehören. Weiterhin ist zu klären, ob Informationen über Organisationseinheiten abespeichert werden sollen.

B.3 Strukturen

Strukturen beziehen sich im Bereich der OOA sowohl auf das Anwendungsgebiet als auch auf die Systemaufgaben. Nach [10] ist Struktur eine Darstellung der Komplexität innerhalb eines Anwendungsgebiets im Zusammenhang mit den Systemaufgaben. Der Begriff Struktur ist hierbei der Oberbegriff für die Generalisierungs-/ Spezialisierungs-Struktur und die Ganzes-/Teile-Struktur.

Generalisieruns-/Spezialisierungs-Strukturen können aus der Sicht der Spezalisierung als "... ist ein ..." gelesen werden. Ein Beispiel hierfür ist die Beziehung "Ein LKW ist ein Fahrzeug". Hierbei ist "Fahrzeug" die Generalisierung und "LKW" die Spezialisierung. Innerhalb von Generalisierungs-/ Spezialisierungs-Strukturen existiert ein Vererbungsmachanismus.

Ganzes-/Teile-Strukturen können aus der Sicht des Ganzen als "... hat ein ..." gelesen werden. Ein Beispiel hierfür ist die Beziehung "Ein Fahrzeug hat einen Motor". Hierbei ist das Ganze "Fahrzeug" und ein Teil der "Motor".

Strukturen dienen dazu, die Komplexität des Zusammenwirkens mehrerer Klassen und Objekte zu erfassen. Weiterhin dient die Generalisierungs-/Spezialisierungs-Struktur mit ihren Vererbungsbeziehungen einer Beschreibung gemeinsamer Attribute und Services und ihrer anschliessenden Spezialisierung.

B.4 Subjekte

Subjekte dienen in der OOA dazu, Leser durch ein grosses und komplexes System zu führen und nach ersten OOA-Arbeiten an einem Projekt dieses in kleinere Arbeitspakete zu unterteilen. Die Subjekte dienen nur dazu, bei grossen OOA-Projekten den Überblick zu wahren.

B.5 Attribute

Attribute beschreiben die Zustandsinformationen, die für jedes Objekt einer Klasse einen Wert besitzen. Die Attribute werden nur durch die exklusiven Services des

Objekts gelesen oder geändert. Wenn ein anderes Teil des Systems auf Attributwerte zugreifen will, so ist dies nur über eine Nachrichtenverbindung zu einem Service des Objekts möglich. Dies führt zu schmalen und gut definierten Schnittstellen.

B.6 Services

Ein Service ist in [10] definiert als ein bestimmtes Verhalten, für dessen Durchführung ein Objekt verantwortlich ist. Die Definition eines Services erfolgt durch die Definition des erforderlichen Verhaltens und der Definition der notwendigen Kommunikation zwischen den verschiedenen Objekten. Die Services zeigen, welches Verhalten ein Objekt einer Klasse haben wird.

Anhang C

Objektorientiertes Design

C.1 Allgemeines

Das Ziel des Objektorientierten Design ist die Umsetzung der Ergebnisse der Objektorientierten Analyse für eine bestimmte Hardware oder Software [11]. Wie Goos in [26] darlegt, wird beim Objektorientierten Entwurf ein System als Menge kooperierender Objekte aufgefasst.

Dieses Kapitel erläutert einige wesentliche Elemente und Grundbegriffe. Zur Vertiefung wird auf [11] verwiesen.

C.1.1 Objekte und Klassen

Nach Goos besteht ein Informatik-System aus Komponenten, die zueinander in Beziehung stehen [26]. Die Komponenten sind Teilsysteme und elementare Teilsysteme sind *Objekte*. Zwischen den Teilsystemen bestehen statische und dynamische Beziehungen. Aussagen über die statischen Beziehungen werden in Objektmodellen und Aussagen über die dynamischen Beziehungen werden in Verhaltensmodellen dargestellt [1].

Ein Objekt kann einen Gegenstand der realen Welt repräsentieren. Es besitzt besitzt durch Werte erfassbare Eigenschaften, die *Attribute* und Tätigkeiten, die es ausführen kann, die *Methoden*. Die Attribute und die Methoden bilden zusammen die Merkmale eines Objekts. Besitzen mehrere Objekte die gleichen Merkmale und das gleiche Verhalten, so gehören sie zur gleichen *Klasse*. Die Objekte werden dann als *Ausprägungen*[2] der Klasse bezeichnet. Beim *Objektbasierten Programmieren* reichen diese Begriffe zur Beschreibung von Objekten und dem Gesamtsystem aus. Beim *Objektorientierten Entwurf* wird das objektbasierte Vorgehen erweitert um

[1] Die Beschreibung der Systeme kann beispielsweise mit der Unified Modeling Language (UML) erfolgen.

[2] Goos verwendet hier den Begriff *Ausprägungen* und weist darauf hin, dass hierfür auch der Begriff *Instanz* verwendet wird.

die Aspekte der Klassifikation von Objekten, Teilsystemen und Systemen mit überwiegend gleichem Aussenverhalten ohne Beachtung der internen Realisierung.

C.1.2 Vererbung

Mit dem Begriff der *Vererbung* werden unterschiedliche Beziehungen zwischen Klassen ausgedrückt.

A **ist ein** B beschreibt eine Beziehung, in der eine Klasse A alle Merkmale der Klasse B übernimmt.

A **ist eine Spezialisierung von** B drückt aus, dass A gegenüber B zur Erzielung eines spezifischeren Verhaltens eingeschränkt ist.

A **implementiert** B sagt aus, dass A die Konzepte von B realisiert.

A **verwendet** B bedeutet, dass A den Code von B verwendet.

Im Falle einer *Mehrfachvererbung* erbt eine Klasse die Verhaltensweisen mehrerer Klassen.

C.1.3 Polymorphie

Die wesentlichen Formen der Polymorphie sind die statische Bindung, auch überladen von Methoden genannt und die dynamische Bindung. Das Überladen von Methoden ermöglicht es, in einer abgeleiteten Klasse Methoden der Oberklasse zu überschreiben um somit beispielsweise eine Spezialisierung zu implementieren. Massgebend ist hierbei, dass bereits zum Zeitpunkt der Übersetzung des Programmcodes bekannt ist, welche Methode aufgerufen werden muss. Mit der dynamischen Bindung ist es möglich, einen einheitlichen Gebrauch von Objekten gleichen Aussenverhaltens zu erreichen. Massgebend ist hierbei, dass zum Zeitpunkt der Übersetzung nicht bekannt ist, welche Methode auszuführen ist.

