in: Informatik - Forschung und Entwicklung 13(1), Springer, 1998, pp. 1-17

Konfigurierungskonzepte für datenbank-basierte, technische Entwurfsanwendungen

Wolfgang Käfer[1], **Norbert Ritter**[2], **Harald Schöning**[3]

[1] Debis T&M MOB, Karlstr. 31, 89073 Ulm, Tel.: (0731) 96894-17, e-mail: wkaefer@debinet.de

[2] Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, 67653 Kaiserslautern
Tel.: (0631) 205-3264, e-mail: ritter@informatik.uni-kl.de

[3] Software-AG, Uhlandstr. 12, 64297 Darmstadt, Tel.: (06151) 92-1085, e-mail: hsg@software-ag.de

Zusammenfassung. Schwerpunkte einer Datenbank-Unterstützung von technischen Entwurfsanwendungen sind eine geeignete Ablaufkontrolle sowie ein geeignetes Datenmodell. Letzteres muß Versionen komplexer Objekte verwalten können. Um aus der Gesamtmenge der in einer Datenbank enthaltenen Versionen konsistente Einheiten (Mengen 'kompatibler' Versionen) hervorheben zu können, werden geeignete Konfigurierungskonzepte benötigt. Dieses Papier gibt eine Klassifikation verschiedener DB-bezogener Konfigurierungsansätze, bewertet diese und begründet damit die Wahl, die im Rahmen des Objekt- und Versionsdatenmodells OVM hinsichtlich des zu unterstützenden Konfigurierungskonzeptes getroffen wurde. Wir werden argumentieren, daß insbesondere die Flexibilität, die das gewählte Konzept sowohl hinsichtlich der Definition von Konfigurationstypen als auch hinsichtlich des Anlegens von Konfigurationen bietet, zu einer besseren Unterstützung von technischen Entwurfsanwendungen als in bestehenden objekt-orientierten Systemen führt.

Schlüsselwörter: Versionierung/Konfigurierung, Entwurfsumgebungen, Entwurfsdatenbanken.

Abstract. Data model and process control are key issues of database support for technical design applications. The data model has to provide adequate versioning and configuration services. The latter are of major importance for establishing consistent units of versions (sets of 'compatible' versions). This paper classifies and assesses db-related configuration concepts. This assessment proves that the approach taken by our model OVM (Object and Version Model) is appropriate to flexibly support technical design applications. We will argue that especially the flexibility of the OVM configuration concepts (w. r. t both, definition of configuration types as well as handling of configurations) leads to a better support as the one provided by current object-oriented systems.

Keywords: Versioning and configuration concepts, technical design applications, design databases.

CR Subject Classification: H.1.2, H.2.1, H.2.3, H.2.8, J.6, J.7.

---

1. Einleitung

1.1 Motivation

Entwerfen ist von jeher eine iterative Tätigkeit, bei der aus einer vagen Idee die Pläne für ein Produkt entstehen - sei es ein Gebäude, eine Maschine, ein Software-Paket oder ein VLSI-Chip. In vielen Entwurfsbereichen werden die zu entwerfenden Produkte zunehmend komplexer. Ein Paradebeispiel ist der VLSI-Entwurf. Dies hat zur Konsequenz, daß organisatorische Maßnahmen

ergriffen werden müssen, um zu ermöglichen, daß ein Entwurf der geforderten Qualität in annehmbarer Zeit fertiggestellt werden kann. Solche Maßnahmen umfassen unter anderem

- den Einsatz von Entwurfswerkzeugen, die die Arbeit des Ingenieurs erleichtern oder automatisierbare Tätigkeiten übernehmen (z. B. graphische Editoren, Design-Rule-Checker),
- die Vorgabe einer Entwurfsmethode und
- die Aufteilung der Entwurfsarbeit auf mehrere kooperierende Entwickler.

Diese Maßnahmen sind jedoch nur dann wirkungsvoll, wenn die Entwickler während ihrer Arbeit eine entsprechende Unterstützung erhalten. So muß der Entwickler stets einen Überblick über den aktuellen Stand und die Geschichte des Entwurfsprozesses haben. Er muß wissen, welche Werkzeuge im momentanen Zustand seines Entwurfs angewendet werden können oder müssen [31]. Der Ingenieur muß in der Lage sein, zu einem früheren (Zwischen-)Ergebnis seines Entwurfs zurückzukehren. Ferner muß der Austausch von (evtl. vorläufigen) Entwurfsdaten zwischen Entwerfern koordiniert werden. Dies alles sind Aufgaben von Entwurfsumgebungen, die im Idealfall die Entwickler vom Beginn ihrer Arbeit an führen und unterstützen. Offensichtlich ist ein Kernstück von Entwurfsumgebungen die Verwaltung von (Zwischen-)Ergebnissen des Entwurfs (Versionen) und damit verbunden Hilfe beim Zusammenfügen von Einzelergebnissen zu einem Gesamtergebnis (Konfigurierung).

1.2 Beitrag dieses Artikels

In diesem Artikel soll vor allem der Konfigurierungsaspekt vertieft werden. Zur Vorbereitung dieser Diskussionen werden wir zunächst in Kapitel 2 grundlegende Anforderungen von technischen Entwurfsanwendungen an ein Datenmodell anführen und diese an einem Anwendungsbeispiel aus dem Bereich des VLSI-Entwurfs weiter erläutern. Kapitel 3 gibt einen Überblick über Konfigurierungskonzepte, klassifiziert verschiedenartige Konfigurationen und diskutiert deren Anwendbarkeit in Entwurfsumgebungen. In Kapitel 4 wird kurz erläutert, welche Konfigurierungskonzepte in anderen (vorwiegend DB-basierten) Ansätzen gewählt worden sind und inwieweit diese für technische Entwurfsanwendungen geeignet erscheinen. Aufbauend auf der in Kapitel 3 gegebenen Klassifikation und der Diskussion verwandter Ansätze in Kapitel 4 beschreibt Kapitel 5 den in unserem Objekt- und Versionsdatenmodell (OVM) gewählten Konfigurierungsansatz und zeigt seine Vorteile hinsichtlich einer Nutzung in technischen Entwurfsanwendungen auf. Kapitel 6 beschließt den Artikel mit einer Zusammenfassung.

2. Anforderungen und Anwendungsbeispiel

In diesem Abschnitt wollen wir zunächst die grundlegenden Anforderungen, die technische Entwurfsanwendungen an ein Datenmodell stellen, anführen und diese anschließend anhand eines konkreten Beispiels aus dem Bereich des VLSI-Entwurfs weiter verdeutlichen.

2.1 Anforderungen

An wesentlichen Anforderungen sind zu nennen:

- Komplexe Objekte
  Ein Entwurfsobjekt ist in der Regel als Komposition von verschiedenen Elementarobjekten zu sehen; das Entwurfsobjekt ist also ein komplexes Objekt. Daher muß ein adäquates Objekt- und Versionsmodell komplexe Objekte handhaben können.
- Explizite Handhabung der Objektzustände
  In Ingenieuranwendungen möchte man *bestimmte Objektzustände explizit benennen* und sich *explizit auf diese Zustände beziehen* können. Entsprechend sollen nur diese benannten Objektzustände aufgezeichnet werden (und nicht, wie z. B. in einem temporalen Datenmodell, alle jemals aufgetretenen Zustände).
- Alternative Objektzustände
  Typischerweise werden in Ingenieuranwendungen technische Objekte sowohl in aufein-

anderfolgenden Verarbeitungsschritten als auch innerhalb eines einzigen Verarbeitungsschrittes iterativ erstellt bzw. konkretisiert. Hierbei können neue (ggf. ‘verbesserte’) Objektzustände nicht nur vom ‘neuesten’ Objektzustand, sondern auch von älteren Objektzuständen abgeleitet werden, so daß für jedes Objekt *Bäume* (eventuell auch Graphen) von Objektzuständen (die sog. *Abstammungsgraphen*) entstehen (vgl. Abb. 1).

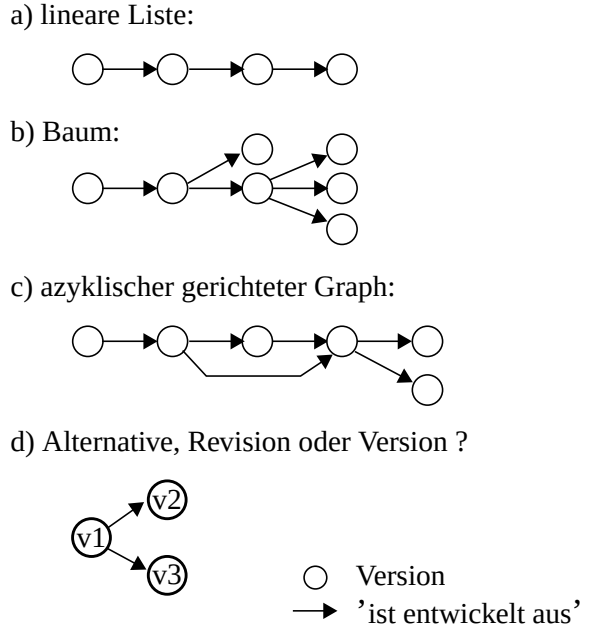


Abb. 1: Abstammungsgraphen

Im folgenden werden wir diese Objektzustände als *Versionen* bezeichnen. Üblicherweise unterscheidet man zwischen Revisionen (Verbesserungen ausgehend von im Abstammungsgraphen vorhergehenden Versionen) und Alternativen (Versionen, die einen anderen Ansatz der Realisierung darstellen, also z. B. eine andere Technologie). Alternativen müssen keinen Vorgänger im Abstammungsgraphen besitzen. Es ist jedoch auch möglich, eine Alternative aus einer oder mehreren Versionen abzuleiten. Beispielsweise kann man eine Version v1 zum einen bzgl. der Laufzeit optimieren (Ergebnis: Version v2), zum anderen bzgl. des Platzbedarfes (Ergebnis: Version v3), wodurch zwei Alternativen entstehen (v2 und v3, vgl. Abb. 1d). Gerade das letzte Beispiel zeigt, daß die Begriffe Alternative und Revision sich jeweils auf weitere Versionen beziehen und ohne diesen Bezug nicht orthogonal sind: v2 ist eine Alternative zu v3, aber eine Revision von v1. Aus diesem Grunde verzichten wir auf die beiden Begriffe und sprechen nur von Versionen.

- Abstraktion
  Es ist wichtig, Objekte abstrakt betrachten zu können. Beispielsweise entscheidet man bei der Konfigurierung anhand der Entwurfsqualität der vorliegenden Versionen und nicht anhand der konkreten Details. Diese Abstraktion sollte vom Modell unterstützt werden.
- Hierarchisierung
  In nahezu allen Entwurfsanwendungen werden komplexe Entwurfsaufgaben in einfachere Teilaufgaben zerlegt. Hiermit ist im allgemeinen auch eine Dekomposition der Entwurfsdaten verbunden.
- Konfigurierung
  Die Zerlegung des Entwurfsobjektes muß zur Herstellung eines endgültigen Ergebnisses wieder rückgängig gemacht werden. Da für jede Komponente im allgemeinen mehrere Realisierungen gewählt werden können, kann die Zusammensetzung nicht automatisch geschehen, sondern stellt wiederum Entwurfsarbeit dar. Dabei muß das Ergebnis bezüglich der durch das Anwendungsgebiet vorgegebenen Kriterien konsistent sein.

2.2 Anwendungsbeispiel

In diesem Abschnitt wollen wir nun die oben angesprochenen Anforderungen anhand eines Anwendungsbeispiels aus dem Bereich des VLSI-Entwurfs [46] näher erläutern. Dabei soll jedoch bereits der Schwerpunkt darauf gelegt werden, die Notwendigkeit der Unterstützung eines geeigneten Konfigurierungsbegriffs herauszuarbeiten.

Das angesprochene Beispiel ist in Abb. 2 in Form eines Makrodatenschemas (in der grafischen Notation des E/R-Modells) illustriert. Wir bezeichnen den Teil des Gesamtschemas, der sich ausschließlich mit der Komplexobjektbildung beschäftigt und somit von den sich ‘hinter’ den Komplexobjekten verbergenden Elementarobjekt-

netzen abstrahiert, als *Makrodatenschema*. Jeder Objekttyp des Makrodatenschemas abstrahiert von einer Reihe von für die Entwurfswerkzeugläufe relevanten Mikrodaten, wie z. B. Netzlisten und Verdrahtungsinformationen, etc. Somit ergibt sich eine Gesamtsicht erst unter weiterer Berücksichtigung des Mikrodatenschemas, das elementare Objekttypen und zwischen diesen auftretenden (elementare) Beziehungstypen erfaßt. Zur Motivation wollen wir jedoch zunächst beim Makrodatenschema bleiben.