Anhang D

UML - Unified Modeling Language

D.1 Allgemeines

UML, die Unified Modeling Language ist eine Sprache zur Beschreibung von Softwaresystemen. Ziel von UML war es, alle Softwaresysteme darstellen zu können [57]. Eine umfassende Darstellung von UML kann [50] entnommen werden. Für eine Notationsübersicht wird auf [56] verwiesen. Die Spezifikationen der UML sind in [55] dokuentiert.

In UML werden Diagramme zur Darstellung verwendet, die im folgenden einführend dargestellt werden. Die verschiedenen Diagramme und ihre Beziehungen untereinander sind in Abbildung D.1 dargestellt.

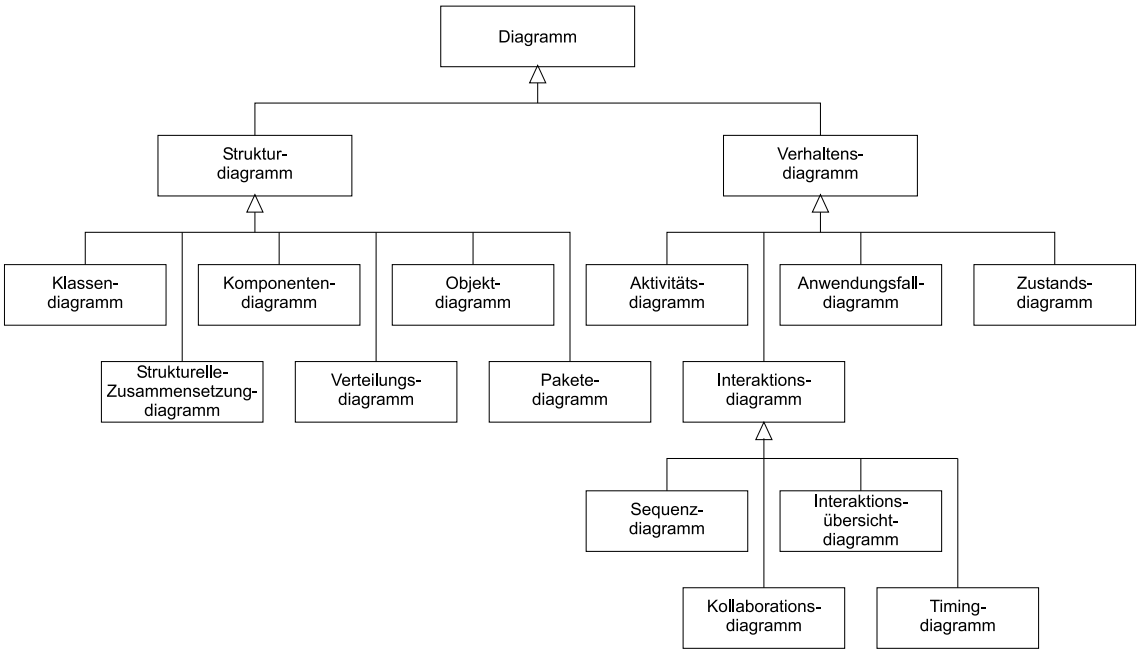

Abbildung D.1: Taxonomie der Diagramme

Klassendiagramme beschreiben die statische Struktur der Objekte und die Beziehungen untereinander. Die Elemente zur Darstellung der Struktur werden in den nachfolgenden Abschnitten kurz erläutert.

Komponentendiagramme zeigen die Abhängigkeiten zwischen den Komponenten. Während Pakete-Diagramme ein allgemeines Mittel zur Strukturierung sind, zeigen die Komponentendiagramme Aspekte der Implementierung.

Objekt-Diagramme dokumentieren Objekte und ihre Beziehungen zueinander zu einem bestimmten Zeitpunkt.

Strukturelle-Zusammensetzung-Diagramme zeigen die interne Struktur von Klassen, Schnittstellen und Komponenten. Dabei zeigen sie die, wie die einzelnen Bestandteile miteinander in Beziehung stehen, um das Verhalten der Klasse, der Schnittstelle oder der Komponente zu erreichen.

Verteilungs-Diagramme werden zur Darstellung der Hardware eingesetzt. In diese Verteilungs-Diagramme können auch die Komponenten und ihre Abhängigkeiten eingetragen werden. Damit erhält man einen Überblick, wo die jeweiligen Komponenten und Objekte ausgeführt werden.

Pakete-Diagramme dienen der Strukturierung. Sie geben beispielsweise bei grossen Systemen einen Überblick über die verschiedenen Übersetzungseinheiten.

Aktivitätsdiagramme zeigen Aktivitäten, Zustände und Zustandsübergänge.

Anwendungsfall-Diagramme dienen der Beschreibung der Geschäftsprozesse und allgemeiner Einsatzmöglichkeiten. Sie zeigen das Verhalten des Systems nach aussen und somit insbesondere das Zusammenwirken des Systems mit Personen. Die Beschreibung erfolgt zumeist über mehrere Anwendungs-Szenarien.

Zustands-Diagramme stellen das dynamische Verhalten des Systems dar. Sie werden aus den Interaktionsdiagrammen entwickelt.

Sequenz-Diagramme zeigen Klassen und ihre Beziehungen und den zeitlichen Ablauf ihres Nachrichtenaustauschs.

Interaktions-Übersicht-Diagramme zeigen die Interaktionen mit dem Ziel, eine Übersicht über den Kontrollfluss darzustellen.

Kollaborationsdiagramme zeigen Objekte und ihre Beziehungen und den räumlichen Ablauf ihres Nachrichtenaustauschs.

Timing-Diagramme zeigen die Zustandsänderungen von Objekten in ihrem zeitlichen Verlauf, die als Reaktion des Objekts auf eine Nachricht folgen.

D.2 Übersicht über die wichtigsten Notationen

D.2.1 Pakete

Pakete fassen Modellelemente beliebigen Typs zu überschaubaren Einheiten zusammen. Pakete können hierarchich gegliedert werden und hierfür andere Pakete beinhalten. Das oberste Paket beinhaltet das gesamte System. Die Darstellung von Paketen zeigt Abbildung D.2. Ziel für die Gliederung in Pakete ist es, die Übersicht über grosse Modelle zu verbessern. Die Aufteilung unterschiedlicher Modellelemente in Pakete soll nach logischen Gesichtspunkten erfolgen.

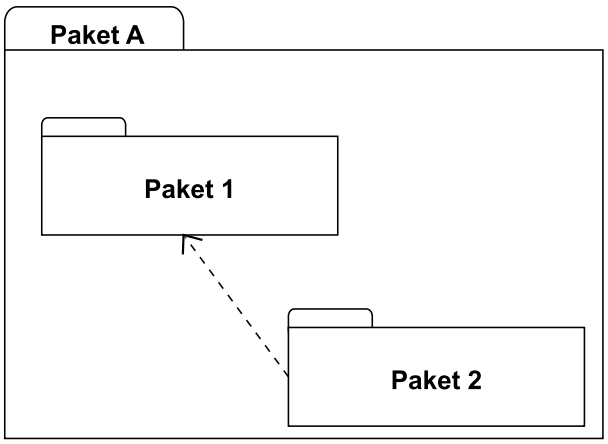

Abbildung D.2: Pakete

D.2.2 Komponenten

Eine Komponente ist in UML ein Modul mit eigener Identität und mit definierten Schnittstellen. Sie enthält eine oder mehrere Klassen. Im Unterschied zu Klassen sollen Komponenten austauschbar sein. Beispiele für die Darstellung von Komponenten zeigt Abbildung D.3.

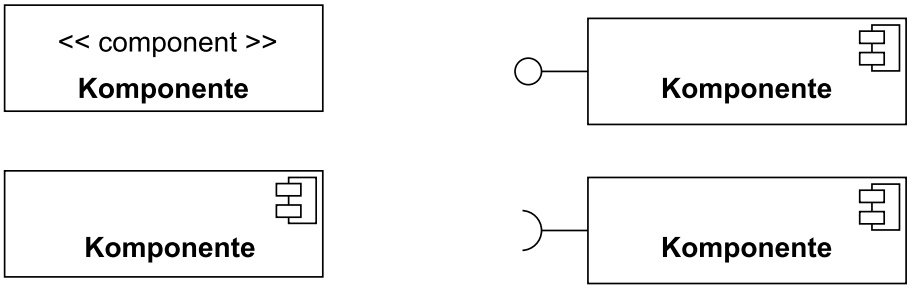

Abbildung D.3: Komponenten

D.2.3 Klassen, Schnittstellen und Instanzen von Klassen

Abbildung D.4 zeigt die Darstellung von Klassen. Zur Definition der Sichtbarkeit von Attributen und Operationen können den Attributnamen beziehungsweise den Namen der Operationen die Symbole der Tabelle D.1 vorangestellt werden.

Abbildung D.4: Klassen

+	public element
#	protected element
−	private element
∼	package element

Tabelle D.1: Sichtbarkeit

Die Notation von Schnittstellen erfolgt wie in Abbildung D.5 gezeigt.

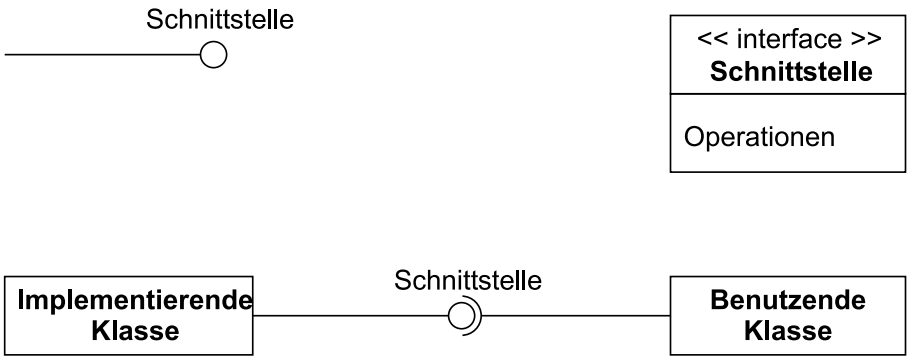

Abbildung D.5: Schnittstellen

Die Darstellung von Objekten zeigt Abbildung D.6.

D.2.4 Notiz

Notizen enthalten Anmerkungen, Kommentare, Erläuterungen oder zusätzliche Beschreibungstexte zu beliebigen Modellelementen.

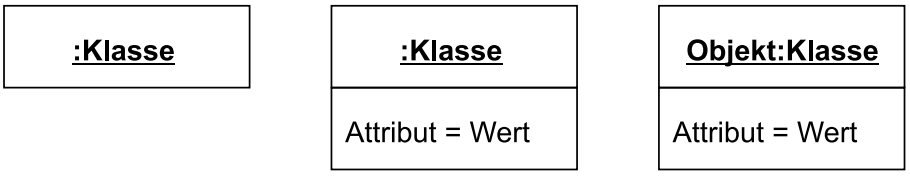

Abbildung D.6: Objekte

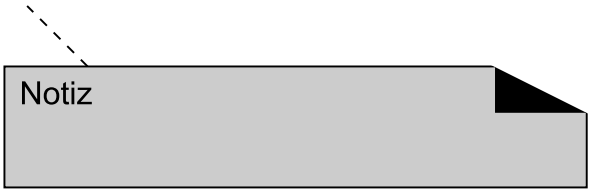

Abbildung D.7: Notiz

D.2.5 Beziehungen von Elementen zueinander

Abbildung D.8 gibt einen Überblick über die möglichen Assoziationen. Die Bezeichnung der Assoziation kann um die Angabe der Leserichtung ergänzt werden. Dies erfolgt durch ein ausgefülltes Dreieck.

Die Assoziationen können an den jeweiligen Enden durch die Angabe der Multiplizität der Beziehung ergänzt werden. Beispiele für mögliche Angaben dokumentiert Abbildung D.9. Die Multiplizität wird als einzelne Zahl oder als Wertebereich auf jeder Seite der Assoziation notiert.

Die Darstellung einer Vererbungshierarchie zeigt Abbildung D.10.