Ein VLSI-Chip kann durch die in Abb. 2 dargestellten Komplexobjekttypen modelliert werden. Das *Interface* eines Chips beschreibt die Außenanschlüsse und den Rahmen. Eine Trennung von genereller Beschreibung und aktueller Verwendungsbeschreibung ist sinnvoll, wie im folgenden noch deutlich werden wird. Dies führt dazu, daß neben dem Interface (generelle Beschreibung) als weiterer Komplexobjekttyp *Instance* (aktuelle Verwendungsbeschreibung) betrachtet wird und diese beiden Komplexobjekttypen durch den Beziehungstyp *Instantiation* verbunden werden. Während eine Instanz (Instance) genau einer Schnittstelle (Interface) zugeordnet ist, können für

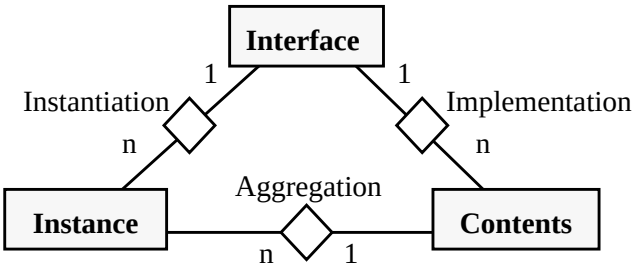
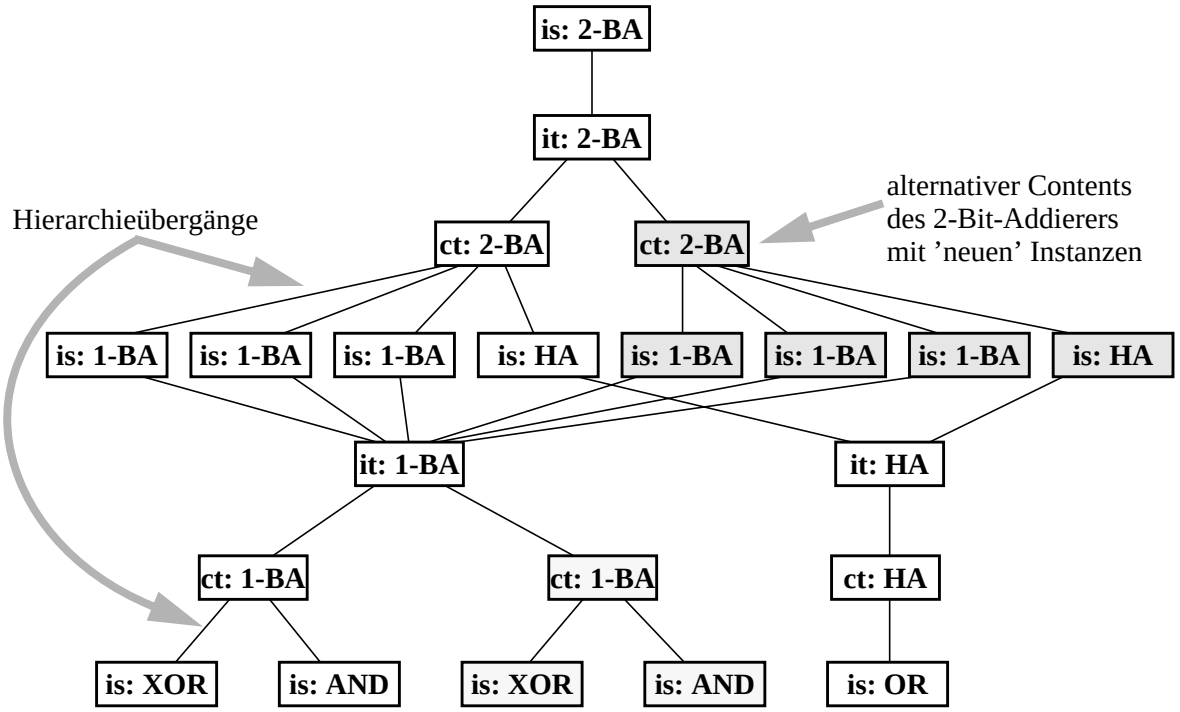


Abb. 2: Makrodatenschema als E/R-Diagramm

letztere mehrere alternative Implementierungen existieren. Die Kenntnis der generellen Beschreibung (Interface) ist hinreichend für die Implementierung (Beziehungstyp *Implementation*) des Chips, die sich im Komplexobjekttyp *Contents* niederschlägt.

Mit dem Beziehungstyp *Aggregation* schließt sich nun das in Abb. 2 dargestellte Dreieck. Dies wiederum ist auf den hierarchischen Aufbau eines Chips zurückzuführen. Die dargestellten Informationsstrukturen beschreiben demnach nicht nur den Chip als Ganzes, sondern jeden Knoten in der den Chip manifestierenden Modul- bzw. Zell-Hierarchie. Entsprechend beinhaltet das den inneren Aufbau einer Zelle beschreibende Contents-



1-BA : 1-Bit-Addierer HA : Halbaddierer it: Interface ct: Contents
2-BA : 2-Bit-Addierer is: Instance

Abb. 3: Beispielhafte Makrodaten

Objekt die konkreten Beschreibungen der Schnittstellen (Instance) der direkten Submodule.

Abb. 3 zeigt eine beispielhafte Ausprägungsstruktur der angesprochenen Makrodatentypen (Objektebene). Dargestellt ist die einem 2-Bit-Addierer 'untergeordnete' Struktur. Der Addierer liegt in zwei Implementierungen vor. Jede dieser Implementierungen integriert drei 1-Bit-Addierer sowie einen Halbaddierer, die wiederum entsprechenden Schnittstellen und zugehörigen Implementierungen zugeordnet sind. Die Hierarchieübergänge innerhalb der Modulhierarchie sind somit durch die Aggregationsbeziehung zwischen den Typen Contents und Instance gegeben.

Eine die Diskussionen dieses Beitrags vereinfachende Annahme besteht darin, daß alle angesprochenen Makrodatentypen (vgl. Abb. 2) versioniert sein sollen und daß die angegebenen Beschreibungen der Beziehungstypen sowohl für die Objekt- als auch für die Versionsebene gelten. Dies bedeutet zunächst, daß die Komplexobjekte, die, wie bereits gesagt, Elementarobjektnetze kapseln, in verschiedenen, jeweils für den Entwurfsprozeß bedeutsamen Zuständen auftreten können. Diese Zustände entsprechen Versionen, so daß ein Objekt als Menge seiner Versionen betrachtet werden kann. Eine zwischen Objekten bestehende (Objekt-)Beziehung ist nun sinnvollerweise auf der Versionsebene zu konkretisieren/verfeinern. Steht beispielsweise ein Contents-Objekt mit einem Instance-Objekt in einer (Aggregations-)Beziehung, so bedeutet dies nicht, daß jede Versionen des Instance-Objektes mit jeder Version des Contents-Objektes (hinsichtlich der Semantik der Aggregationsbeziehung) 'kompatibel' ist. Vielmehr müssen konkrete Aggregationsbeziehungen zwischen Contents- und Instance-Versionen angelegt werden, die die 'Kompatibilität' dieser Versionen beschreiben und die zugehörige Objektbeziehung konkretisieren.

Durch die in diesem Abschnitt gewählte Beschreibung wurden die meisten der im vorangegangenen Abschnitt angeführten Anforderungen (komplexe Objekte, Hierarchisierung, explizite Handhabung der Objektzustände, alternative Objektzustände) direkt verdeutlicht. Der Anforderung der Abstraktion wird durch die Erfassung des Makrodatenschemas Rechnung getragen, das (ausschließlich) die entwurfsprozeß-relevanten Datenelemente beschreibt. Jeder Objekttyp des Makrodatenschemas abstrahiert von für Entwurfswerkzeuge relevanten Mikrodaten.

Lediglich auf den Punkt der Konfigurierung ist hier noch einzugehen. Wir verstehen unter einer *Konfiguration* die Zusammenfassung einer (nach Anwendungsgesichtspunkten) 'konsistenten' Teilmenge der in der Datenbank gespeicherten Versionen. Damit ist eine Konfiguration gegeben durch eine spezielle, strukturelle Teile-Ganzes-Beziehung, wobei das Konfigurationsobjekt als Ganzes aggregierte Information tragen kann, die die Gesamtmenge der Versionen beschreibt (Abstraktion). Sie dient damit im wesentlichen der Identifikation 'konsistenter' Versionsstrukturen, so daß die wichtigsten Operationen in der Erzeugung (Konfigurieren) und dem Selektieren von Konfigurationen bestehen.

Den angesprochenen Konsistenzbegriff wollen wir nun etwas näher betrachten, da er grundlegend für den gesamten Konfigurationsbegriff ist. Es ist intuitiv klar, daß ein Konfigurationstyp einen Schemaausschnitt betrifft. Beispielsweise erwartet das Entwurfswerkzeug Chip-Planner eine Konfiguration, die einer Zelle ihre direkten Subzellen zuordnet. In unserem Beispielschema aus Abb. 2 betrifft dies also den folgenden Schemaausschnitt: Von der Interface-Version (der betrachteten Zelle) über die Beziehung vom Typ Implementation zu einer Contents-Version und von dort aus über Aggregationsbeziehungen zu den Instance-Versionen der Subzellen. Dies bedeutet, daß die Beziehungstypen die Grundlage einer jeden Konfigurierung bilden und somit nur entsprechend kohärente Schemaausschnitte sinnvollerweise zu Konfigurationstypen herangezogen werden sollten. Betrachten wir nun beispielsweise den Beziehungstyp Implementation, so muß man beachten, daß nicht nur einem Interface-Objekt mehrere alternative Contents-Objekte zugeordnet werden können, sondern, daß jedes dieser Objekte nun auch in mehre-

ren Versionen vorliegen kann, von denen jeweils mehrere mit der betrachteten Zelle (Interface-Version) in Beziehung stehen. Innerhalb einer Konfiguration macht es jedoch nur Sinn, genau eine Contents-Version auszuwählen, um mit dieser Konstellation weiterzuarbeiten. Wir sehen, obwohl es sinnvoll ist, den Beziehungstyp Implementation als (1:n)-Beziehungstyp zu deklarieren, sollte in diesem Fall bei der Konfigurierung genau eine Version ausgewählt werden. Dies liegt jedoch an der Semantik dieses Beziehungstyps und kann auf keinen Fall verallgemeinert werden, wie der Beziehungstyp Aggregation verdeutlicht. Hier würden einige in der Literatur vorgeschlagene Versionsmodelle (siehe Kapitel 4) erzwingen, daß pro Instance-Objekt genau eine Version in die Konfiguration aufgenommen werden muß und arbeiten dazu beispielsweise mit impliziten Mechanismen wie generische Referenzen [15]. Man stelle sich jedoch den Fall vor, daß man innerhalb einer Zelle zwei Halbaddierer(-Versionen) benötigt und aus bestimmten Gründen eine platzoptimierte und eine laufzeitoptimierte Version des gleichen Objektes aufnehmen will. Wir sehen, die Konfigurierung ist äußerst abhängig von der anwendungsspezifischen Semantik der Beziehungstypen und sollte daher durch sehr flexible Mechanismen unterstützt werden. Neben diesen strukturellen Gesichtspunkten, muß es natürlich auch möglich sein, semantische Integritätsbedingungen an einen Versionsverbund, der eine Konfiguration darstellen soll, zu stellen. Beispielsweise sollte das DBMS überprüfen können, daß die Summe der Flächen der Subzellen die in der Contents-Version ausgewiesene Gesamtfläche der betrachteten Zelle nicht übersteigt.

Nachdem wir nun eine grobe, informale Einführung in die Problematik der Konfigurierung gegeben haben, wollen wir im sich anschließenden Kapitel eine Klassifikation von Konfigurationen geben.

3. Klassifikation von Konfigurationen

Im folgenden werden wir zunächst den Vorgang des Konfigurierens systematisch untersuchen. Die verschiedenen Vorgehensweisen beim Konfigurieren führen zu unterschiedlich gearteten Konfigurationen, die wir anhand einer Klassifikation diskutieren. Da jedoch die im vorangegangenen Kapitel gegebene Motivation des Konfigurationsbegriffes bereits über das hinausgeht, was in einigen, in der Literatur vorgeschlagenen Ansätzen angeboten, wollen wir den folgenden Diskussion eine allgemeine Definition der Begriffe *Version* und *Konfiguration* voranstellen.

Eine Version ist ein semantisch bedeutungsvoller Zustand in der Änderungsgeschichte eines evtl. komplex strukturierten (Entwurfs-)Objektes.

Eine Konfiguration ist eine konsistente, vollständige Sicht auf ein durch evtl. mehrere, komplex strukturierte und versionierte (Entwurfs-)Objekte und deren Beziehungen untereinander gebildetes, übergeordnetes (Entwurfs-)Objekt.

Das Ergebnis der Diskussionen dieses Kapitels ist durch die in Abb. 4 gezeigte Klassifikation vorweggenommen; diese kann somit auch als eine Art 'Wegweiser' durch dieses Kapitel gesehen werden.