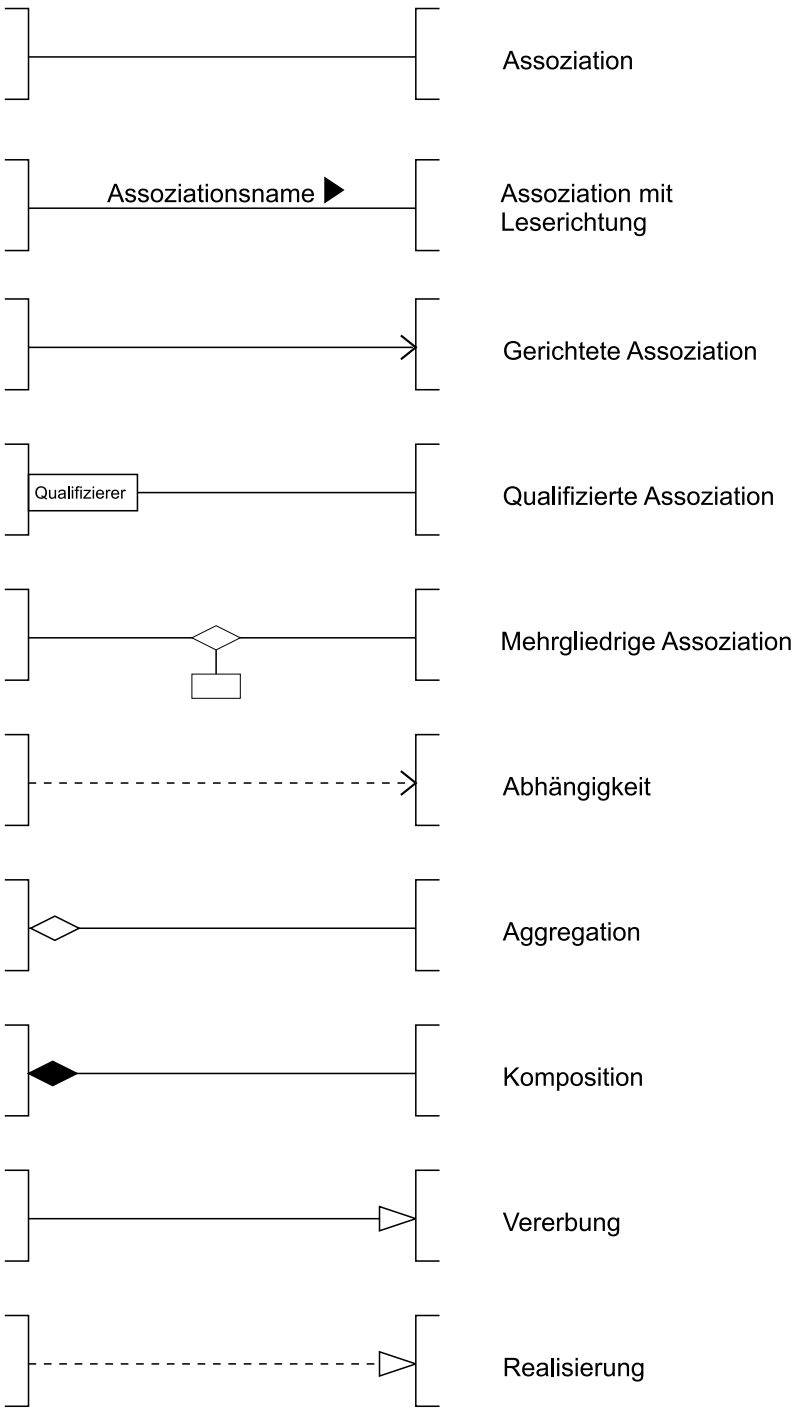

Abbildung D.8: Assoziationen

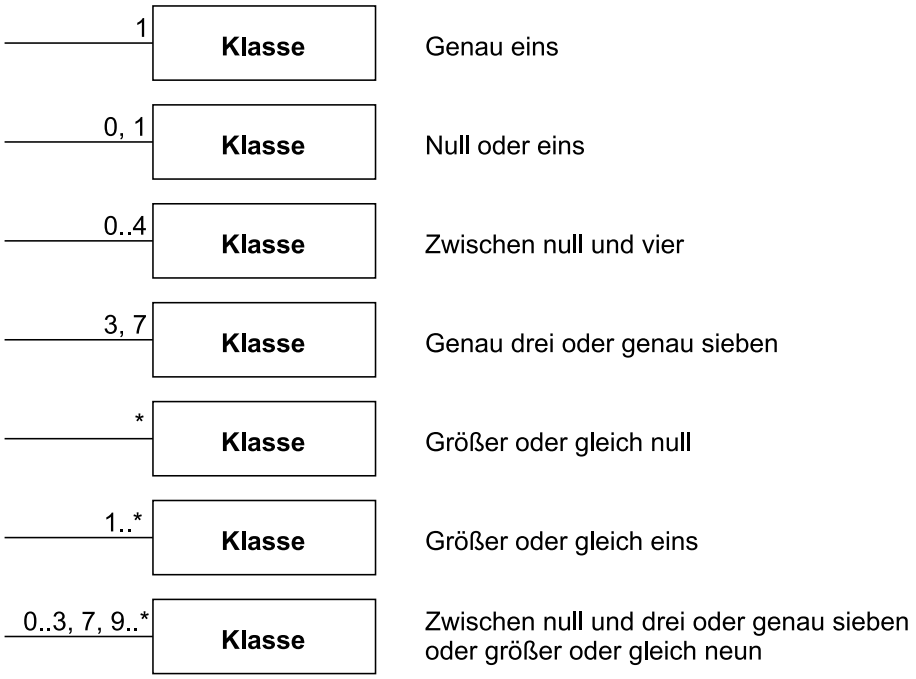

Abbildung D.9: Multiplizitäten aus [50]