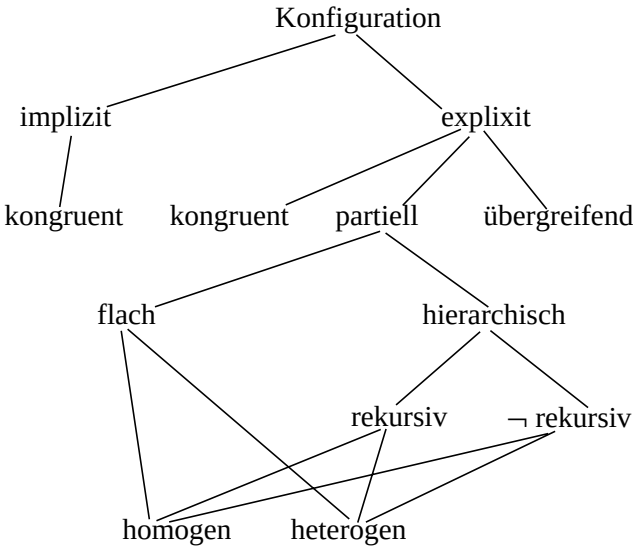


Abb. 4: Klassifikation von Konfigurationen

3.1 Implizite Konfigurationen

Ziel jeder Konfigurierung ist es, in einer Menge von (versionierten) Objekten eine Versionsauswahl so zu treffen, daß die ausgewählten Objektversionen eine konsistente Einheit bilden. In diesem Sinne ist etwa die Auswahl eines Autos mit

einer bestimmten Ausstattung (Motor, ABS, Farbe, Polster, …) ein Konfigurierungsvorgang (es sei hier angemerkt, daß üblicherweise mehrere tausend 'Ausstattungskonfigurationen' möglich sind - es ist deshalb unsinnig, alle Kombinationen quasi vorab zu bilden und dann eine auszuwählen; es handelt sich also tatsächlich um einen Konfigurierungsvorgang). Wie das Beispiel schon zeigt, kann die Konfigurierung eines Objekts sehr viel später erfolgen als die Objekt- bzw. Versionserzeugung. Dieser Aspekt wird von vielen objektorientierten Datenmodellen [6, 8] nicht ausreichend berücksichtigt. In manchen dieser Modelle wird direkt bei Transaktionsende eine Konfigurierung angestoßen, die von jedem übergeordneten Objekt eine neue Version erzeugt, die sich von der vorhergehenden nur dadurch unterscheidet, daß sie auf die neu erzeugte(n) Version(en) verweist. Dadurch entsteht ein Schneeballeffekt, da auch für in Beziehung stehende Objekte neue Versionen erzeugt werden müssen. Eine andere Möglichkeit besteht darin, diese (automatische) Versionserzeugung solange zu verzögern, bis eine Änderung des übergeordneten Objekts notwendig wird, bei der dann die Konfigurierung bzw. die Auswahl der neuen untergeordneten Objekte mit erfolgen muß. Beide Möglichkeiten verbinden Objekt- bzw. Versionserzeugung und Konfigurierung, weshalb wir solche Konfigurationen als ***implizite*** Konfigurationen bezeichnen.

Abb. 5 zeigt eine implizite Konfiguration beispielhaft auf. Die Objektstruktur 'o1-o2-o3' wird für jede Versionsentstehung wiederholt. Im Beispiel liegt o1 in einer Version vor, o2 in zwei Versionen und o3 in drei Versionen (jeweils gekennzeichnet durch die erste Ziffer nach dem Punkt in der Versionsnumerierung). Um nun den Objektaufbau für jede Version wiederholen zu können, müssen von Objekt o1 pro Version von o2 je drei (Anzahl der Versionen von o3) bis auf die Beziehung identische Versionen erzeugt werden (letzte Ziffer in der Versionsnumerierung). Entsprechend erhalten wir (2 x 3 =) sechs Versionen von o1, die sich jeweils nur durch ihre Beziehung zu unterschiedlichen Versionen von o2 unterscheiden. Dieses Beispiel verdeutlicht den angesprochenen Schneeballeffekt, der bei der impliziten Versionierung durch ein verzögertes Auswählen (Aufnehmen von Versionen in die Konfiguration) abgeschwächt werden kann.

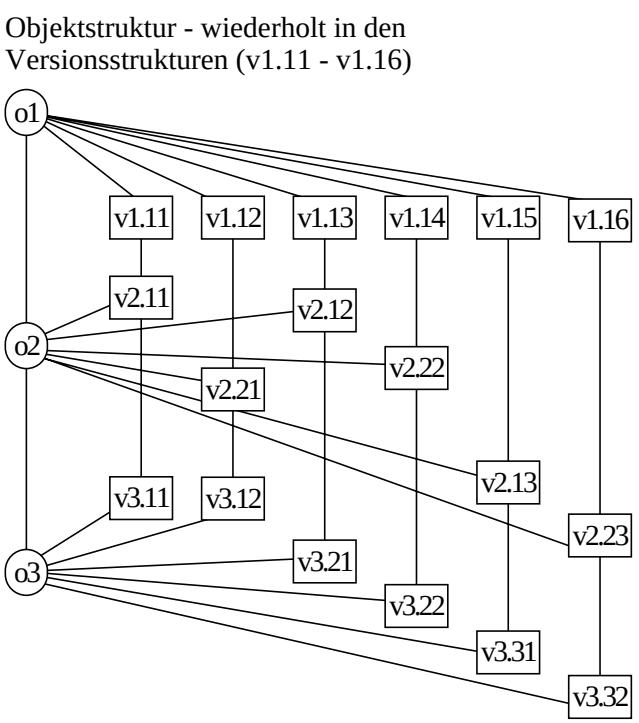


Abb. 5: Beispiel einer impliziten Konfiguration

3.2 Explizite Konfigurationen

Offenbar entspricht bei impliziten Konfigurationen die Konfigurationsstruktur der Objektstruktur; sie ist somit ***kongruent*** (vgl. Abb. 5). Dies bedeutet weiter, daß eine implizite Konfiguration immer genau eine Version pro Objekt betrachtet (siehe Begriff der *homogenen* Konfiguration, unten).

Im Falle ***expliziter*** Konfigurationen kann hingegen die Struktur der Konfiguration von der Objektstruktur getrennt betrachtet werden. Ziel der *expliziten* Konfigurierung ist es, eine konsistente Sicht auf eine komplexe Objektstruktur zu realisieren, deren (Teil-)Objekte versioniert sind - und zwar unabhängig von der Versionserzeugung. Dies bedeutet, daß bei der Konfigurierung ausgehend von einem Objekt für erreichbare versionierte Objekte Versionen (explizit) ausgewählt werden müssen. Im Gegensatz zur vorgestellten impliziten Konfigurierung trennen wir damit den Vorgang der Konfigurierung vollkommen von dem der Versionserzeugung. Dies bedeutet, daß

wir eine *explizite* Struktur zur Modellierung der Konfiguration zur Verfügung stellen müssen.

Diese Struktur kann natürlich trotzdem *kongruent* zur Objektstruktur sein, wie wir in einem der nachfolgenden Beispiele sehen werden. Daneben ist es denkbar, daß eine Konfiguration Versionen von nicht zusammenhängenden Objekten zusammenfaßt; wir bezeichnen solche Konfigurationen als ***übergreifende*** Konfigurationen[1]. In Entwurfsanwendungen bietet sich jedoch sicherlich eine engere Sichtweise auf Konfigurationen an, die auf ein bestimmtes zusammengesetztes Objekt ausgerichtet ist (wie bereits in Abschnitt 2.2. motiviert). Eine dritte Möglichkeit der Gestaltung der Konfigurationsstruktur ergibt sich daraus, daß es in großen Objektnetzen oft unerwünscht und auch unpraktikabel ist, das gesamte zusammenhängende Objektnetz zu konfigurieren. Dies führt zu ***partiellen*** Konfigurationen[2], die nur einen Teil einer gegebenen Objektstruktur konfigurieren, d. h., Teile werden nicht konfiguriert. Beispielsweise wird die Motorvariante eines Autos nicht konfiguriert. Es bleibt dem Käufer überlassen, diesen Teil der Konfigurierung vorzunehmen, d. h., eine bestimmte Motorversion aus der Menge der verfügbaren Motorversionen auszuwählen.

Abb. 6 zeigt eine explizite, kongruente Konfiguration auf. In diesem Beispiel wird aus Übersichtlichkeitsgründen auf die Darstellung von Beziehungen zwischen den Versionen der Objekte verzichtet. Dennoch sollte beachtet werden, daß wir die Versionsbeziehungen als den eigentlichen Indikator der Verträglichkeit von Versionen und damit als die Basis der Aufnahme von Versionen in eine Konfiguration betrachten. Entsprechend können wir den Konfigurationsbegriff hier verschärfen: die in einer Konfiguration ausgewählten Ob-

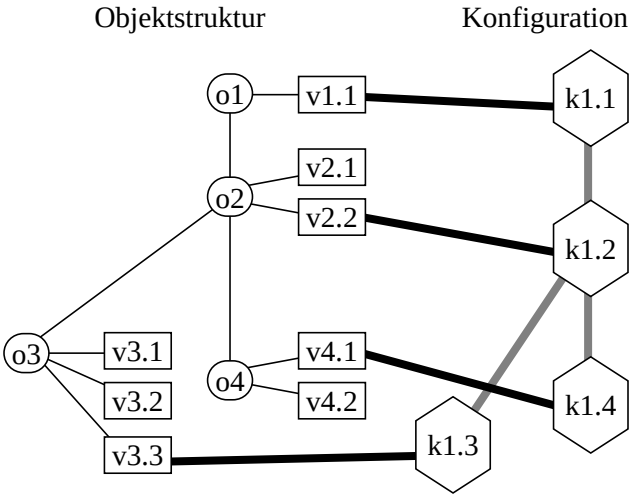


Abb. 6: Beispiel einer *expliziten, kongruenten* Konfiguration

jekt*versionen* müssen einen zusammenhängenden Graphen bilden. Wir halten diese Forderung für außerordentlich wichtig, da das Konzept der Versionsbeziehung wesentlich dazu beiträgt, auf natürliche Weise die ‘Kompatibilität’ von Versionen ausdrücken zu können. Dieser Aspekt von Versionsbeziehungen wurde bereits in Abschnitt 2.2 angesprochen und sei hier noch einmal betont, da die Abbildungen dieses Kapitels aus Übersichtlichkeitsgründen auf eine Darstellung von Versionsbeziehungen verzichten.

Es ist festzustellen, daß die Beziehungen zwischen den (Teil-)Konfigurationen die Beziehungen zwischen den Objekten widerspiegeln - die Konfigurationshierarchie ist kongruent zur Objekthierarchie[3]. Dadurch kann der Vorgang der Versionierung getrennt von dem Vorgang der Konfigurierung betrachtet werden; zusätzlich eröffnet die explizite Modellierung der Konfigurationshierarchie aber auch neue Freiheitsgrade. Bevor wir diesen Aspekt vertiefen, wollen wir jedoch noch kurz auf die Bedeutung der schon im OVM eingeführten Versionsbeziehungen eingehen.

1. Wir vertiefen diese Art der Konfigurierung nicht weiter, da der Zusammenhang zwischen den Objekten als Teil des Datenbankschemas beschrieben werden sollte und nicht durch den Konfigurierungsmechanismus.
2. Partielle Konfigurationen können auch vollständig sein; wir betrachten sie deshalb gleichwertig mit kongruenten und übergreifenden Konfigurationen.
3. Entsprechend könnte man auf die Beziehungen zwischen den Konfigurationen verzichten, da sie ja kongruent zur Objektstruktur ist (vgl. [35]). Dies erfordert jedoch einen erheblich höheren Interpretationsaufwand bei der Verarbeitung der Konfiguration.

3.2.1 Flache Konfigurationen

Ein offensichtlicher Nachteil der expliziten, kongruenten Konfigurationen ist, daß sie den Objektaufbau exakt wiederholen und somit keine Abstraktion erlauben. Diese ist jedoch in größeren Objekthierarchien zweifelsohne wünschenswert. Wir führen deshalb ***flache*** Konfigurationen ein. Eine flache Konfiguration besteht aus einer Liste von Verweisen zu Objektversionen (vgl. Abb. 7). Die Konfiguration selbst ist also 'flach' und somit vollständig unabhängig vom aktuellen Aufbau des konfigurierten Objekts selbst. In größeren Objekthierarchien kann dies sehr unübersichtlich werden, da diese ggf. sehr lange Liste von Versionsverweisen offensichtlich nur schwer erstellbar/wartbar ist. Das Erstellen der Konfiguration selbst wird somit zu einem iterativen Prozeß, der sogar selbst wieder durch Versionierung unterstützt werden kann. Man beginnt mit einer (unvollständigen) Konfiguration, die nur Teile des zu konfigurierenden Objekts adressiert, und vervollständigt diese, bis ein Konfigurationszustand erreicht ist, der die gesamte zu konfigurierende Objektstruktur erfaßt. Diesen Vorgang kann man unterstützen, in dem Konfigurationen im Sinne des 'Weiterkonfigurierens' änderbar gestaltet werden oder in dem man Konfigurationsversionen unterstützt. Im letzteren Falle bedeutet das Erstellen ähnlicher Konfigurationen im allgemeinen ein Kopieren einer Konfigurationsversion und deren Anpassung. Um jedoch ein hohes Aufkommen von Konfigurationsversionen zu verhindern, ist es wohl sinnvoller, das angesprochene 'vervollständigende Ändern' bestehender Konfiguration bzw. eine kombinierte Anwendung beider Verfahren

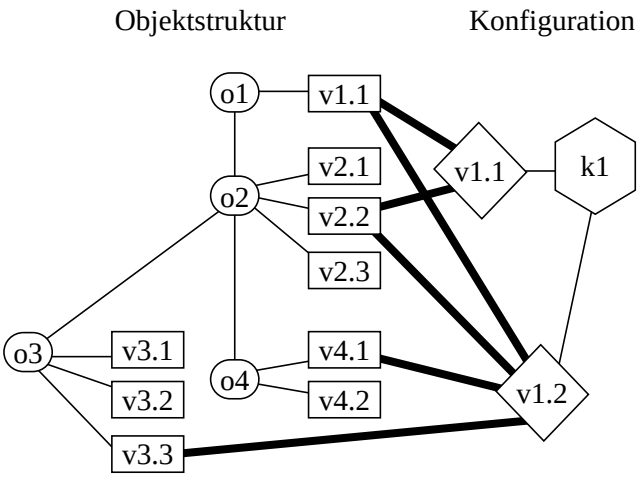


Abb. 7: Beispiel einer *flachen*, versionierten Konfiguration

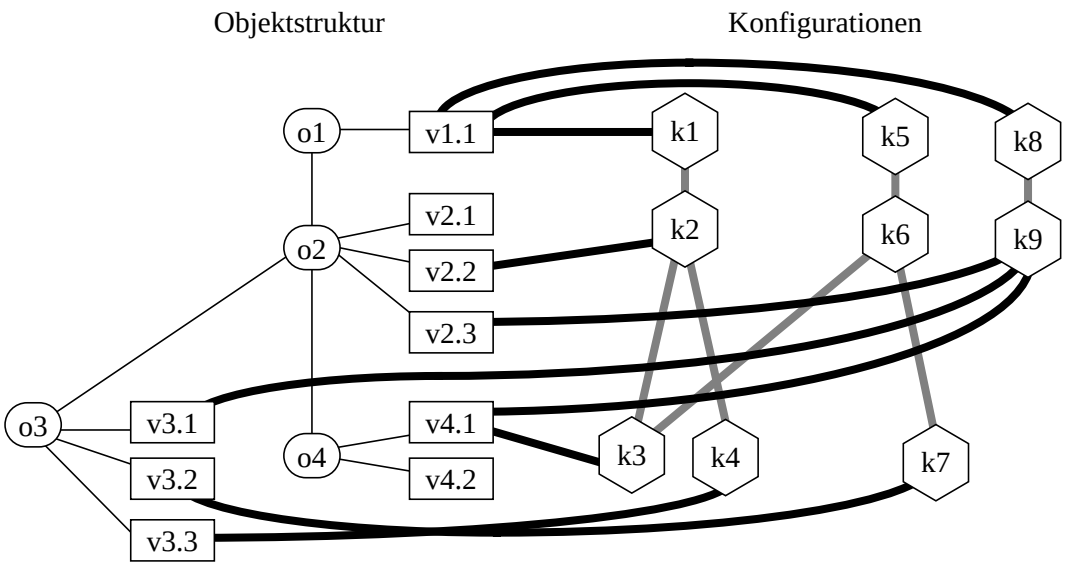


Abb. 8: Beispiele *hierarchischer* Konfigurationen

(vervollständigendes Ändern von Konfigurationsversionen) zu erlauben.

Neben der schwierigen Wartbarkeit flacher Konfigurationen macht das Vorhandensein mehrerer Konfigurationsversionen einer Konfiguration einen neuerlichen Bewertungsschritt, diesmal nicht des erzeugten Objekts, sondern der Konfiguration selbst, notwendig. Ein strukturierteres Vorgehen, bei dem Konfigurationen aus anderen Konfigurationen aufgebaut werden können, führt uns zum Begriff der *hierarchischen* Konfigurationen.

3.2.2 Hierarchische Konfigurationen

Eine ***hierarchische*** Konfiguration besteht aus mehreren (Teil-)Konfigurationen. Jede (Teil-) Konfiguration dient zur Konfigurierung eines disjunkten Teilobjekts des zu konfigurierenden Gesamtobjekts. Im Extremfall konfiguriert eine (Teil-)Konfiguration genau ein Objekt (vgl. explizite, kongruente Konfiguration) oder die gesamte Objekthierarchie (vgl. flache Konfiguration). Abb. 8 zeigt drei strukturierte Konfigurationen (k1, k5 und k8). Konfiguration k1 bildet den expliziten, kongruenten Konfigurationstyp nach, wobei allerdings (Teil-)Konfigurationen genutzt werden können. So wird beispielsweise (Teil-) Konfiguration k3 von k1 und k5 genutzt. Interessantere Möglichkeiten ergeben sich jedoch, wenn flache Konfigurationen in hierarchische Konfigurationen mit einbezogen werden können, wie dies im Falle von Konfiguration k8 gezeigt ist. Hier wird der Teil der Objektstruktur, der im Bild unterhalb von o2 zu sehen ist, durch eine flache Konfiguration (k9) abgedeckt. Dies ermöglicht es, in der Konfigurationsstruktur von der Objektstruktur zu abstrahieren.

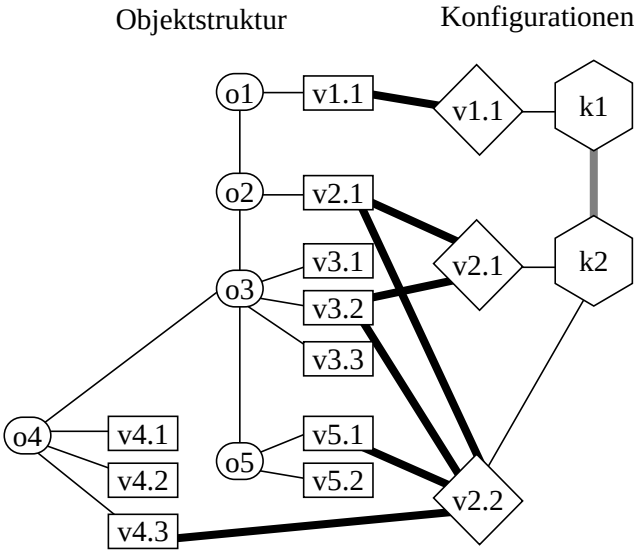


Abb. 9: Beispiel einer *flachen*, versionierten Konfiguration als Teil einer *hierarchischen* Konfiguration

Die Verwendung von Konfigurationen als Teile von hierarchischen Konfigurationen führt allerdings zu Problemen, falls die Teilkonfigurationen versioniert werden. Abb. 9 verdeutlicht dies an einem Beispiel. Die flache Konfiguration k2 ist versioniert und wird als Teil der hierarchischen Konfiguration k1 verwendet. Wenn nun eine neue Version von k1 erstellt wird, so muß festgelegt werden, ob k2 in Version v2.1 oder in Version v2.2 für die hierarchische Konfiguration k1 verwendet werden soll - dies führt offensichtlich zu dem Problem, daß eine ‘Versionsauswahl’ innerhalb einer hierarchischen Konfiguration getroffen werden muß. Es ist demnach sinnvoll, im Falle

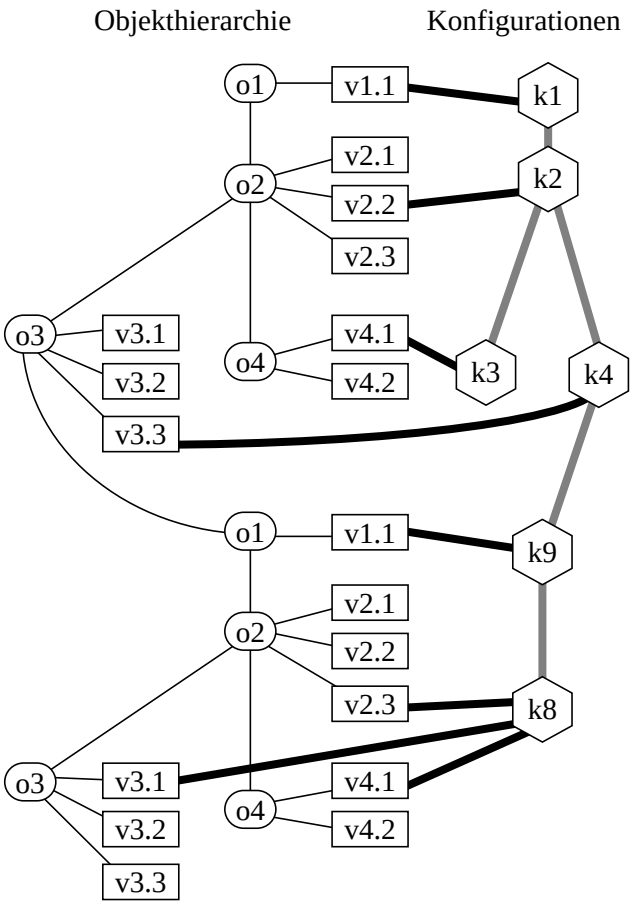


Abb. 10: Beispiel einer ‘rekursiven’, heterogenen Konfiguration

hierarchischer Konfigurationen auf die Versionierung von Teilkonfigurationen zu verzichten.

Eine Unterstützung ***rekursiver*** Konfigurationen, also Konfigurationen von rekursiven Objekten, erfordert keine neuen Konzepte. Insbesondere kongruente Konfigurationen folgen offensichtlich automatisch der rekursiven Objektstruktur. Dies gilt demnach immer für implizite Konfigurationen. Flache Konfigurationen sind hingegen vollkommen von der Objektstruktur losgelöst, so daß die Rekursion als solche nicht in der Konfiguration sichtbar wird; jedoch kann im Falle hierarchischer Konfigurationen Rekursion in einfacher Weise unterstützt werden (vgl. Abb. 10, Objekte und Versionen sind in dieser Abbildung nur der klareren Darstellung wegen dupliziert dargestellt), wenn die Zusammensetzungsbeziehung zwischen den Teilkonfigurationen Rekursion erlaubt (die Rekursion kann auch in den flachen Teilkonfigurationen ‘versteckt’ sein).

Eine Konfiguration heißt ***homogen***, falls von jedem beteiligten Objekt genau eine Version adressiert wird, d. h., falls ein Objekt mehrfach in einer Konfiguration enthalten ist, so wird für jedes Auftreten des Objekts die gleiche Version von ihm verwendet. Falls unterschiedliche Versionen eines Objekts in einer Konfiguration enthalten sind, so heißt die Konfiguration ***heterogen*** (vgl. Abb. 10).

Nachdem wir Konfigurationen klassifiziert haben, werden wir im nachfolgenden Kapitel versuchen, in der Literatur vorgeschlagene Modelle kurz zu charakterisieren und in die gegebene Klassifikation einzuordnen. Danach werden wir in Kapitel 5 einen eigenen Ansatz präsentieren.

4. Konfigurationen in der Literatur

Dieses Kapitel legt den Schwerpunkt der Abgrenzung deutlich auf den Bereich der datenbank-basierten Ansätze. Wir werden anschließend auch einige Ansätze aus dem Bereich der Software-Entwicklung ansprechen, da auch hier der Konfigurationsaspekt eine zentrale Bedeutung hat. Obwohl beide Bereiche in den letzten Jahren an ähnlichen Problemen arbeiteten und vergleichbare Lösungen hervorbrachten, waren sie jedoch immer sehr unabhängig voneinander.