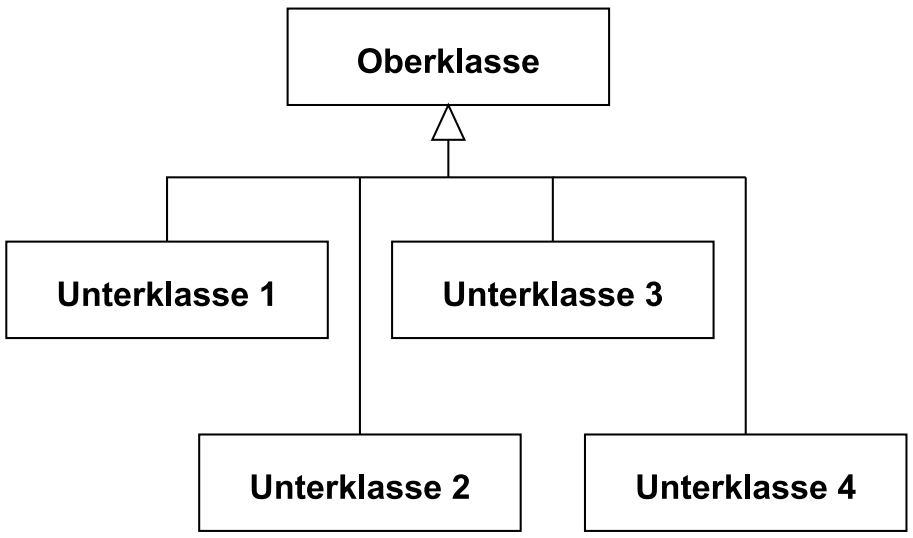

Abbildung D.10: Vererbung

Anhang E

Grammatiken

E.1 Allgemeines

Dieses Kapitel fasst die wesentlichen Grundlagen über Grammatiken zusammen. Grundlage hierfür waren die Arbeiten von Albert el al. [1] und von Kopp [37]. Für eine Darstellung der Operationen mit Mengen und Mengensystemen sei auf [7] verwiesen.

E.2 Grundlagen

Nach [1] dient die Backus-Naur-Form (BNF) zur Beschreibung der Regeln einer Grammatik. Jeder dieser Regeln besteht aus einer linken und einer rechten Seite, getrennt durch das Symbol ::=. Ersetzungsalternativen auf der rechten Seite werden durch das Symbol | voneinander getrennt.

Ausser diesen Hilfssymbolen sind alle weiteren Zeichen entweder

Terminalzeichen oder

Nichtterminalzeichen, dieses sind Mengen von Zeichenreihen über dem Alphabet der Terminalzeichen.

Die Regeln – auch Produktionen genannt – geben an, wie ausgehend von einem speziellen Nichtterminalzeichen, dem Startsymbol, durch wiederholtes Ersetzen eine zulässige Terminalzeichenreihe entsteht.

E.2.1 Beispiel aus [1]

Wir definieren die Menge der Ausdrücke mit zwei Variablen a und b, den Zahlkonstanten 0,...,9 und den Operatoren +, −, ∗ und : durch die folgenden Regeln.

```
1. <Ausdruck> ::= <Variable>|<Konstante>|
                  (<Ausdruck><Operator><Ausdruck>)
2. <Variable> ::= a|b
3. <Konstante> ::= 0|1|2|3|4|5|6|7|8|9
4. <Operator> ::= +|-|*|:
```

Die Startvariable S ist

```
<Ausdruck>.
```

Die Ableitung der Terminalzeichenreihe $(a + (a * b))$ ist damit:

```
<Ausdruck>
(<Ausdruck><Operator><Ausdruck>)      ; Regel 1
(<Variable><Operator><Ausdruck>)      ; Regel 1
(a<Operator><Ausdruck>)               ; Regel 2
(a+<Ausdruck>)                        ; Regel 4
(a+(<Ausdruck><Operator><Ausdruck>)) ; Regel 1
(a+(<Variable><Operator><Ausdruck>)) ; Regel 1
(a+(a<Operator><Ausdruck>))           ; Regel 2
(a+(a*<Ausdruck>))                    ; Regel 4
(a+(a*<Variable>))                    ; Regel 1
(a+(a*b))                             ; Regel 2
```

Bei der Ableitung wird beim Übergang von einer Zeile zur nächsten das erste Nichtterminalsymbol von links durch eine Alternative der rechten Seite einer anwendbaren Regel ersetzt. Daher nennt man diese Form der Ableitung eine *Linksableitung*.

E.3 Semi-Thue Systeme

Ein Semi-Thue[1] System ist ein Quadrupel

$$\Gamma = (V_N, V_T, R, w) \tag{E.1}$$

mit V_N : Endliche Menge von Nichtterminalzeichen
V_T : Endliche Menge von Terminalzeichen mit $V_N \cap V_T = \emptyset$
R : Endliche Menge von Regeln, es gilt $R \subseteq (V_N \cup V_T)^* \times (V_N \cup V_T)^*$
w : Das Startwort, es gilt $w \in (V_N \cup V_T)^*$

[1] benannt nach Axel Thue, norwegischer Mathematiker und Logiker [25]

Für die Regeln kann statt der BNF-Schreibweise $\alpha ::= \beta$ kann auch die Schreibweise $\alpha \to \beta$ verwendet werden. Damit kann die Anwendung einer Regel $\alpha \to \beta$ auf ein Wort w wie folgt definiert werden.

Kommt in w das Teilwort α vor, das heisst es gilt $w = x\alpha y$, dann kann α durch β ersetzt werden und es ergibt sich $x\beta y$. Man schreibt

$$x\alpha y \Rightarrow x\beta y.$$

Also gilt

$$\begin{aligned} & w \Rightarrow w' \\ & \exists x, y \in (V_N \cup V_T)^*, \alpha \to \beta \in R : \\ & w = x\alpha y \quad \text{und} \quad w' = x\beta y. \end{aligned}$$

Das Symbol $\Rightarrow$ zeigt an, dass genau ein Schritt zur Ableitung eines Wortes aus einem anderen Wort durgeführt wird. Für Ableitungsfolgen wird das Symbol $\overset{*}{\Rightarrow}$ verwendet.

Ist $\Gamma = (V_N, V_T, R, w)$ ein Semi-Thue System, dann heisst die Menge

$$L(\Gamma) = \{v \in V_T^* \mid w \overset{*}{\Rightarrow} v\} \tag{E.2}$$

die von Γ erzeugte Sprache.

E.4 Chomsky Grammatiken

Wenn das Startwort S einer Grammatik $G = (V_N, V_T, R, S)$ genau aus einem Nichtterminalzeichen $S \in V_N$ besteht und es in R keine Regeln der Form $\epsilon \to \alpha$ gibt, spricht man von einer *Chomsky Grammatik*. Hierbei ist ϵ das leere Wort. Die Chomsky Grammatiken lassen sich nach der Form ihrer Produktionen in einer Hierarchie klassifizieren.

Eine Chomsky Grammatik $G = (V_N, V_T, R, S)$ heisst

Chomsky Grammatik vom Typ 0 wenn die Regeln R keinen weiteren Einschränkungen unterliegen,

Chomsky Grammatik vom Typ 1 oder kontextsentitiv, wenn alle Regeln R die Form $\alpha A\beta \to \alpha\gamma\beta; \gamma \neq \epsilon$ oder $S \to \epsilon$ haben. Ist $S \to \epsilon$ in R enthalten, dann darf S in keiner rechten Seite von einer Regel in R auftreten,

Chomsky Grammatik vom Typ 2 oder kontextfrei, wenn alle Regeln aus R der Form $A \to \gamma$ sind,

Chomsky Grammatik vom Typ 3 oder regulär, wenn alle Regeln aus R der Form

$$\begin{aligned} A &\rightarrow Ba \\ A &\rightarrow a \end{aligned}$$

oder alle Regeln der Form

$$\begin{aligned} A &\rightarrow aB \\ A &\rightarrow a \end{aligned}$$

sind. Im ersten Fall spricht man von einer einseitig linkslinearen Grammatik, im zweiten Fall von einer einseitig rechtslinearen Grammatik.

Weiterhin gilt

$$\begin{aligned} A, B &\in V_N \\ a, b &\in V_T \\ \alpha, \beta, \gamma &\in (V_N \cup V_T)^* . \end{aligned}$$

Anhang F

Relationen und Graphen

F.1 Allgemeines

Dieser Anhang gibt einen Einblick in die Graphentheorie. Das in dieser Arbeit beschriebene Planungssystem bildet die grundlegenden Elemente in einem gerichteten azyklischen Graph (DAG - directed acyclic graph) ab. Aus diesem Grund werden in diesem Anhang Graphen und Algorithmen für gerichtete Graphen dargestellt. Für eine eingehendere Betrachtung sei auf [25] und [45] verwiesen.

F.2 Grundlagen

Ein Graph

$$G = (E, K)$$

besteht aus einer Menge E, den Ecken und einer Menge K, den Kanten. Der Graph heisst endlich, wenn E eine endliche Menge ist. Jede Kante gehört zu einem Paar Ecken und repräsentiert eine Relation zwischen den Ecken. Ein Graph kann ein gerichteter oder ein ungerichteter Graph sein.

Die Ecken in einem gerichteten Graph sind geordnete Paare, die Ecken in einem ungerichtetem Graph sind ungeordnete Paare.

Endliche ungerichtete Graphen können dadurch gezeichnet werden, dass die Ecken durch Punkte und die Kanten durch Linien gezeichnet werden (Abbildung F.1). Bei endlichen gerichteten Graphen ist eine Darstellung möglich, indem die Ecken als Punkte und die Kanten als Pfeile gezeichnet werden (Abbildung F.2).

F.3 Tiefensuche

Bei der Tiefensuche wird systematisch jeder Knoten eines Graphen besucht. Algorithmus F.1 aus [45] führt eine Tiefensuche in einem Graphen G vom Knoten v ausgehend aus. Den so entstehenden Tiefensuchbaum zeigt Abbildung F.3.

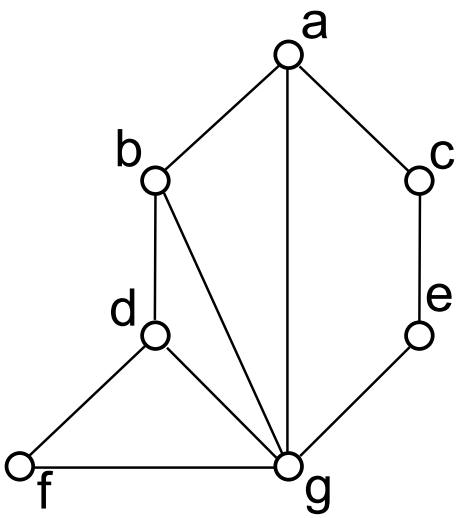

Abbildung F.1: Beispiel für einen ungerichteten Graphen

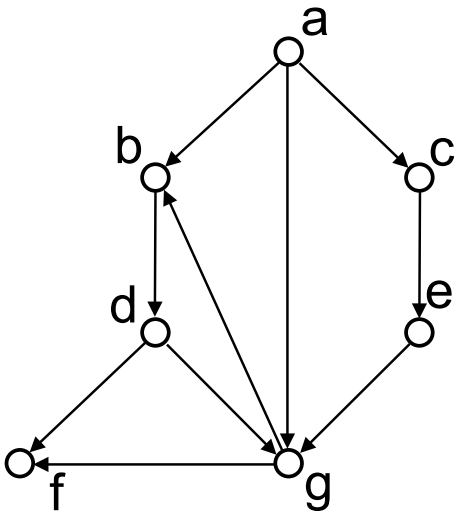

Abbildung F.2: Beispiel für einen gerichteten Graphen

Algorithmus F.1 Depth_First_Search(G: graph, v: vertex of G)

```
1: mark v;
2: for all edges (v,w) do
3:    if w is unmarked then
4:       Depth_First_Search(G,w);
5:    end if
6: end for
```

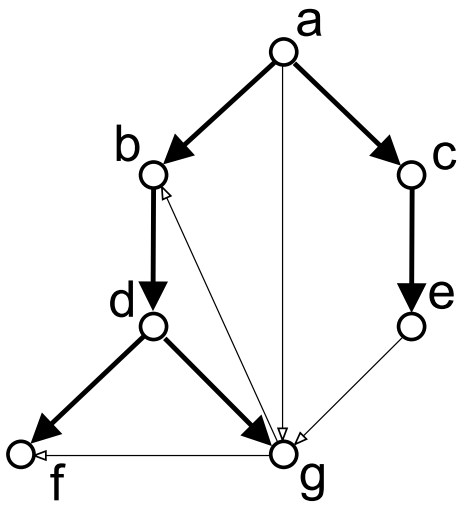

Abbildung F.3: Tiefensuchbaum für einen gerichteten Graphen

F.4 Breitensuche

Bei der Breitensuche werden ausgehend von einem Knoten v zunächt alle Knoten besucht, die über Kanten erreicht werden können. Im nächsten Schritt wird diese Operation für alle im vorhergehenden Schritt besuchten Knoten wiederholt. Algorithmus F.2 aus [45] führt eine Breitensuche in einem Graphen G vom Knoten v ausgehend aus. Den so entstehenden Breitensuchbaum zeigt Abbildung F.4.