4.1 Datenbank-bezogene Ansätze

Die Betrachtung verwandter Ansätze geschieht jeweils, in dem wir die Grundprinzipien der Konfigurierung kurz darlegen und damit eine Einordnung in die im vorangegangenen Abschnitt gegebene Klassifikation versuchen werden. Aufgrund der Vielzahl von in der Literatur vorgeschlagenen Versionsmodellen kann die folgende Charakterisierung natürlich keinen Anspruch auf Vollständigkeit erheben, soll aber eine repräsentative Auswahl darstellen. Wir verweisen an dieser Stelle auch auf den Übersichtsartikel von R. Katz [15], der einen guten Überblick über bis 1990 vorgeschlagene Modelle gibt und zeigt, daß die meisten dieser Modelle auf gemeinsamen Grundprinzipien beruhen und sogar eine gemeinsame Terminologie eingeführt werden kann. Katz [15] betrachtet ausschließlich Modelle, in denen (hinsichtlich des

Konfigurationsbegriffes) ein hierarchischer Objektbegriff zugrundegelegt ist, d. h., es werden Aggregationshierarchien betrachtet. Damit wird die Konfigurierungsproblematik darauf reduziert zu entscheiden, wie sich die Konfigurierung von Komponenten und Aggregaten in der Dekompositionshierarchie fortsetzt. Entsprechend betrachten diese Modelle keine eigenen Konfigurationsstrukturen und sind daher in unserer Klassifikation in die Gruppe der impliziten Konfigurationen einzuordnen. Sie unterscheiden sich im wesentlichen darin, wie eng der Vorgang der Konfigurierung bzw. die Auflösung generischer Referenzen zeitlich an den Vorgang der Versionsfortschreibung gekoppelt ist. Die ersten drei der in der nachfolgenden Liste angeführten Modelle fallen in diese Gruppe; wir wollen hier im wesentlichen auf die jeweils gewählte Art der impliziten Konfigurierung eingehen und dabei mögliche Spielarten des in Abschnitt angesprochenen verzögerten Auswählens und Propagierens aufzeigen.

McLeod et al.

Der in [27] vorgeschlagene Ansatz verbindet die Konzepte der Versionierung (Oder-Knoten) und der Komposition (Und-Knoten) im Rahmen von Und/Oder-Bäumen. Eine Konfiguration entspricht folglich einer Auflösung der Oder-Verzweigungen. Das Modell bietet jedoch keine spezifische Unterstützung von Beziehungen zwischen Versionen bzw. von Konfigurationen. Da keine Konfigurationsstrukturen unterstützt werden, ordnen wir diesen Ansatz bei den impliziten Konfigurationen ein.

Batory et al.

[3, 2] betrachten sogenannte *molecular objects*, die als Aggregationen einfacherer Objekte aufgefaßt werden können. Diese sind wiederum in Schnittstelle und Implementierung aufgeteilt, wobei unterschiedliche Implementierungen als Versionen des Schnittstellenobjektes betrachtet werden. Das Konzept der *parametrized versions* führt zu einer dynamischen Konfigurierung. Es erlaubt, statt Versionen lediglich das zugehörige Objekt zu referenzieren. Dynamische Konfigurierung bedeutet nun hier, daß eine solche Referenz dann aufgelöst wird, wenn die entsprechende Beziehung traversiert wird. Man unterstellt jedoch die Semantik, daß jede der Versionen des referenzierten Objektes in das betrachtete *molecular object* aufgenommen werden kann. Damit wird ein äußerst schwacher Konfigurationsbegriff unterstützt. Der Ansatz ist als implizite Konfigurierung einzustufen, vermeidet aber den angesprochenen Schneeballeffekt durch die verzögerte Auflösung generischer Referenzen zum Zeitpunkt des Traversierens der entsprechenden Beziehung.

Dittrich et al.

Das in [11] beschriebene Modell definiert ein *design object* als Menge seiner Versionen. Versionen können miteinander in Beziehung stehen, wobei auch hier eine spezielle Art der generischen Beziehungen zur Verfügung gestellt wird, die es erlaubt, Versionen aufgrund bestimmter durch den Benutzer zugeordneten Zustandskennzeichen, wie z. B. *current_version* oder *last_frozen_version*, zu identifizieren. Es liegt demnach auch hier, wenn überhaupt, eine implizite Konfigurierung vor.

Im bereits angesprochenen Übersichtsartikel von Katz [15] werden eine Reihe weiterer Modelle betrachtet, wie z. B. das Modell von Beech und Mahbod [5], das Modell von Landis [24] sowie das Modell von Katz et al. [16, 17]. Ebenso fällt der in [4] beschriebene Ansatz in diese Gruppe. Diese Modelle unterscheiden sich sehr wohl in den unterstützten Versionierungsprinzipien, jedoch nicht sehr in den Konfigurierungsaspekten. In all diesen Modellen liegt, ähnlich wie in den drei oben angeführten, eine Art der impliziten Konfigurierung vor, die auf generischen Referenzen beruht. Lediglich der Zeitpunkt der Auflösung von generischen zu statischen Referenzen und der Einsatz von Notifikationsmechanismen unterscheidet diese Modelle leicht. Wir wollen deshalb nun zu neueren Modellen über.

Wilkes et al.

In Ergänzung des in [40, 22] vorgeschlagenen Versionsmodell beschreibt [41] komplexe Entwurfsobjekte als sogenannte gefaltete Dekompositionshierarchien, in denen aufgrund der Unter-

scheidung von Schnittstellen und Implementierungen mehrfach genutzte Komponenten nur einmal repräsentiert werden müssen. Auf der Implementierungsebene können Entwurfsobjekte/Komponenten versioniert werden. Ein expliziter *Konfigurationsbaum* wird für eine ein komplexes Entwurfsobjekt darstellende, gefaltete Dekompositionshierarchie verwaltet und repräsentiert die entfaltete Hierarchie, die u. a. durch entsprechende Versionsauswahl entsteht. Die Auswahl von Versionen auf den verschiedenen Ebenen kann statisch oder dynamisch organisiert werden. Es handelt sich insgesamt um explizite, kongruente Konfigurationen, die darüberhinaus entsprechend der Entwurfsobjektstruktur hierarchisch aufgebaut sind. Da eine Entfaltung der Entwurfsobjekthierarchie durch den Konfigurationsbaum vorgenommen wird, unterstützt dieses Modell auch heterogene Konfigurationen.

Schwartz, Westfechtel

In [36] wird ein graphbasierter Ansatz zur Verwaltung von (CAD-)Dokumenten, (hierarchisch aufgebauten) Dokumentengruppen und Konfigurationen vorgestellt. Es wird ein bestimmtes Graphmodell zur Verwaltung der angesprochenen Informationsstrukturen vorgegeben, das, obwohl es mit einem anwendungsspezifischen Teil integriert wird, bekannt sein muß, um Transformationen (Graphersetzungen) vornehmen zu können. Der Konfigurationsgraph bildet auch hier (ähnlich wie im Ansatz von Wilkes et al., siehe oben) eine explizite Struktur, wobei jeder Dokumentengruppe ein Konfigurationsknoten zugeordnet ist, wodurch sich explizite, hierarchische Konfigurationen ergeben. Obwohl [36] nicht detailliert darauf eingeht, gehen wir davon aus, daß ausschließlich homogene Konfigurationen erlaubt werden, die zudem versioniert werden können.

Cellary et al.

Dieser Ansatz [6, 8] faßt zunächst versionierbare Objekte in sogenannten Konstellationen zusammen, die als bestimmten Entwerfern zugeordnete Verantwortungsbereiche gesehen werden können. Versionen von Konstellationen werden als Konfigurationen bezeichnet, wobei für jedes in der Konstellation enthaltene Objekt höchstens eine Version in der Konfiguration enthalten sein muß. Es handelt sich demnach in diesem Ansatz um eine explizite Konfigurierung, wobei flache, homogene Konfigurationen unterstützt werden.

Sciore

[34] betrachtet Anfrageergebnismengen als Konfigurationen, so daß diese nur dynamisch bzgl. der Anfrageauswertung auftreten. Die Stärke dieses Modells liegt jedoch darin, daß Schemaelemente, insbesondere benutzerdefinierte Beziehungen bei dieser dynamischen Konfigurierung berücksichtigt werden können und bei Bedarf (d. h., falls nach Anwendungsgesichtspunkten angemessen) auch mehrere Versionen desselben Objektes in einer Konfiguration enthalten sein dürfen. Da jedoch keine Konfigurationsobjekte bzw. -strukturen verwaltet werden, ordnen wir auch diesen Ansatz bei den impliziten Konfigurationen ein. Da, wie bereits gesagt, der Begriff der Konfigurationen hier eigentlich ‘nur’ im Sinne einer strukturierten Ausgabe von Anfrageergebnismengen benutzt wird, handelt es sich eigentlich um *dynamische*, implizite Konfigurationen.

Golendziner, Santos

Ähnlich wie bei den zu Beginn dieses Abschnittes angesprochenen Ansätzen der impliziten Konfigurierung besteht auch in [14] die Konfigurierung im wesentlichen in der Auflösung von generischen Referenzen. Als besonderen Vorteil dieses Ansatzes nennt [14] die Möglichkeit der gleichförmigen Behandlung von Versionen und Konfigurationen. Dabei ist jedoch zu beachten, daß eigentlich keine (expliziten) Konfigurationen unterstützt werden, es wird vielmehr eine Version als ‘konfiguriert’ betrachtet, wenn in ihrem (bzgl. benutzerdefinierter und spezieller IS_A-Beziehungen transitiven) Umfeld keine nicht-aufgelösten generischen Referenzen mehr existieren. Wie bei allen (genannten) impliziten Ansätzen liegt ein Nachteil darin, daß aufgrund fehlender Typisierung der Konfigurationen die Konfigurierung nicht gezielt auf bestimmte Schemaelemente be-

zogen werden und damit keine dedizierte systemgestützte Kontrolle angeboten werden kann.

ObjectStore

Als einen repräsentativen, in existierenden objektorientierten Datenbankverwaltungssystemen unterstützten Ansatz wollen wir den im DBVS ObjectStore [23] gewählten ansprechen. Hier steht die Konfigurierung (zeitlich) vor dem Anlegen von Versionen. Dies bedeutet, daß die Menge der Objekte, die als Einheit der Versionierung betrachtet werden soll, als Konfiguration bezeichnet wird. Da wir in diesem Artikel die Konfigurierung als Mittel der Bildung konsistenter Dateneinheiten 'oberhalb' der Versionierung von (komplexen) Objekten diskutieren, betrachten wir solche Ansätze, wie den in ObjectStore verfolgten, nicht weiter.

4.2 SE-Ansätze

Nachdem wir im vorangegangenen Abschnitt der Intension dieses Artikels entsprechend eine Charakterisierung von Ansätzen, die direkt im Datenbankbereich angesiedelt sind und der Unterstützung technischer Entwurfsbereiche wie CAD dienen, vorgenommen haben, wollen wir nun eine Kurzcharakterisierung des Konfigurationsbegriffs der Software-Entwicklung (SE) geben [38]. Auch in diesem Bereich hat die Konfigurationsverwaltung eine hohe Bedeutung, was auch daran zu erkennen ist, daß es zum Thema *Software-Configuration-Management (SCM)* eine eigene Workshop-Reihe gibt [37]. Zum Zwecke einer Kurzcharakterisierung orientieren wir uns an [9, 10], die einen guten Überblick über den aktuellen Stand im Bereich SCM geben. Dort werden Konfigurierungswerkzeuge zunächst eingeteilt in die Klassen *change-oriented* und *version-oriented*.

Bei ersteren (*change-oriented*) handelt es sich um solche, die dem Benutzer eine versionsfreie Sicht auf die Software-Dokumente geben. Dies geschieht, in dem einerseits (statt eigentlicher Versionen) Änderungen (intern) verwaltet werden und beim Zugriff jeweils ein impliziter Filter benutzt wird, der es erlaubt, Versionen ausgehend von den Änderungsprotokollen zu berechnen und zu einer gültigen Konfiguration zusammenzusetzen. Ein prominenter Vertreter dieser Klasse ist der COV-Ansatz [28]. Bei solchen Ansätze handelt es sich also (bzgl. der im vorangegangenen Kapitel gegebenen Klassifikation) um implizite Konfigurationen, oder genauer um dynamische, implizite Konfigurationen wie im Ansatz von Sciore. Hier werden die Konfigurationen jedoch nicht durch Anfragen des Benutzers gebildet, sondern durch vordefinierte (spezifische) Filter/Kontrollanweisungen.

Die als *version-oriented* bezeichnete Klasse von Konfigurierungswerkzeugen basiert auf der Organisation von Dokument- und Versionierungsstrukturen in Und/Oder-Bäumen. In der Regel werden hier Dateien als Gegenstand der Versionierung betrachtet. Ein frühes Beispiel ist das RCS-Werkzeug [39]. Es erlaubt die Bildung von Versionsgruppen und die Annotation der Versionen mit Zustandskennzeichen (z. B. 'freigegeben'). Zur Konfigurationsbildung wird es ermöglicht, beim Checkout über die Zustandskennzeichen miteinander verträgliche Versionen (verschiedener Versionsgruppen) zu spezifizieren. Weitere Vertreter sind ClearCase [26], CONFIG [42], Adele [13], SIO [25] und ICE [45], von denen wir lediglich auf die letzten beiden kurz eingehen wollen.

SIO erweitert relationale Datenbanktechnologie. Tupel beschreiben Versionen. Konfigurationen können explizit erzeugt werden mit Hilfe von SQL-ähnlichen, deskriptiven Anfragen. Die Kompatibilität von Versionen kann durch explizite Bedingungen beschrieben werden, die den bekannten Assertions des Relationenmodells ähneln.

ICE ist das Konfigurierungswerkzeug der Software-Entwicklungsumgebung NORA. Es basiert auf Feature-Logik, auf deren Basis Verträglichkeiten von Versionen beschrieben werden (Feature-Terme). Auch hier können Versionen explizit erzeugt werden und sogar eine schrittweise Konfigurierung durchgeführt werden.

Somit sind die in SIO und ICE unterstützten Konfigurationen bezüglich unserer Klassifikation als explizite Konfigurationen zu betrachten. Da je-

doch keine Struktur unterstützt wird, handelt es sich um flache Konfigurationen.

4.3 Bewertung

Die in diesem Abschnitt betrachteten verwandten Ansätze weisen sehr unterschiedliche Konfigurierungsmechanismen auf. Unser Ziel ist die Bereitstellung eines geeigneten Konfigurierungskonzeptes für technische Entwurfsanwendungen. Wir denken, daß hier zunächst für den Benutzer sichtbare Versionen angeboten werden müssen, da der Entwerfer eine Sicht auf die Entwicklungsgeschichte seines Entwurfsobjektes benötigt, um geeignete Entwurfsentscheidungen treffen zu können. Dies gilt auch für die Konfigurationsebene. Daher scheiden implizite Prinzipien aus; insgesamt halten wir diese in technischen Entwicklungsumgebungen aus folgenden Gründen für ungeeignet:

- die spezielle Semantik der Beziehungen, die als generische Referenzen abgebildet werden, wird nicht berücksichtigt;
- die geforderte Homogenität bei der impliziten Konfigurierung kann Anwendungsanforderungen entgegenstehen;
- es wird keine anwendungsspezifische Kontrolle ausgeübt, d. h., das Ausmaß der Auflösung von generischen Referenzen kann nicht direkt festgelegt werden;
- der Benutzer kann nicht flexibel genug mit Konfigurationen arbeiten, d. h. er kann nicht dynamisch um- bzw. weiterkonfigurieren;
- es wird auf Konfigurationsebene keinerlei Möglichkeit der Abstraktion und der Verdichtung von konfigurationsspezifischer Information gegeben, da kein(e) explizite(s) Konfigurationsobjekt/-struktur verwaltet wird.

Geeignete explizite Konfigurierung bieten folglich lediglich die Ansätze von Wilkes et al., Schwartz und Westfechtel (siehe Abschnitt 4.1) sowie einige Software-Konfigurierungswerkzeuge der Klasse *version-oriented*. Diese Modelle orientieren sich jedoch sehr stark an Aggregationsbeziehungen und durch diese gebildete Hierarchien und erlauben daher nicht die Berücksichtigung beliebiger, anwendungsspezifischer Beziehungssemantik auf der Konfigurationsebene. Wir werden im folgenden Abschnitt einen Ansatz vorstellen, der diese Flexibilität bietet.

5. Konfigurationen im Objekt- und Versionsdatenmodell (OVM)

5.1 Grundkonzepte des OVM

Das Objekt- und Versionsdatenmodell OVM [18, 21] setzt direkt die in Kapitel 2.2 genannten Anforderungen um [20]. Im folgenden werden wir

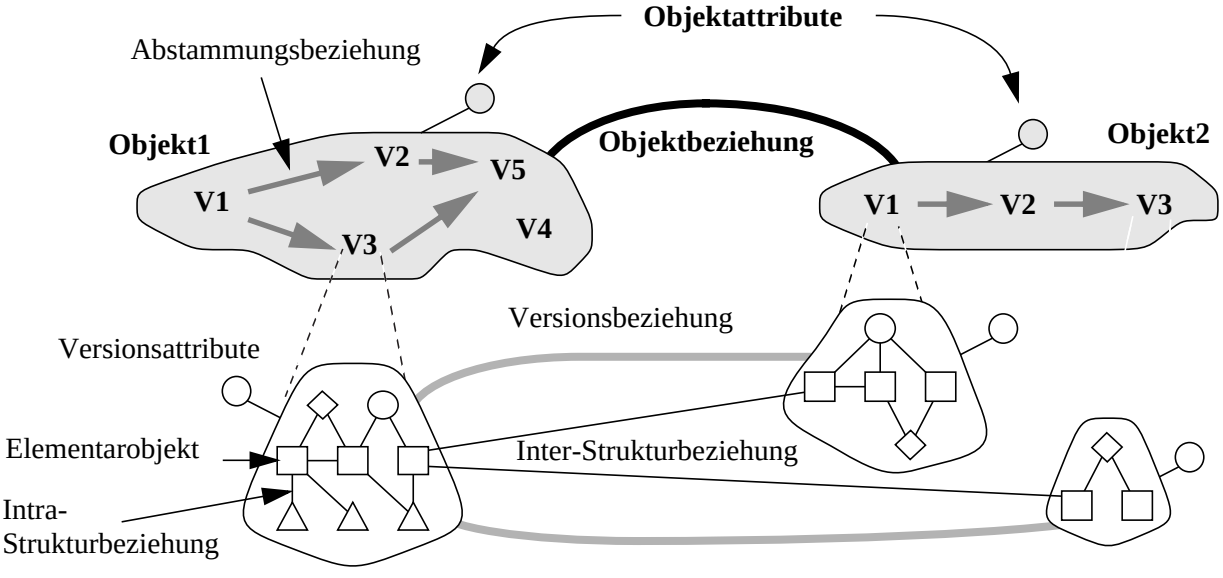


Abb. 11: OVM Grundkonzepte

zunächst die Grundkonzepte des OVM erläutern, um die Besprechung des Konfigurationskonzeptes vorzubereiten. Wir können an dieser Stelle keine ausführliche Beschreibung des Modells liefern, sondern müssen uns auf die Skizzierung der zur anschließenden Diskussion des Konfigurationsbegriffes notwendigen Grundbegriffe beschränken.

5.1.1 Komplexe Objekte und explizite Versionen komplexer Objekte

Komplexe Objekte sind eindeutig benennbare Ausprägungen von komplexen Objekttypen und stellen strukturierte Ansammlungen von elementaren Daten dar. Ein komplexes Objekt ist in seiner Gesamtheit durch eine Reihe von *Objektattributen* beschrieben. Ein Objekt faßt sogenannte *Elementarobjekte* zusammen, die den Tupeln des Relationenmodells entsprechen und in Abb. 11 durch kleine Kreise, Quadrate und Rauten symbolisiert sind. Sie sind ebenfalls durch Attribute, die *Elementarobjektattribute*, beschrieben (nicht in Abb. 11 dargestellt). Über das Relationenmodell hinausgehend können Elementarobjekte durch (typisierte) *Strukturbeziehungen* verbunden sein. Offensichtlich besteht eine der Hauptaufgaben des Entwurfs darin, 'Inhalte' von Objekten, d. h. Netze von Elementarobjekten (aus Sicht der Datenhaltung), zu erzeugen. Der Entwurf ist im allgemeinen ein iterativer Prozeß, in dem üblicherweise mehrere ähnliche Elementarobjektnetze entstehen, die wir im folgenden als ***Versionen*** bezeichnen. Versionen sind somit *unterschiedliche Zustände der im Objekt zusammengefaßten Elementarobjektnetze*[1]. Analog zu den Objekten können die in Versionen zusammengefaßten Daten in ihrer Gesamtheit durch *Versionsattribute* charakterisiert werden. 'Erzeuger' und 'Teststatus' sind typische Beispiele für Versionsattribute. Die Abhängigkeiten zwischen (Ausgangs-)Versionen und den daraus entwickelten (neueren) Versionen, die die Fortentwicklung der beteiligten Objekte beschreiben, werden in *Abstammungsgraphen* repräsentiert. Der Abstammungsgraph ist gerichtet und kann sich als Liste, Baum oder allgemein als azyklischer Graph entwickeln (vgl. Abb. 1).

1. Demzufolge werden hier explizite Versionen modelliert, wobei das komplexe Objekt (und *nicht* das Elementarobjekt oder das Attribut) das Granulat der Versionierung bildet.

5.1.2 Beziehungen zwischen Objekten und zwischen Versionen

Objektbeziehungen repräsentieren zwischen Objekten bestehende Relationen [43]. Offensichtlich gelten diese Beziehungen auch für die Versionen der entsprechenden Objekte. Das heißt, ***Versionsbeziehungen*** treten als Verfeinerungen der Objektbeziehungen auf. Sie stellen Beziehungen zwischen Versionen dar, deren Objekte durch eine Objektbeziehung verbunden sind. Dies bedeutet, daß Versionsbeziehungen von der Existenz entsprechender Objektbeziehungen abhängig sind bzw. durch diese erst legitimiert werden.

Zusätzlich zu den beschriebenen *expliziten* Verknüpfungen zwischen Versionen können weitere *implizite* Beziehungen vorhanden sein. Sie ergeben sich durch *Datenüberlappungen* als Folge von überlappenden Objekttypdefinitionen. Die Datenüberlappungen können dabei über Elementarobjekte selbst oder über sogenannte *Inter-Strukturbeziehungen* entstehen, die im Gegensatz zu *Intra-Strukturbeziehungen* Objektversionsgrenzen überschreiten.

Beziehen wir uns auf Abb. 11 und nehmen wir den (durch Anwendungssemantik bestimmten) Fall an, daß es sich bei der zugehörigen Objektbeziehung um eine (1:1)-Beziehung handelt. Beispielsweise könnte es sich um eine Karosserie (object1) und einen Motor (object2) handeln; es ist klar, daß zu einem Auto genau eine Karosserie und genau ein Motor gehören. Diese (1:1)-Zuordnung muß jedoch nur auf Objektebene gewahrt werden; auf Versionsebene hingegen können einer Karosserieversion mehrere (zu ihr passende) Motorversionen zugeordnet werden. Dieses Beispiel macht die Notwendigkeit einer anschließenden Konfigurierung deutlich. In Bezug auf Abb. 11 bedeutet dies, daß Version 3 von Objekt 1 entweder nur mit Version 1 oder nur mit Version 3 von Objekt 2 in Beziehung stehen darf. Dabei wird auch deutlich, daß *Inter-Strukturbeziehun-*

gen nur innerhalb einer Konfiguration interpretiert werden sollten. In unserem Fall müßten also zwei Konfigurationen (Objekt 1, Version 3 mit Objekt 2, Version 1 und Objekt 1, Version 3 mit Objekt 2, Version 3) gebildet werden. Offensichtlich beeinflußt die Objektüberlappung wiederum den *Konfigurierungsvorgang* [5, 15].

Zusammenfassend haben wir in diesem Abschnitt drei Faktoren identifiziert, die für die Konfigurierung in Versionsumgebungen wichtig sind:

- *Objektbeziehungen* beschreiben Beziehungen zwischen versionierten Objekten.
- *Versionsbeziehungen* beschreiben 'Kompatibilitäten' zwischen Versionen.
- *Objektüberlappungen* beschreiben weitere Abhängigkeiten zwischen Versionen.

5.2 Konfigurationen im OVM

OVM unterstützt das gesamte Spektrum *kongruenter* und *partieller, expliziter* Konfigurationen. Im folgenden werden wir den Konfigurationsbegriff des OVM näher erläutern und entsprechende Beispiele geben.

5.2.1 Definition von Konfigurationstypen

Um die verschiedenen Möglichkeiten der Definition von Konfigurationstypen, die das OVM bietet, zu erläutern, greifen wir auf das in Abschnitt 2.2 gegebene VLSI-Beispiel zurück (siehe Illustration des Makrodatenschemas in Abb. 2 und der Ausprägungsstruktur in Abb. 3).

Flache Konfigurationen

Durch die Möglichkeit, eine Schnittstelle (siehe Komplexobjekttyp Interface in Abb. 2) durch mehrere alternative Implementierungen (Beziehungstyp Implementation und Komplexobjekttyp Contents) zu realisieren, sowie durch die Tatsache, daß sich hinter den in Abb. 2 dargestellten Objektstrukturen aufgrund der möglichen Versionierung aller Objekte weitaus komplexere Versionsstrukturen ergeben können, entsteht ein Konfigurationsbedarf. So muß letztendlich für eine Interface-Version genau eine zugehörige Contents-Version ausgewählt werden. Dies kann beispielsweise mit dem Konfigurationstyp *Cell1* realisiert werden (siehe zugehörige Definition in Abb. 12).

DEFINE CONFIGURATION_TYPE Cell1
FOR Instance
ATTRIBUTES
Cell_Id**:** **IDENTIFIER,**
Name**:** **STRING**
VIA Path1**:** Instance-<Instantiation>Interface-<Implementation>Contents [1,1]
OBJECT_SELECTION;

Abb. 12: Definition eines flachen, homogenen Konfigurationstyps

Der Konfigurationstyp Cell1 ordnet einer Instance-Version genau eine Contents-Version zu. Der dem Konfigurationstyp zugrundeliegende Schemaausschnitt ist in der VIA-Klausel beschrieben. Es handelt sich hierbei um einen sogenannten *Pfad* (hier mit dem expliziten Namen Path1), der insgesamt eine (1:n)-Beziehung bildet. So können auf dem in diesem Beispiel beschriebenen Pfad von einer Instance-Version aus n Contents-Versionen erreicht werden. Durch die Angabe von Kardinalitätsrestriktionen kann weiter eingeschränkt werden, wieviele der ausgehend von einer (zu konfigurierenden) Instance-Version aus erreichbaren Contents-Versionen in die Konfiguration mit aufgenommen werden dürfen. Zusätzlich kann einer der Auswahlmodi OBJECT_SELECTION, VERSION_SELECTION, NO_SELECTION angegeben werden. Diese haben auf den Vorgang des Konfigurierens, d. h. des Erstellens einer Ausprägung des betrachteten Konfigurationstyps, folgende Auswirkung: OBJECT_SELECTION erzwingt, daß aus der Menge auf dem angegebenen Pfad erreichbaren Blattversionen höchstens eine Version pro Objekt ausgewählt wird; VERSION_SELECTION erlaubt beliebige Auswahl aus der Menge der erreichbaren Versionen, was wiederum die Grundlage für die Erzeugung heterogener Konfigurationen ist.(der eigentliche Auswahlvorgang bei VERSION_SELECTION und OBJECT_SELECTION wird im folgenden Abschnitt erläutert); NO_SELECTION erzwingt die Aufnahme aller erreichten Version in die Konfiguration. Die Angabe OBJECT_SELECTION ist in obigem Beispiel aufgrund der Angabe der Kar-

dinalitätsrestriktion ([1,1]) eigentlich unnötig; sie wurde lediglich hinzugefügt, um die verschiedenen Auswahlmodi ansprechen zu können.

```
DEFINE CONFIGURATION_TYPE Cell2
FOR Contents
ATTRIBUTES
        Cell_Id:  IDENTIFIER,
        Name:     STRING
VIA     Path1:    Contents-<Interface>-
                  Supercell(Instance) [1,1]
VIA     Path2:    Contents-<Aggregation>-
                  Subcell(Instance)
                  VERSION_SELECTION;
```

Abb. 13: Definition eines flachen Konfigurationstyps, der heterogene Ausprägungen zuläßt

Das obige Beispiel macht bereits deutlich, daß im OVM jede Konfiguration genau eine Version des in der FOR-Klausel angegebenen Objekttyps (*Konfigurationsobjekttyp*) beinhaltet und dieser weitere Versionen zugeordnet werden können, die über (evtl. mehrstufige) Versionsbeziehungen, d. h. über einen in einer VIA-Klausel spezifizierten Pfad erreichbar sind. Das OVM erlaubt auch die Angabe mehrerer Pfade innerhalb eines Konfigurationstyps, so daß eine Konfigurationstypdefinition letztlich einer Hierarchie von Pfadspezifikationen entspricht. Abb. 13 zeigt die Definition eines Konfigurationstyps *(Cell2*) mit zwei Konfigurationsbeziehungstypen, die beide vom Konfigurationsobjekttyp Contents ausgehen. Da der Objekttyp Instance mehrmals in der Definition vorkommt, müssen entsprechende Rollennamen hinzugefügt werden. Ein einfaches Beispiel für eine hierarchische Anordnung von Pfadspezifikationen geben wir anschließend.

Hierarchische/rekursive Konfigurationen

Die beiden oben angegebenen Beispielkonfigurationstypen betrachten jeweils nur einen Hierarchieübergang in der Modul-Hierarchie der beispielhaften VLSI-Modellierung. Der im folgenden Beispiel (siehe Abb. 14) definierte Konfigurationstyp *Cell3* erlaubt die Konfigurierung der gesamten Hierarchie und damit des gesamten Chips. Hierarchische Konfigurationstypen können mit Hilfe der FOR-USE-Klausel definiert werden. So ist es im Beispiel des Konfigurationstyps Cell3 erlaubt, statt einer Subzellen-Version vom Typ Instance (siehe Endknoten des Pfads Path2) eine (untergeordnete) Subzellen-Konfiguration (vom Typ Cell3) einzuhängen. Somit kann man mittels dieser Klausel nicht nur hierarchische, sondern wie in unserem Beispiel auch rekursive Konfigurationstypen definieren.

```
DEFINE CONFIGURATION_TYPE Cell3
FOR  Instance
ATTRIBUTES
        Cell_Id:  IDENTIFIER,
        Name:     STRING
VIA     Path1:    Supercell(Instance)-
                  <Instantiation>Interface-
                  <Implementation>Contents [1,1]
                  (VIA Path2:  Contents-
                               <Aggregation>-
                               Subcell(Instance))
FOR  Subcell USE Cell3;
```

Abb. 14: Definition eines rekursiven, hierarchischen Konfigurationstyps

Es ist wohl einfach nachzuvollziehen, daß mit den beschriebenen Möglichkeiten kongruente und partielle Konfigurationstypen definiert werden können.

5.2.2 Anlegen und 'Verarbeiten' von Konfigurationen

In der Überschrift dieses Abschnitts wurde *Verarbeiten* in Hochkommata gesetzt, da Konfigurationen nicht im üblichen Sinne verarbeitet werden können. Wie bereits des öfteren betont, dienen Konfigurationen lediglich als eine Art 'Container' für 'zusammenpassende' Versionen. In diesem Sinne dienen Konfigurationen eher dem gezielten Zugriff auf die Menge der jeweils enthaltenen Versionen, um diese (Versionsmenge) dann zu verarbeiten. Dementsprechend bietet das OVM eine CREATE-Anweisung zum Anlegen von Konfigurationen, eine SELECT-Anweisung zum Selektieren der durch eine Konfiguration repräsentierten Menge von Versionen und eine DELETE-Anweisung, die lediglich die Menge der Versionen freigibt, d. h., das Löschen der Konfiguration impliziert nicht das Löschen der enthaltenen Versionen aus der Datenbank[1]. Wir wollen des-

halb im folgenden nur die Operation zum Anlegen von Konfigurationen betrachten.

Im allgemeinen können Konfigurationen in mehreren Schritten angelegt bzw. ergänzt werden. Die Konfigurierung (Anlegen einer Konfiguration) beginnt mit der Spezifikation einer Version des Konfigurationsobjekttyps. Dies kann deskriptiv oder durch direkte Angabe eines Version-Identifikators geschehen. In den weiteren Schritten können jeweils Auswahlvorgänge für die einzelnen im betroffenen Konfigurationstyp definierten Pfade durchgeführt werden.

Beispielhaft wollen wir eine Konfiguration vom Typ Cell3 (siehe Definition in Abb. 14) auf den in Abb. 3 dargestellten Ausprägungen anlegen (man beachte, daß es sich in Abb. 3 um eine Darstellung von Objekten handelt und sich dahinter eine weitaus komplexere Versionsebene verbergen kann). Wir wollen dies in zwei Schritten tun. Abb. 15 zeigt eine Anweisung, die den ersten Pfad (siehe erste VIA-Klausel in Abb. 14) belegt. Durch die explizite Angabe eines Identifikators (ISV1)[1] für die Superzelle wird die Wurzel der Konfiguration ausgewählt. Nun können entsprechende Auswahlvorgänge für die von dem Konfigurationsobjekttyp startenden Pfade vorgenommen werden. Im Beispiel ist dies nur der Pfad mit dem Namen Path1. Bei diesem Auswahlvorgang ist nun davon auszugehen, daß das Versionsverwaltungssystem alle von dem Ausgangsobjekt (hier Superzelle mit dem Identifikator ISV1) aus über den angegeben Pfad erreichbaren Versionen ermittelt (hier alle erreichbaren Contents-Versionen). Aus diesen kann der Benutzer nun diejenigen auswählen, die in die Konfiguration übernommen werden sollen, solange diese Auswahl mit den Spezifikationen des Konfigurationstyps, z. B. Auswahlmodus, verträglich ist. Die Auswahl kann im allgemeinen deskriptiv geschehen; wir müssen jedoch in diesem Papier aus Platzgründen auf eine eingehende Beschreibung des Sprachansatzes verzichten. Im Beispiel (Abb. 15) ist ein Prädikat in intuitiver Notation angegeben. Es besagt, daß die neueste Version des Contents-Objektes, das in Abb. 3 dunkelgrau hinterlegt ist, übernommen werden soll. Dies führt uns zu einer Teilkonfiguration, die in nachfolgenden Anweisungen vervollständigt werden kann.

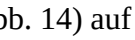
CREATE Cell3 **CONFIGURATION**
{ Name **:=** ’Beispiel-Konf. des Typs Cell3’ }
Supercell = **’**ISV1**’**
{ Path1 **PREDICATE**
Contents**.OBJECT LIKE**  **AND**
Contents**.VERSION** = **TOPIC** }**;**

Abb. 15: (Partielles) Anlegen einer Konfiguration vom Typ Cell3

Angenommen, die Ausführung der in Abb. 15 angeführten Anweisung führt zu einer Konfiguration mit dem Identifikator Cell3K1. Diese kann anschließend mit der in Abb. 16 gezeigten Anweisung vervollständigt werden. Mit dieser Anweisung wird der zu Pfad Path2 gehörende Auswahlvorgang durchgeführt, in dem bereits vorab erzeugte Konfigurationen als Sub-Konfigurationen eingebunden werden. Bei expliziter Angabe von Subkonfigurationen (durch Auflistung der Konfigurationsidentifikatoren) prüft das System selbstverständlich, ob die entsprechenden Wurzelversionen der Subkonfigurationen über den angegebenen Pfad mit den entsprechenden (bisherigen) Blattversionen der Superkonfiguration datenbankseitig verbunden sind.

CREATE Cell3 **CONFIGURATION**
USES **’**Cell3K1**’**
{ Path2 **IDENT_LIST (’**Cell3K2**’,**
’Cell3K3**’, ’**Cell3K4**’,**
’Cell3K5**’)};**

Abb. 16: Vervollständigung einer Konfiguration vom Typ Cell3

Mit den gegebenen Beispielen sollte ein Überblick über die Konzepte der Unterstützung von Konfigurationen im OVM vermittelt werden. Selbstverständlich sollte der Auswahlvorgang,

1. Selbstverständlich können bei der Verarbeitung von Versionen boolsche Operatoren, die das Enthaltensein von Versionen in Konfigurationen testen, benutzt werden.

1. Es handelt sich hierbei um systemvergebene Surrogate; d. h., beim Speichern einer neuen Konfiguration erzeugt das System einen Identifikator und gibt ihn als Ergebnis der Speicherungsoperation zurück. Diese Identifikatoren können auch vom Benutzer explizit erfragt und, wie in diesem Fall, in weiteren Anweisungen benutzt werden.

den der Prozeß des Konfigurierens darstellt, durch eine geeignete Benutzerschnittstelle unterstützt werden. So sollten in diesem Zusammenhang entsprechende Browsing-Funktionen zur Unterstützung der Entscheidungsfindung angeboten werden, bzw. graphische Auswahlmöglichkeiten, die dann intern in die erläuterten CREATE-CONFIGURATION-Anweisungen umgesetzt werden, bereitgestellt werden.

5.2.3 Bewertung des OVM-Konfigurationsansatzes

In den beiden vorangegangenen Kapiteln wurde bereits an einigen Stellen eine Bewertung der verschiedenen möglichen Ansätze für Konfigurationen hinsichtlich ihrer Nutzbarkeit in technischen Entwurfsanwendungen gegeben und damit eine Begründung der Entscheidung zugunsten der in das OVM integrierten Konzepte gegeben. Ergänzend wollen wir an dieser Stelle den gewählten Ansatz hinsichtlich der nachfolgend aufgeführten Aspekte betrachten bzw. bewerten.

- Modellierung
  Die im OVM explizit modellierten Versionsbeziehungen bieten eine natürliche Grundlage für die Beschreibung der ‘Kompatibilität’ von Versionen. Sie beschreiben jedoch alle möglichen Kompatibilitäten, so daß in Ergänzung die Konfigurierung zur Beschreibung der ‘Gesamt-Kompatibilität’ innerhalb einer komplexen Versionsstruktur notwendig ist. Dies wird durch das beschriebene Konzept in natürlicher Weise unterstützt, so daß sich ein für die Anwendung intuitiver Konfigurationsbegriff ergibt.
- Handhabbarkeit
  Entsprechend ergibt sich auch eine einfache Handhabbarkeit der OVM-Konfigurationen. Das Konfigurieren orientiert sich an bekannten Schemastrukturen und die durch den Benutzer durchzuführenden einzelnen Auswahlvorgänge werden durch die Möglichkeit der schrittweisen Erstellung sowie durch mögliche Hierarchisierung vereinfacht.
- Integrität
  Nachdem der Auswahlvorgang an sich sowie die damit verbundene Zuordnung von Versionen zu einer Konfiguration natürlicherweise der Benutzerkontrolle unterliegt, können selbstverständlich neben den im Konfigurationstyp spezifizierten strukturellen Integritätsbedingungen auch inhaltliche (wertebezogene) Integritätsbedingungen spezifiziert werden. Dazu wird eine auf das Konfigurationskonzept zugeschnittene Klasse von semantischen Integritätsbedingungen angeboten, die eine ähnliche Semantik wie die aus relationalen Systemen bekannten Assertions [12] haben (die natürlich auf die hinter ‘unseren’ Konfigurationen stehenden komplexen Versionsstrukturen zugeschnitten worden sind). Leider müssen wir aus Platzgründen auf eine eingehende Diskussion dieses Aspektes verzichten.
- Realisierung
  Das beschriebene Konfigurationskonzept wurde in zwei prototypischen Implementierungen des OVM integriert. Beide Implementierungen folgen einem Zusatzebenen-Ansatz; ersterer [18] auf dem Non-Standard-DBMS PRIMA, der zweite [30] auf dem OODBMS ObjectStore. Die letztgenannte Implementierung vervollständigte das prototypische Datenbankverwaltungssystem VStore [30], das wir als Testplattform und beispielhaftes DBMS zur Nutzung innerhalb eines CAD-Frameworks [44] nutzen.

6. Schlußbemerkung

Mit diesem Artikel haben wir versucht, eine allgemeine Klassifikation von Konfigurationskonzepten für technische Entwurfsanwendungen zu geben. Das Ergebnis dieser Bemühungen ist der in Abb. 3 dargestellte Klassifikationsbaum. Das in Kapitel 5 dargestellte, in unser Objekt- und Versionsdatenmodell integrierte Konfigurationskonzept entstand aus einer Bewertung aller hier betrachteten Konfigurationskonzepte im Hinblick auf eine geeignete Unterstützung technischer Ent-

wurfsanwendungen. Die Gründe für dieses Konfigurationskonzept lassen sich wie folgt zusammenfassen:

- Konfigurationen sollten nicht notwendigerweise immer kongruent zu den Objektstrukturen sein, da flache Konfigurationen eher der Anforderung der Abstraktion nachkommen; trotzdem sollten für bestimmte Anwendungsfälle auch kongruente Konfigurationen beschrieben werden können.
- Da die beispielsweise im OVM unterstützten Versionsbeziehungen (als Verfeinerungen von Objektbeziehungen) auf natürliche Weise dazu beitragen, die 'Kompatibilität' von Versionen auszudrücken, und das Ziel einer jeden Konfigurierung die Zustammenstellung einer (nach Anwendungsgesichtspunkten) konsistenten Menge von Versionen ist, sollten Versionsbeziehungen als Grundlage der Konfigurierung dienen.
- Eine Versionierung der Konfigurationen selbst ist aus Übersichtlichkeitsgründen und hinsichtlich einer einfachen Handhabung hierarchischer Konfigurationen eher hinderlich.
- Hierarchische und rekursive Konfigurationen unterstützen ein strukturiertes Anlegen von Versionen (Konfigurieren) und sind deshalb als äußerst benutzerfreundlich zu bewerten.
- Die Unterstützung partieller Konfigurationen und die damit verbundene Möglichkeit des 'Weiterkonfigurierens bestehender Konfigurationen' hilft ähnlich wie Hierarchisierung (siehe vorgenannter Aspekt) dabei, komplexe Strukturen einfacher zu handhaben.
- Heterogenität/Homogenität von Konfigurationen dürfen nicht im Modell direkt verankert werden; vielmehr liegen hier Abhängigkeiten zur Objektmodellierung vor, so daß die Entscheidung, ob ein Konfigurationstyp heterogen oder homogen sein sollte, von der Anwendung getroffen werden können muß.

Alle diese Punkte werden von dem im OVM realisierten Konfigurationskonzept reflektiert. Dadurch eignet sich das beschriebene Konfigurations- sowie das darunterliegende Versionsmodell zur Nutzung innerhalb eines datenbank-basierten CAD-Frameworks, denn es wird sowohl die entwurfswerkzeug-orientierte Sicht als auch die entwurfsprozeß-orientierte Sicht unterstützt und die notwendige Flexibilität für die Verarbeitung von Versionen und Konfigurationen durch mehrere kooperierende Entwerfer geboten.

Danksagung: Wir danken Prof. Dr. T. Härder sowie den anonymen Gutachtern für die konstruktive Kritik.

Literatur

1. Agrawal, R., Buroff, S., Gehani, N., Shasha, D.: Object Versioning in Ode. Proc. Int. Conf. on Data Engineering, Kobe, Japan, 1991, S. 446-455
2. Batory, D.S., Buchmann, A.P.: Molecular Objects, Abstract Data Types and Data Models: A Framework. Proc. 10th Conf. on Very Large Data Bases, VLDB 84, Singapore, 1984, S. 172-184
3. Batory, D., Kim, W.: Modeling Concepts for VLSI CAD objects. ACM TODS, Vol. 10, No. 3, S. 322-346
4. Blanken, H.: Implementing Version Support for Complex Objects. Data & Knowledge Engineering, Vol. 6, 1991, S. 1-25
5. Beech, D., Mahbod, B.: Generalized Version Control in an Object-Oriented Database. Proc. of the 4th Int. Conf. on Data Engineering, 1988, S. 14-22
6. Cellary, W., Jomier, G.: Consistency of Versions in Object-Oriented Databases. Proc. 16th Int. Conf. on Very Large Data Bases, VLDB 90, Brisbane, 1990, S. 432-441
7. Chou, H.T., Kim, W: A Unifying Framework for Versions in a CAD Environment. Proc. 12th Int. Conf. on Very Large Data Bases, VLDB 86, Kyoto, Japan, 1986, S. 336-344
8. Cellary, W., Vossen, G., Jomier, G.: Multiversion Object Constellations for CAD Databases. Bericht Nr. 9105, Arbeitsgruppe Informatik, Universität Giessen
9. Conradi, R., Westfechtel, B.: Configuring Versioned Software Products. in [37]

10. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. Technischer Bericht AIB 96-10, RWTH Aachen, Oktober, 1996
11. Dittrich, K.R., Lorie, R.A.: Version Support for Engineering Database Systems. IEEE Transactions on Software Engineering, Vol. 14, No. 4, 1988, S. 429-437
12. Date, C., Darwen, H.: A Guide to the SQL Standard - A user's guide to the standard relational language SQL (3rd Edition). Boston, Mass.: Addison-Wesley 1993
13. Estublier, J., Casallas, R.: The Adele configuration manager. in [38], S. 99-134
14. Goldstein Golendziner, L., Saraiva dos Santos, C.: Versions and configurations in object-oriented database systems: a uniform treatment. Proc. 7th. Int. Conf. on Management of Data, COMAD'95, Pune, India, Dezember, 1995
15. Katz, R.H.: Toward a Unified Framework for Version Modeling in Engineering Databases. ACM Computing Surveys, Vol. 22, No. 4, 1990, S. 375-408
16. Katz, R.H., Anwarrudin, M., Chang, E.: A Version Server for Computer-Aided Design Data. Proc. ACM/IEEE 23rd Int. Conf. on Design Automation, Las Vegas, 1986, S. 27-33.
17. Katz, R.H., Chang, E., Bhateja, R.: Version Modeling Concepts for Computer-Aided Design Databases. ACM SIGMOD Record, Vol. 15, No. 2, 1986, S. 379-386.
18. Käfer, W.: Geschichts- und Versionsmodellierung komplexer Objekte - Anforderungen und Realisierungsmöglichkeiten am Beispiel des NDBS PRIMA. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, 1992
19. Kim, W., Bertino, E., Garza, J.F.: Composite objects revisited. Proc. ACM SIGMOD Conference, Oregon, 1989, S. 337-347
20. Käfer, W., Mitschang, B.: Flexible Entwurfsdatenverwaltung für CAD-Frameworks: Konzept, Realisierung und Bewertung. Proc. der GI-Fachtagung 'Datenbanksysteme in Büro, Technik und Wissenschaft', Braunschweig, 1993, S. 144-163
21. Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model. Proc. 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992, S. 348-357
22. Klahold, P., Schlageter, G., Wilkes, W.: A General Model for Version Management in Databases. Proc. 12th Conf. on Very Large Databases, VLDB'86, Kyoto, Japan, 1986, S. 319-327.
23. Lamb, C.W., Landis, G., Orenstein, J.A., Weinreb, D.L.: ObjectStore. Communications of the ACM, Vol. 34, No. 10, 1991, S. 51-63
24. Landis, G.S.: Design evolution and history in an object-oriented CAD/CAM database. Proc. 31st COMPCON Conf., San Francisco, CA, März, 1986, S. 297-327
25. Lavency, P., Vanhoedanaghe, M.: Knowledge based configuration management. Proc. 21st Hawaii Int. Conf. on System Sciences, 1988, S. 83-92
26. Leblang, D.: The CM challenge: Configuration management that works. in [38], S1-38
27. McLeod, D., Narayanawamy, K., Bapa Rao, K.: An approach to information management for CAD/VLST applications. Proc. SIGMOD Conference on Databases for Engineering Applications, San Jose, CA, 1983, S. 39-50
28. Munch, B.P., Larsen, J.-O., Gulla, B., Conradi, R., Karlsson, E.-A.: Uniform versioning: The change-oriented model. Proc. 4th Int. Workshop on Software Configuration Management, Baltimore, Maryland, Mai, 1993
29. Nagl, M. (Hrsg.): Building Tightly-Integrated Software Development Environments: The IPSEN Approach. LNCS 1170, Springer, Berlin, 1996
30. Nink, U., Ritter, N.: A Database Application Programming Interface vor Complex-Object Versions. Proc. der GI-Fachtagung 'Datenbanksysteme in Büro, Technik und Wissenschaft', Ulm, 1997
31. Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach. Proc. of the 10th Int. Conf. on Data Engineering, Houston, Texas, 1994, S. 440-451.
32. Ritter, N., Mitschang, B., Härder, T.: Conflict Management in CONCORD. Proc. 6th Int. Conf. on Data and Knowledge Bases for Manufacturing and Engineering, Tempe, Arizona, Oktober, 1996
33. Ritter, N., Mitschang, B., Härder, T., Nink, U.: Unterstützung der Ablaufsteuerung in Entwurfsumgebungen durch Versionierung und Konfigurierung. Proc. STAK'94 (Softwaretechnik in Automatisierung und Kommunikation - Datenbanken unter Realzeit- und technischen Entwicklungsanforderungen), Ilmenau, März 1994, S.135-159

34. Sciore, E.: Versioning and Configuration Management in an Object-Oriented Model. VLDB Journal 3, 1994, S. 77-106
35. Siepmann, E.: Entwurfstheorie und Entwurfsdatenmodellierung für CAD-Frameworks. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, 1991
36. Schwartz, J., Westfechtel, B.: Konfigurationsverwaltung in einer heterogenen CIM-Umgebung. Proc. STAK'94 (Softwaretechnik in Automatisierung und Kommunikation - Datenbanken unter Realzeit- und technischen Entwicklungsanforderungen), Ilmenau, März 1994, pp. 179-197
37. Sommerville, I. (Hrsg.): Proc. 6th Int. Workshop on Software Configuration Management. LNCS 1167, Berlin: Springer 1996
38. Tichy, W. (Hrsg.): Configuration Management. New York: John Wiley and Sons 1994
39. Tichy, W.: RCS - A system for version control. Software Practice and Experience, 15(7):637-654, Juli, 1985
40. Wilkes, W.: Der Versionsbegriff und seine Modellierung in CAD/CAM-Datenbanken. Dissertation, Fern Universität / Gesamthochschule Hagen, 1987
41. Wilkes, W., Kemper, F.: Einsatz indirekter Komponenten zur Versionsverwaltung in Entwurfsdatenbanken. Proc. STAK'94 (Softwaretechnik in Automatisierung und Kommunikation - Datenbanken unter Realzeit- und technischen Entwicklungsanforderungen), Ilmenau, März 1994, pp. 161-177
42. Winkler, J.F.H.: Version control in families of large programs. Proc. 9th Int. Conf. on Software Engineering and Configuration Control, Stuttgart, 1988
43. Wedekind, H., Müller, T.: Stücklistenorganisation bei großen Variantenzahlen. Angewandte Informatik, Heft 9, Sept. 1981, S. 377-383
44. van der Wolf, P.: CAD Frameworks - Principles and Architecture. Boston: Kluwer Academic 1994
45. Zeller, A., Snelting, G.: Handling version sets through feature logic. Proc. 5th European Software Engineering Conf., LNCS 989, S. 191-204, Berlin: Springer 1995
46. Zimmermann, G.: PLAYOUT - A Hierarchical Layout System, Proc. 18. GI Konferenz, Hamburg, Informatik-Fachberichte 188, S.31-51, Berlin: Springer 1988