Algorithmus F.2 Breadth_First_Search(G: graph, v: vertex of G)

```
 1: mark v;
 2: put v in a queue; /*First In First Out*/
 3: while the queue is not empty do
 4:    remove the first vertex w from the queue;
 5:    for all edges (w,x) such that x is unmarked do
 6:       mark x;
 7:       add (w,x) to the tree T;
 8:       put x in the queue;
 9:    end for
10: end while
```

F.5 Transitive Hülle

Für einen gerichteten Graphen

$$G = (E, K)$$

ist die transitive Hülle

$$H = (E, L)$$

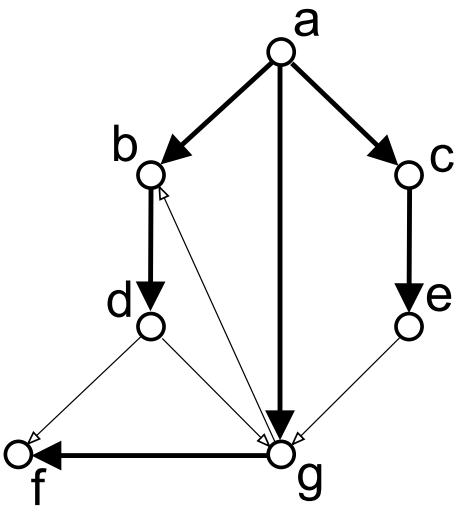

Abbildung F.4: Breitensuchbaum für einen gerichteten Graphen

von G ein gerichteter Graph mit einer Kante (v,w) dann und nur dann, wenn es einen gerichteten Pfad von v nach w in G gibt.

Literaturverzeichnis

[1] Albert, J.; Ottmann, Th.: Automaten, Sprachen und Maschinen für Anwender. BI Wissenschaftsverlag; 1990.

[2] Alder, G.: JGraph Swing Component.
http://www.jgraph.com

[3] Amor, R.; Faraj, I: Misconceptions about Integrated Product Databases. Electronic Journal of Information Technology in Construction; Vol. 6; 2001.
http://itcon.org/2001/5/

[4] Bittrich, D.: Industry Foundation Classes: Ein Überblick. Lehrstuhl Informatik im Bauwesen; Bauhaus-Universität Weimar; März 1998.
http://www.uni-weimar.de/ bauinf/lehre/CAEPlan/scripte/IFC_Ueberblick.pdf

[5] Allgemeine bauaufsichtliche Zulassung Z-8.1-16.2: Gerüstsystem "'Layher-Blitzgerüst 70 S"'; Deutsches Institut für Bautechnik; Berlin, Ausgabe 2002.

[6] Bretschneider, D.: Modellierung rechnerunterstützter, kooperativer Arbeit in der Tragwerksplanung. Fortschritt-Berichte VDI Reihe 4 Nr. 151; VDI-Verlag; 1998.

[7] Bronstein-Semendjajew: Taschenbuch der Mathematik. Verlag Harri Deutsch; Thun; 1989.

[8] Brown, A.; Rezgui, Y.; Cooper G.; Yip, J.; Brandon, P.: Promoting Computer Integrated Construction Through the Use of Distribution Technology. Electronic Journal of Information Technology in Construction; Vol. 1; 1996.
http://itcon.org/1996/3/

[9] Chen, P. P.-S.: The Entity-Relationship Model – Toward a Unified View of Data. ACM Transactions on Database Systems; Vol. 1; No. 1; S. 9-36; 1976.

[10] Coad, P.; Yourdon, E.: Objektorientierte Analyse. Prentice Hall Verlag; 1994.

[11] Coad, P.; Yourdon, E.: Objektorientiertes Design. Prentice Hall Verlag; 1994.

[12] Crnkovic, I.; Dahlkvist, A.P.; Svensson, D.: Managing Complex Systems – Challenges for PDM and SCM. Mälardalen University Sweden; 2001.
http://www.mrtc.mdh.se/publications/0329.pdf

[13] Dahlkvist, A.P.; Asklund, U.; et al.: Product Data Management and Software Configuration Management – Similarities and Differences. The Association of Swedish Engineering Industries; 2001;
http://www.mrtc.mdh.se/publications/0373.pdf

[14] Dart, S.: Spectrum of Functionality in Configuration Management Systems; Technical Report. Software Engineering Institute; Carnegie Mellon University; Pittsburgh, Pennsylvania; 1990.

[15] Dart, S.: Concepts in Configuration Management Systems. Software Engineering Institute; Carnegie Mellon University; Pittsburgh, Pennsylvania; 1991.
http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html

[16] DIN 4420, Teil 1: Arbeits- und Schutzgerüste – Allgemeine Regelungen, Sicherheitstechnische Anforderungen, Prüfungen. Beuth Verlag GmbH; Berlin; Ausgabe Dezember 1990.

[17] Deutscher Stahlbau-Verband: Standardbeschreibung Produktschnittstelle Stahlbau – Teil 1: Empfehlungen für den Anwender; DSTV-Arbeitsausschuß EDV; Schnittstellenversion: April 2000.

[18] Verordnung über ernergiesparenden Wärmeschutz und energiesparende Anlagentechnik bei Gebäuden (Energieeinsparverordnung - EnEV). Verkündet im Bundesgesetzblatt (BGBl) Teil I Nr. 59 vom 21. November 2001, S. 3085 ff.

[19] Estublier, J.; Favre, J.-M.; Morat, P.: Toward SCM/PDM Integration? System Configuration Management; ECOOP'98 SCM-8 Symposium; Proceedings; July 1998; S. 75-94.

[20] Estublier, J.: Software Configuration Management: A Roadmap. Dassault Systemes / LSR; Grenoble University; 2000.

[21] Firmenich, B.: CAD im Bauplanungsprozess: Verteilte Bearbeitung einer strukturierten Menge von Objektversionen. Dissertation; Bauhaus-Universität Weimar; 2001.

[22] Froese, T.; Fischer, M.; et al.: Industry Foundation Classes for Project Management – A Trial Implementation. Electronic Journal of Information Technology in Construction; Vol. 4; 1999.
http://itcon.org/1999/2/

[23] Funk, N.: JEP – Java Math Expression Parser.
http://jep.sourceforge.net/index.html

[24] Gamma, E.; Helm, R.; Johnson, R; Vlissides, J.: Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software. 1. Auflage 1996; 5., korrigierter Nachdruck; Addison Wesley Verlag; 1996.

[25] Goos, G.: Vorlesungen über Informatik, Band 1: Grundlagen und funktionales Programmieren. 3. überarbeitete Auflage; Springer-Verlag; 2000.

[26] Goos, G.: Vorlesungen über Informatik, Band 2: Objektorientiertes Programmieren und Algorithmen. 2. Auflage; Springer-Verlag; 1999.

[27] Gupta, A.; Tiwari, S.: Constraint Management On Distributed Design Databases. Bulletin of the Technical Committee on Data Engineering; IEEE Computer Society; Vol. 17 No. 2; June 1994; S. 47-51.
ftp://ftp.research.microsoft.com/pub/debull/june94-a4final.ps

[28] Haller, H.-W.: Ein Produktmodell für den Stahlbau. Berichte der Versuchsanstalt für Stahl, Holz und Steine der Universität Fridericiana in Karlsruhe; 1994.

[29] Hedin, G.; Ohlsson, L.; McKenna, J.: Product Configuration Using Object Oriented Grammars. System Configuration Management; ECOOP'98 SCM-8 Symposium; Proceedings; July 1998; S. 107-126.

[30] Honorarordnung für Architekten und Ingenieure. Fassung der Bekanntmachung vom 4. März 1991 (BGBl. I S. 533); Deutscher Taschenbuch Verlag; 1992.

[31] Hörenbaum, C.: Model Extensions of IFC2x, Volume II – Steel Construction Domain; Lehrstuhl für Stahl- und Leichtmetallbau, Universität Karlsruhe (TH); August 2001.

[32] Howard, H.C.; Keller, A.M.; et al.: Versions, Configurations, and Constraints in CEDB.
http://citeseer.nj.nec.com/366689.html

[33] Horstmann, C. S.; Cornell, G.: Core JAVA 2. Band 1 – Grundlagen. Prentice Hall; 1999.

[34] Horstmann, C. S.; Cornell, G.: Core JAVA 2. Band 2 – Expertenwissen. Markt und Technik Verlag; 2000.

[35] Initial Graphics Exchange Specification (IGES) - A Digital Representation for Communication of Product Definition Data. ANSI Standard Y 14.26 M, American National Standards Institute (ANSI); Ausgabe 1981.

[36] ISO 10303: Product Data Representation and Exchange. National Institute of Standards and Technology; Gaithersburg; USA; 1992.

[37] Kopp, H.: Compilerbau. Grundlagen, Methoden, Werkzeuge. Carl Hanser Verlag; 1988.

[38] Kretz, J.: Zukünftiges Projektmanagement im Bauwesen. 2000.
http://www.kretz.de/kretz/2/i9c63f36-001.APD/projektmanagement.pdf

[39] Lin, Yi-Jing: Configuration Management in Terms of Logical Structures. Ph.D. Dissertation; Brown University; Providence, Rhode Island; 1995.
http://www.cs.brown.edu/publications/techreports/reports/CS-95-31.html

[40] Lin, Yi-Jing; Reiss, S.P.: Configuration Management in terms of Modules. Proceedings of the 5th International Workshop on Software Configuration Management; S. 17-26; April 1995.
http://www.cs.brown.edu/publications/techreports/reports/CS-94-45.html

[41] Lin, Yi-Jing; Reiss, S.P.: Configuration Management with Logical Structures. July 1995.
http://www.cs.brown.edu/publications/techreports/reports/CS-95-23.html

[42] Lu, S. C.-Y.; Cai, J.: Modelling Collaborative Design Process with a Socio-Technical Framework. The IMPACT Laboratory; University of Southern California, Los Angeles; 1999.
http://impact.usc.edu/cerl/publications/CE99Paper.pdf

[43] Lu, S. C.-Y.; Cai, J.; Burkett, W.; Udwadia, F.: A Methodology for Collaborative Design Process and Conflict Analysis. The IMPACT Laboratory; School of Engineering; University of Southern California, Los Angeles; 2000.
http://impact.usc.edu/cerl/publications/CIRP2000prin.pdf

[44] Magnusson, B.: Vorwort, System Configuration Management; ECOOP'98 SCM-8 Symposium; Proceedings; July 1998.

[45] Manber, U.: Introduction To Algorithms. A Creative Approach. Addison-Wesley Publishing Company Inc.; 1989.

[46] Mayer, V.: WWW-basierte Groupware: Ein Überblick. Studienarbeit am Fachbereich Informatik der Universität Hamburg; 1999.

[47] Müller, Chr.: Der Virtuelle Projektraum – Organisatorisches Rapid-Prototyping in einer internetbasierten Telekooperationsplattform für Virtuelle Unternehmen im Bauwesen. Dissertation; Karlsruhe; 1999.

[48] N.N.: Common Object Request Broker Architekture /IIOP Specification; 2002.
http://www.omg.org/technology/documents/formal/corba_iiop.htm

[49] N.N.: Industry Foundation Classes (IFC): Management Übersicht. Industrie Allianz für Interoperabilität, e.V. (IAI); Dezember 1999.

[50] Oestereich, B.: Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language – 5., völlig überarb. Aufl. – München; Wien: Oldenbourg; 2001.

[51] Sauer, H.: Relationale Datenbanken: Theorie und Praxis. 3. korrigierte Auflage; Addison Wesley; 1994.

[52] Scheele, M.-O.: Textverarbeitung mit Java - Reguläre Ausdrücke in Java 1.4. Java Magazin; Software & Support Verlag GmbH; Ausgabe 10.02.

[53] Schöttner, J.: Produktdatenmanagement in der Fertigungsindustrie. Carl Hanser Verlag; 1999.

[54] Stein, D.: Definition und Klassifikation der Begriffswelt um CSCW, Workgroup Computing, Groupware, Workflow Management. Seminararbeit; Universität Gesamthochschule Essen; Allgemeine Wirtschaftsinformatik; 1996. http://www-stud.uni-essen.de/ sw0136/AWi_Seminar.html

[55] OMG-Unified Modeling Language. http://www.omg.org/uml/

[56] Unified Modeling Language (UML)1.4.; Notationsübersicht; 2002. http://www.oose.de/uml

[57] Wahl, G.: UML kompakt. http://www.sigs-datacom.de/sd/publications/os/1998/02/ OBJEKTspektrum_UM_kompakt.htm

[58] Weiß, J.: Der Layher Gerüstplaner – EDV-Programm zur Planung von Fassadeneinrüstungen. Wilhelm Layher GmbH & Co. KG; 2001.

[59] Westfechtel, B.; Conradi, R.: Software Configuration Management and Engineering Data Management: Differences and Similarities. System Configuration Management; ECOOP'98 SCM-8 Symposium; Proceedings; July 1998; S. 95-106.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis