ECONSTOR
Make Your Publications Visible.

A Service of

ZBW Leibniz-Informationszentrum Wirtschaft
Leibniz Information Centre for Economics

Meckenstock, Axel; Unland, Rainer; Zimmer, Detlef

Working Paper

Flexible Unterstützung kooperativer Entwurfsumgebungen durch einen Transaktions-Baukasten

Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 25

Provided in Cooperation with:
University of Münster, Department of Information Systems

Suggested Citation: Meckenstock, Axel; Unland, Rainer; Zimmer, Detlef (1993) : Flexible Unterstützung kooperativer Entwurfsumgebungen durch einen Transaktions-Baukasten, Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 25, Westfälische Wilhelms-Universität Münster, Institut für Wirtschaftsinformatik, Münster

This Version is available at:
https://hdl.handle.net/10419/59339

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.

WWW.ECONSTOR.EU

Mitglied der Leibniz-Gemeinschaft

Inhalt

1 Einleitung 3
2 Der Transaktions-Baukasten 4
3 Konzepte für Entwurfsumgebungen 7
4 Unterstützung von Entwurfstransaktionen 9
4.1 Objekte und Objektversionen 9
4.2 Rechte für Zugriffe auf Versionen 10
4.3 Rechte für Kooperation 11
4.4 Rechte für **commit** und **abort** 12
4.5 Rechte für Abhängigkeiten 13
4.6 Beispiel 13
4.7 Rechte für Konsistenzzustände 15
4.8 Zusammenfassung der Rechte 15
5 Definition von Protokollen 16
6 Beispiel für eine DT-Hierarchie 18
7 Vergleich mit anderen Ansätzen 20
8 Zusammenfassung und Ausblick 20
Literatur 21

Zusammenfassung

Entwurfsumgebungen stellen besondere Anforderungen an das Transaktionsmanagement, so z.B. bei der Realisierung von Kooperations- und Recoverymechanismen. Hierbei ergibt sich ein Zielkonflikt: Einerseits soll das System größtmögliche Sicherheit garantieren, andererseits sollen Flexibilität und Freiheit der Entwickler nicht eingeschränkt werden. Dieser Konflikt kann durch konfigurierbare Systeme gelöst werden, die die Definition unterschiedlicher Typen von Transaktionen und damit die Anpassung an die Anforderungen der jeweiligen Umgebung erlauben. In diesem Papier werden auf der Basis eines Baukasten-Konzeptes Mechanismen zur Unterstützung von Kooperation und Recovery für Entwurfstransaktionen vorgestellt. Der zentrale Gedanke ist, daß Transaktionen bestimmte Rechte erwerben können und die Vergabe dieser Rechte durch Protokolle kontrolliert wird.

1 Einleitung

Entwurfsumgebungen [1], z.B. für CAD (*Computer Aided Design*), CASE (*Computer Aided Software Engineering*) oder CACE (*Computer Aided Concurrent Engineering*), erfordern ein Transaktionsmanagement, das sich wesentlich vom konventionellen Transaktionsmanagement [2, 3] unterscheidet. Entwurfstransaktionen sind typischerweise von langer Dauer, bearbeiten meist interaktiv komplexe Objekte und erfordern flexible, kooperative Synchronisationsprotokolle. Konventionelle Algorithmen, die auf den sog. ACID-Eigenschaften [4] basieren, sind nur begrenzt einsetzbar.

Ein zentrales Problem ist die in o.g. Umgebungen notwendige *Kooperation* zwischen Entwicklern, da diese der Isolationseigenschaft von Transaktionen entgegensteht. Die Kooperation verkompliziert die *Recovery*, die in kooperativen Umgebungen nicht mehr isoliert für einzelne Transaktionen und auf der Basis der "Alles-oder-Nichts-Eigenschaft" durchgeführt werden kann.

Die generelle Problematik von Transaktionen in Entwurfsumgebungen ergibt sich aus einem Zielkonflikt: Einerseits soll das System größtmögliche Sicherheit garantieren, z.B. durch Verhinderung von "inkonsistenten" Zuständen. Andererseits wollen die Entwickler größtmögliche Freiheiten haben, z.B. bei der Kooperation untereinander oder beim Zugriff auf gemeinsame Daten. Konventionelle Algorithmen (wie das strikte Zwei-Phasen-Sperrverfahren) bieten zwar Sicherheit, schränken aber die Freiheiten erheblich ein. Eine Lösung des Zielkonfliktes besteht darin, *konfigurierbare Systeme* zur Verfügung zu stellen, innerhalb derer Transaktionen mit den Graden an Flexibilität und Sicherheit definiert werden können, die in der jeweiligen Umgebung erforderlich sind. Ein solcher Ansatz, ein *Transaktions-Baukasten*, wurde in [5] vorgestellt. Der Baukasten erlaubt die Definition von Transaktionstypen, denen unterschiedliche Merkmale (z.B. Synchronisationsprotokolle) zugeordnet werden können. Transaktionen verschiedener Typen können zu einer heterogenen Transaktionshierarchie zusammengesetzt werden, die die Anforderungen der jeweiligen Anwendung widerspiegelt.

In diesem Papier sollen durch Erweiterung des Baukastens Mechanismen für die Unterstützung von Kooperation und Recovery bereitgestellt werden. Hierzu werden *Rechte* eingeführt, die von Transaktionen erworben werden müssen, um entsprechende Operationen auszuführen. Die Transaktionstypen kontrollieren die Vergabe von Rechten durch die Definition von *Protokollen*. Dadurch können Transaktionshierarchien definiert werden, die auf die Anforderungen der jeweiligen Entwurfsumgebung zugeschnitten sind.

Für Nichtstandard-Anwendungen existieren bereits eine Reihe von neueren Transaktionsmodellen [6]. Ziel des hier beschriebenen Ansatzes ist es, Basismechanismen zur Verfügung zu stellen, die die Definition unterschiedlicher Transaktionen bzw. Transaktionsmodelle

(auch in kombinierter Form) erlauben.

Kapitel 2 stellt die Grundideen des Baukastens vor. In Kapitel 3 werden wichtige Anforderungen an Transaktionen in Entwurfsumgebungen und Realisierungsalternativen aufgezeigt. Kapitel 4 diskutiert die Verwendung von Rechten zur Unterstützung von Entwurfstransaktionen. Auf dieser Basis beschreibt Kapitel 5 die Definition von Protokollen, gefolgt von einem Beispiel in Kapitel 6. Ein Vergleich mit einigen Konzepten aus der Literatur findet in Kapitel 7 statt. Abschließend folgt eine kurze Zusammenfassung mit Ausblick.

2 Der Transaktions-Baukasten

Der Transaktions-Baukasten [5] erlaubt die Definition von Transaktionshierarchien mit Baumstruktur. Jede Transaktion ist Instanz eines bestimmten **Transaktionstyps**, der die Eigenschaften und das Verhalten der Transaktion beschreibt. Ein Transaktionstyp definiert das verwendete Synchronisationsprotokoll (optimistisch, strikt zweiphasig etc.), die zulässigen Operationen, das Verhalten im Recoveryfall etc. Jede Transaktion besitzt einen **Objektpool**, der alle der Transaktion zugeordneten Objekte (logisch) enthält (Abb. 1). Eine Transaktion holt sich ein Objekt durch *check_out* vom Pool der Eltern-Transaktion in ihren eigenen Pool und gibt es durch *check_in* wieder zurück.

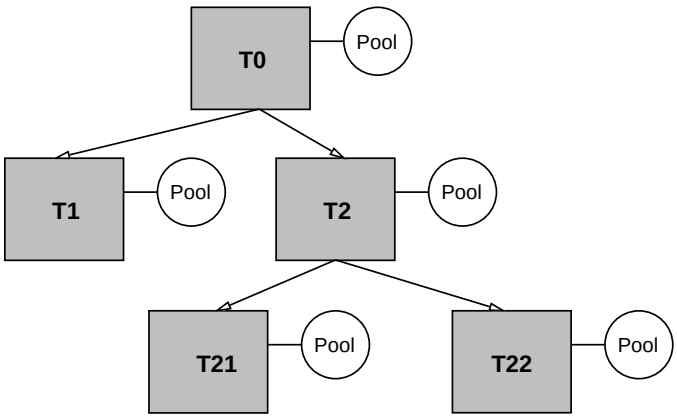

Abb. 1: Transaktions-Hierarchie

Für jeden Pool kann die Synchronisation individuell definiert werden. Hierzu legt eine Transaktion (z.B. T2 in Abb. 1) fest, welche Anforderungen ihre Kind-Transaktionen (T21 und T22) beim Zugriff auf den Pool der Transaktion (T2) beachten müssen (z.B. Zweiphasigkeit). Weiterhin definiert eine Transaktion (z.B. T2) das Protokoll, das sie selbst zum Zugriff auf den Eltern-Pool (T0) verwendet. Hiermit ergibt sich eine sog. **zweistufige Kontrollsphäre**, d.h. eine Transaktion kontrolliert ihr eigenes Verhalten sowie das Verhalten ihrer Kind-Transaktionen. Diese können dann für ihre eigenen Pools wieder ein anderes Verhalten definieren.

Damit eine derartige heterogene Struktur korrekt funktioniert, wird das Konzept des **schrittweisen Objekttransfers** eingeführt. Eine Transaktion kann ein Objekt nur von ihrer Eltern-Transaktion anfordern (*check_out*). Befindet sich das Objekt nicht im Pool der Eltern-Transaktion, so setzt sich dieser Prozeß fort, bis ggfs. die Wurzel-Transaktion erreicht ist, die Zugriff auf alle Objekte der Datenbank hat. Analog findet auch die Freigabe von Objekten (*check_in*) stufenweise statt. Schrittweiser Transfer ist notwendig, damit jede beteiligte Transaktion die Einhaltung ihres Protokolls (z.B. Zweiphasigkeit) sicherstellen kann. Man spricht hier auch von **downward inheritance**, da Objekte auf dem Weg von "oben" nach "unten" in die beteiligten Pools (logisch) eingefügt werden. Andere Ansätze (z.B. [7]) verwenden **upward inheritance**, d.h. Objekte bzw. Sperren werden beim *commit* einer Kind-Transaktion an die Eltern-Transaktion vererbt. Ein Beispiel zeigt, daß *upward inheritance* nicht immer korrekt ist: Würde T2 sich in der Schrumpfungsphase (zweiphasiges Sperrprotokoll) befinden und T21 ein direktes *check_out* für ein Objekt von T0 durchführen, so könnte dieses Objekt nicht mehr von T21 an T2 vererbt werden, da dies der Zweiphasigkeit widersprechen würde.

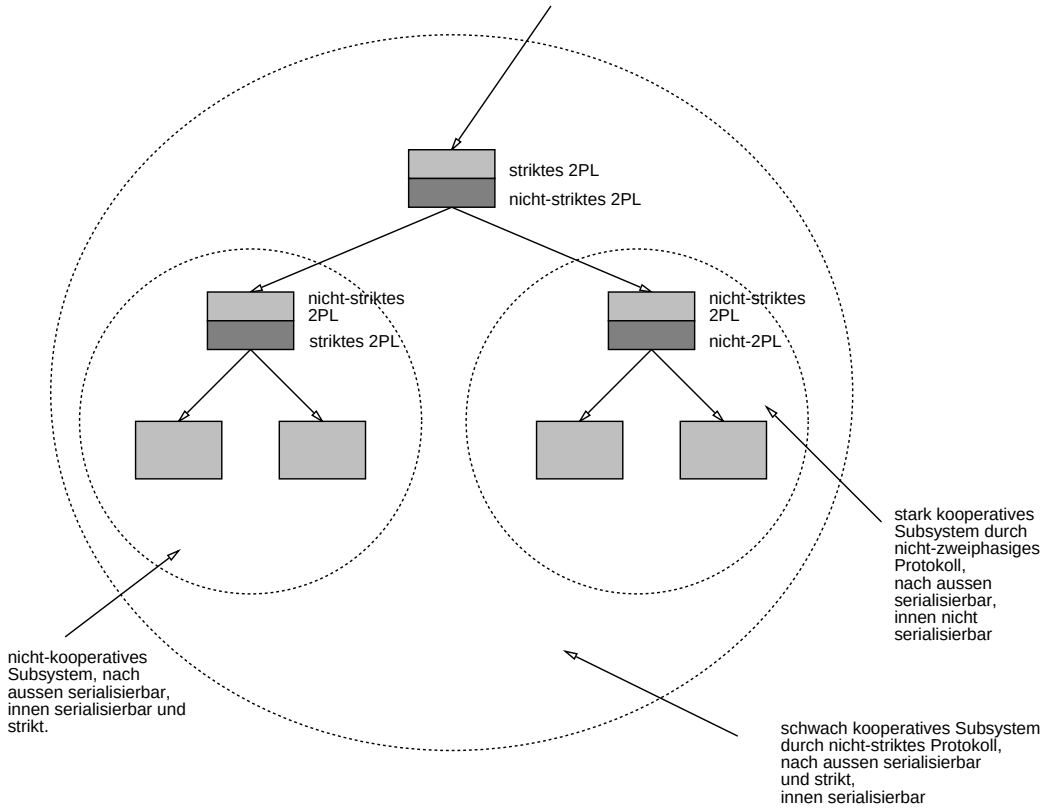

Abb. 2: Beispiel für Subsysteme mit unterschiedlichen Protokollen

Der schrittweise Transfer erlaubt es, Transaktionen unterschiedlicher Transaktionstypen beliebig zu kombinieren, sofern die Eigenschaften der Kind-Transaktionen mit den Anforderungen der Eltern-Transaktion kompatibel sind. Hierdurch läßt sich eine heterogene Transaktionshierarchie konstruieren. Innerhalb der Hierarchie entstehen Subhierarchien mit definierten Eigenschaften (Abb. 2). Beispielsweise kann durch ein striktes Zweipha-

senprotokoll erreicht werden, daß eine Subhierarchie nach außen das Kriterium der Serialisierbarkeit erfüllt und keine Daten vorzeitig freigibt, während innerhalb der Subhierarchie kooperative Protokolle (z.B. nicht-zweiphasige Protokolle) möglich sind. Hierdurch kann den unterschiedlichen Anforderungen innerhalb einer Projekthierarchie Rechnung getragen werden, wo meist innerhalb von Projekten kooperativere Protokolle benötigt werden als über Projektgrenzen hinweg.

Für die Realisierung von Sperrprotokollen definiert der Baukasten flexible **Sperren**. Sperren werden auf Objekte gesetzt, die ggfs. versioniert sind. Eine Sperre hat zwei Wirkungen: die **Innenwirkung**, die angibt, welche Rechte die sperrende Transaktion auf einem Objekt hat, und die **Außenwirkung**, die angibt, welche Rechte konkurrierende Transaktionen maximal besitzen bzw. noch erwerben können. Als mögliche **Sperrmodi** werden unterschieden:

- *X-Modus* (*exclusive*)
  erlaubt Lesen, Schreiben und Löschen eines Objektes sowie bei versionierten Objekten die Ableitung einer neuen Version.
- *U-Modus* (*update*)
  erlaubt Lesen und Schreiben eines Objektes sowie bei versionierten Objekten die Ableitung einer neuen Version.
- *D-Modus* (*derivation*)
  erlaubt Lesen eines Objektes sowie bei versionierten Objekten die Ableitung einer neuen Version.
- *S-Modus* (*shared*)
  erlaubt Lesen eines Objektes.
- *B-Modus* (*browse*)
  erlaubt Lesen eines Objektes unabhängig von anderen für das Objekt gesetzten Sperren (sog. *dirty read*).

Eine **Sperre** ist definiert als ein Tupel *(Innenwirkung / Außenwirkung)*. Beispielsweise erlaubt eine Sperre mit Innenwirkung U und Außenwirkung S (Notation: U/S) einer Transaktion das Ändern eines Objektes, während konkurrierende Transaktionen das Objekt nur lesen dürfen. Durch die Unterscheidung zwischen Innen- und Außenwirkung lassen sich Sperren flexibler definieren als bei klassischen Sperrmodi, bei denen die Außenwirkung nur implizit beschrieben wird.

Zusätzlich zu den **transaktionsbezogenen** (TR-) **Sperren** definiert der Baukasten **objektbezogene** (OR-) und **subjektbezogene** (SR-) **Sperren**. OR-Sperren können dauerhaft, d.h. unabhängig von Transaktionen, auf ein Objekt gesetzt werden und damit

gewisse Operationen auf dem Objekt ausschließen (z.B. bei Bibliotheks-Objekten, die nur gelesen werden dürfen). Sie schränken also die Wirkung von TR-Sperren ein. Über SR-Sperren können Objekte temporär an Subjekte (z.B. Benutzer oder Gruppen) gebunden werden. Damit erlauben SR-Sperren als übergeordnetes Konzept z.B. die gezielte Weitergabe von Objekten an andere Transaktionen.

Kooperation kann im Baukasten über zwei Mechanismen erreicht werden. Einerseits können Sperrprotokolle verwendet werden, die die Freigabe von Objekten vor Ende einer Transaktion erlauben (z.B. nicht-strikte oder nicht-zweiphasige Protokolle). Andererseits können Sperren eingesetzt werden, die nicht-isoliertes Arbeiten zulassen (z.B. U/U, also paralleles Modifizieren), wobei später i.d.R. ein automatisches oder manuelles Zusammenführen paralleler Modifikationen (s. z.B. [8]) notwendig ist. In jedem Fall muß Kooperation der Transaktionshierarchie (schrittweiser Transfer) folgen, um die Einhaltung der Protokolle aller auf dem Pfad liegenden Transaktionen sicherzustellen.

Da die Außenwirkung für *alle* anderen Transaktionen gilt, kann nicht spezifiziert werden, *welche* Transaktionen als Kooperationspartner in Frage kommen. In [5] wird vorgeschlagen, hierzu die subjektorientierten Sperren zu verwenden, über die ein Subjekt (z.B. ein Benutzer) eine transaktionsorientierte Sperre direkt zu einer anderen Transaktion transferieren kann. Alternativ kann bei der Definition der Außenwirkung eine Differenzierung vorgenommen werden [9].

3 Konzepte für Entwurfsumgebungen

Während der Baukasten [5] für verschiedenartige Anwendungsgebiete konzipiert wurde, soll in diesem Papier besonders auf die Anforderungen von Entwurfsumgebungen eingegangen werden. Dabei soll das Baukastenprinzip konsequent weiterverfolgt werden, d.h. die zusätzlichen Anforderungen sollen durch Kombination geeigneter Basismechanismen realisiert werden können. Es geht also darum, den Baukasten um Primitive zur Definition von Transaktionstypen zu erweitern, die den speziellen Anforderungen von Entwurfsumgebungen gerecht werden.

Im folgenden sollen ohne Anspruch auf Vollständigkeit wichtige Anforderungen an Entwurfstransaktionen (**Design Transactions**, **DTs**) aufgeführt werden.

- **Kooperatives Arbeiten** muß unterstützt werden. Hierzu sollten typische Kooperationsformen wie das Verleihen von Objekten an eine andere DT mit späterer Rückgabe modelliert werden können. Weiterhin sollten parallele Änderungen von Objekten möglich sein, wobei spezifiziert werden kann, welche DTs letztendlich für die Integration der Änderungen verantwortlich sind.

- Beim Anfordern und Freigeben von Objekten sollten die **Konsistenzzustände** der Objekte berücksichtigt werden. Im konventionellen Transaktionsmodell werden lediglich die Zustände "committed" und "uncommitted" unterschieden. In Entwurfsumgebungen durchlaufen Objekte jedoch unterschiedliche Konsistenzstufen (z.B. ungetestete vs. getestete Objekte). Es ist wichtig, daß DTs Objekte nur in bestimmten Konsistenzzuständen freigeben können und bei der Anforderung von Objekten erkennen können, in welchem Zustand sich diese befinden, um den Zugriff auf inkonsistente Daten zu verhindern.

- Die **Recovery** von DTs läßt sich nicht auf der Basis der "Alles-oder-Nichts"-Eigenschaft durchführen, da Abbrüche umfangreicher Entwurfsaktivitäten unakzeptabel sind. Durch Kooperation treten außerdem Abhängigkeiten zwischen DTs auf, die zu kaskadierenden Abbrüchen führen können. Es ist deshalb notwendig, eine Kontrolle über die Entwicklung von Abhängigkeiten und über die Kooperation zwischen DTs zu bekommen. Außerdem muß es möglich sein, das *commit* und *abort* der an einer Kooperation beteiligten DTs kontrollieren zu können.

- **Versionierung** von Objekten sollte berücksichtigt werden, da diese im Entwurfsbereich eine wichtige Rolle spielt.

Die in Kapitel 2 beschriebenen Konzepte reichen für die Definition von Protokollen mit diesen Anforderungen nicht aus. Um beispielsweise ein Objekt zu verleihen, muß ein *check_in* durchgeführt werden. Damit ist der Vorgang des Verleihens jedoch nicht mehr von einem regulären *check_in* unterscheidbar.

Bei der Betrachtung der beschriebenen Sperrmechanismen stellt man fest, daß sie im wesentlichen auf der Vergabe von **Rechten** basieren: Eine DT fordert Rechte an, um bestimmte Operationen durchzuführen, und definiert die maximalen Rechte konkurrierender DTs. Es erscheint naheliegend, diesen Mechanismus zu erweitern und flexibler zu gestalten, um die zusätzlich geforderte Semantik zu realisieren.

Auf der Basis von Rechten können dann **Protokolle** spezifiziert werden, die für einen DT-Typ festlegen, welche Rechte er in welcher Weise anfordern oder freigeben darf. Auf der Basis von Protokollen (und weiteren Eigenschaften) lassen sich unterschiedliche DT-Typen definieren, die zur Konstruktion einer DT-Hierarchie eingesetzt werden können.

Für die Realisierung von Rechten bieten sich zwei Alternativen an. Eine Möglichkeit ist die Verwendung von Sperren, die eine Innen- und Außenwirkung besitzen (Kapitel 2). Eine andere Möglichkeit ist der Einsatz von Triggern oder Regeln[1] [10]. Im folgenden soll keine spezielle Realisierungsform beschrieben werden (dies ist einem späteren Papier vor-

[1]Sperren lassen sich auch als spezielle Form von Regeln auffassen, mit denen u.a. die Kompatibilität von konkurrierenden Anforderungen festgelegt wird.

behalten). Stattdessen sollen Rechte als abstrakter Basismechanismus zur Unterstützung von DTs eingeführt werden.

Es sei darauf hingewiesen, daß es zunächst um die Bereitstellung von Basismechanismen als Elementen des Baukastens geht. Mithilfe dieser Elemente können dann komplexe Vorgänge (z.B. Kooperation) realisiert werden. An der Benutzerschnittstelle sollte man anwendungsorientierte Operationen anbieten, die die Rechtevergabe, die Synchronisation und den Aufruf transaktionsbezogener Operationen transparent für den Benutzer ausführen.

Bei den folgenden Ausführungen wird ausschließlich auf Entwurfstransaktionen (DTs) eingegangen. DTs bestehen aus Entwurfsoperationen und systemdefinierten Operationen (*check_out*, *check_in* etc.). Über Art und Struktur der Objekte werden keine Annahmen gemacht. Die i.d.R. vorhandenen vielfältigen Abhängigkeiten zwischen Objekten (z.B. überlappende oder geschachtelte Objekte) werden bei den folgenden Betrachtungen nicht berücksichtigt.

4 Unterstützung von Entwurfstransaktionen

4.1 Objekte und Objektversionen

Da Objekte in Entwurfsumgebungen oft versioniert werden, soll Versionierung in die folgenden Betrachtungen einbezogen werden. Ein **Objekt** soll als eine Menge von **Versionen** aufgefaßt werden. Zwischen Objektpools werden ausschließlich Versionen transferiert (durch *check_in* bzw. *check_out*). Die Hierarchie von DTs bzw. Objektpools führt dazu, daß die Versionen eines Objektes in verschiedenen Pools liegen können und keinen geschlossenen Versionsgraphen bilden. Man kann zwei Arten von Versionierung unterscheiden (Abb. 3):

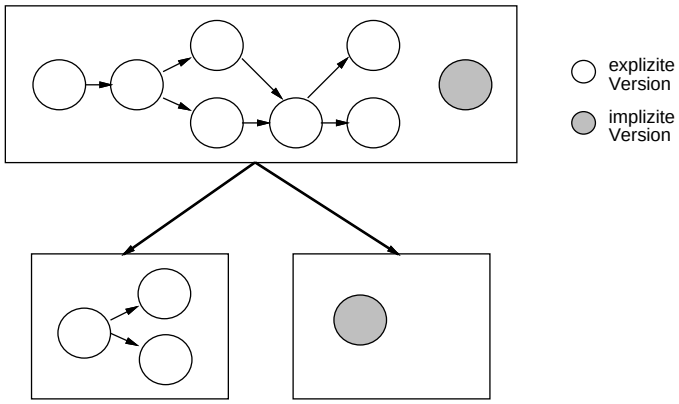

Abb. 3: Beispiel für ein Objekt als Menge von Versionen

- **explizite Versionierung**
  Für jeden Pool kann als DT-Typ-Eigenschaft festgelegt werden, ob und wie eine Versionierung, d.h. die Bildung eines Versionsgraphen erfolgen soll. Es ist sinnvoll, die Versionierung in den Pools unabhängig voneinander durchzuführen, da meist nur relevante Versionen zwischen Pools transferiert werden, während Zwischenversionen nur lokal benötigt werden.

- **implizite Versionierung**
  Wenn Versionen zwischen Objektpools (logisch) transferiert werden (*check_out* bzw. *check_in*), so entstehen implizite, oft temporäre Versionen. Diese Versionen sind ggfs. isoliert, d.h. sie werden nicht unbedingt in den Versionsgraphen aufgenommen.

Wenn im folgenden von *Versionen* gesprochen wird, so sind explizit oder implizit erzeugte Versionen gemeint.

4.2 Rechte für Zugriffe auf Versionen

Für Zugriffe auf Versionen werden folgende elementare Rechte definiert:

- *browse*: DT darf die Version lesen ohne Berücksichtigung von Aktivitäten anderer DTs (*dirty-read*).
- *shared*: DT darf die Version lesen.
- *update*: DT darf den Inhalt der Version modifizieren.
- *delete*: DT darf die Version löschen.
- *derive*: DT darf von der Version eine neue Version ableiten.

Zwischen diesen Rechten besteht keine Inklusionsbeziehung. Die Rechte können auch kombiniert werden, wobei nicht alle Kombinationen sinnvoll sind. Die in Kap. 2 genannten Sperrmodi lassen sich als vordefinierte Kombinationen von Rechten auffassen. Beispielsweise gewährt eine Sperre mit Innenwirkung X die Rechte *shared*, *update*, *delete* und *derive*.

Abb. 4 zeigt ein Beispiel für die Verwendung von Rechten auf Versionen. DT11 und DT12 führen beide ein *check_out* der Version v von DT1 zum Schreiben durch. Demzufolge erhalten beide eine lokale Version (v1 bzw. v2), jeweils mit *update*-Recht. Wenn nur DT11 für die Version v das *derive*-Recht erhält, so ist DT11 verantwortlich für die Integration der parallelen Modifikationen. DT12 kann in diesem Fall zwar ein *check_in* ihrer lokalen Version durchführen, kann diese jedoch nicht von der Version v ableiten.

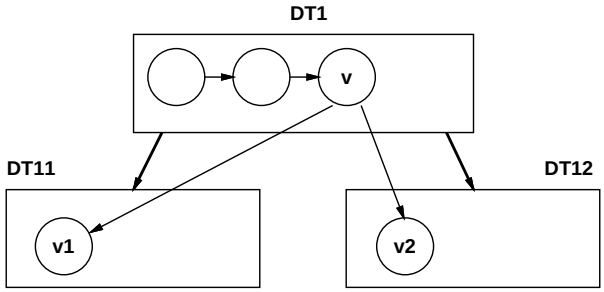

Abb. 4: Beispiel für Rechte auf Versionen

Die Rechte auf den Versionen im eigenen Pool einer DT können durchaus umfangreicher sein als die Rechte auf Versionen im Eltern-Pool. Dies ist sinnvoll, wenn lokale Experimente ausgeführt werden sollen, die nicht in den Eltern-Pool eingebracht werden, oder wenn die Modifikationen durch Kooperation zu einer anderen DT übertragen werden sollen.

Die explizite Einbeziehung von Versionen in das Modell erlaubt also eine sehr differenzierte Definition von Rechten. Dies spielt auch eine wichtige Rolle bei Protokollen zur Kooperation zwischen DTs.

4.3 Rechte für Kooperation

Zur Realisierung von Kooperation sollen nun zusätzliche Rechte eingeführt werden, mit denen der Vorgang des Verleihens realisiert werden kann:

- *return*: DT darf die verliehene Version zurückerhalten.
- *borrow*: DT darf die Version entleihen.

Um eine Version zu verleihen, muß eine DT ein *check_in* in den Eltern-Pool durchführen. Dies kann auf zwei Arten erfolgen, als reguläres *check_in*, bei dem eine explizite Version im Versionsgraphen erzeugt wird (durch Überschreiben oder Ableiten), oder durch (logisches) Erzeugen einer impliziten, isolierten Version, die nur für die Kooperation verwendet wird. Letzteres erscheint sinnvoller, da die ausgetauschte Version typischerweise noch inkonsistent ist und nicht publiziert werden soll.

Der Vorgang des Ausleihens läuft dann vereinfacht dargestellt folgendermaßen ab (Abb. 5; die einzelnen Schritte werden in Abschnitt 4.6 genauer erläutert): Die DT (hier DT11), die eine Version (v1) verleihen möchte, führt ein *check_in* in den Eltern-Pool durch (v2), setzt hierauf das *return*-Recht und erlaubt anderen DTs (ggfs. nur bestimmten) das Entleihen (*borrow*-Recht). Außerdem wird über die Rechte spezifiziert, welche Operationen andere DTs auf der ausgeliehenen Version durchführen dürfen (z.B. *update*). Ggfs. werden die

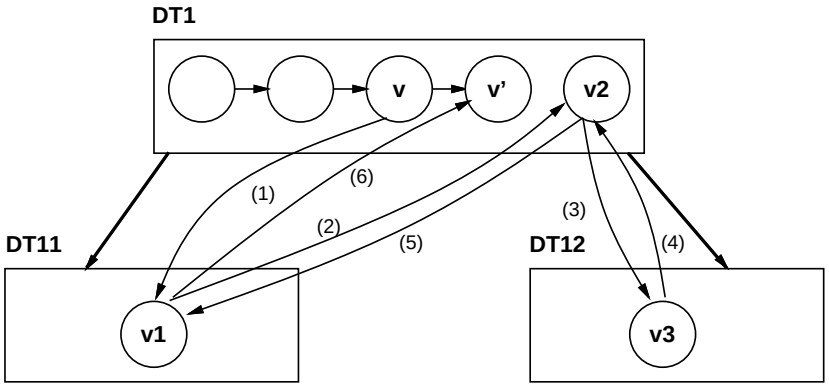

Abb. 5: Beispiel für Kooperation durch Ausleihen

Rechte auf der Version v1 reduziert, damit DT11 während des Verleihens keine Modifikationen vornehmen kann. Eine DT (hier DT12), die die Version entleihen möchte, benötigt nun das *borrow*-Recht und führt ein *check_out* in ihren eigenen Pool durch (v3). Die Rückgabe der Version erfolgt, indem DT12 ein *check_in* und DT11 ein *check_out* ausführt. Die beschriebenen Vorgänge können auch über mehrere Ebenen ablaufen (schrittweiser Transfer).

Die intuitive Lösung, die Version direkt von DT11 zu DT12 zu übergeben, widerspricht dem Konzept des schrittweisen Transfers. Hierdurch könnte das Protokoll, das eine DT auf dem Pfad zwischen den kooperierenden DTs definiert, umgangen werden. In diesem Beispiel könnte DT1 das Verleihen verbieten.

4.4 Rechte für commit und abort

Im konventionellen Transaktionsmodell kann eine Transaktion ein Objekt modifizieren und anschließend freigeben (bei nicht-strikten Sperrprotokollen). Dennoch behält sie bis zum Transaktionsende *implizit* das Recht, ein *commit* oder *abort* durchzuführen, wobei ein *abort* zum kaskadierenden Abbruch von Transaktionen führt, die auf das Objekt zugegriffen haben. Das Recht soll hier *explizit* eingeführt werden:

- *decide*: DT darf über *commit* oder *abort* ihrer Modifikationen entscheiden.

Das oben beschriebene Verhalten läßt sich nun modellieren, indem eine DT beim schreibenden Zugriff auf eine Version das *decide*-Recht bekommt und erst bei Ende der DT freigibt. Eine DT kann jedoch auch das *decide*-Recht auf bestimmten geänderten Versionen vorzeitig aufgeben. Dies ist gleichbedeutend mit der Durchführung eines *commit* nur für einzelne Versionen (nicht für die gesamte DT). Andere DTs können hierdurch erkennen, daß die Änderungen schon stabil sind und nicht mehr von einem *abort* betroffen sein können. Das *decide*-Recht beschreibt also das Verhalten bei der Recovery und bietet

damit die Möglichkeit, die Gefahr eines kaskadierenden Abbruchs zu erkennen und/oder zu vermeiden.

Folgendes Beispiel demonstriert die Verwendung des *decide*-Rechts: Eine DT führt ein *check_out* mit den Rechten *update* und *decide* aus. Nach dem *check_in* ihrer Änderungen gibt sie das *update*-Recht auf. Behält sie das *decide*-Recht, so ist eine spätere Rücknahme ihrer Änderungen möglich, gibt sie es auf, so sind die Änderungen im Eltern-Pool stabil.

4.5 Rechte für Abhängigkeiten

Wenn eine DT auf eine Version mit *decide*-Recht zugreift, so macht sie sich abhängig von der DT, die das *decide*-Recht besitzt, da diese die Version noch invalidieren kann. Sie ist also ggfs. von einem *cascading abort* betroffen. In manchen Fällen ist dies jedoch nicht erwünscht, z.B. wenn eine Version nur zu Informationszwecken gelesen werden soll und nicht in weitere Berechnungen eingeht. Um diesen Fall zu modellieren, soll ein weiteres Recht eingeführt werden:

- *independent* : DT darf auf die Version zugreifen, ohne daß eine Abhängigkeit zu einer anderen DT entsteht.

Die Verwendung des *independent*-Rechtes gibt also einer DT die Möglichkeit, instabile Daten zu verarbeiten, ohne daß dies im Falle eines *abort* Auswirkungen hat.

Eine DT, die eine Version lesen möchte, ohne sie für weitere Berechnungen einzusetzen, kann die Rechte *shared* und *independent* verwenden. Damit kann sie vermeiden, daß sie von einem *abort* betroffen ist. Die Verwendung dieser beiden Rechte ist *nicht* identisch mit der Verwendung des *browse*-Rechts, da letzteres die Aktivitäten anderer DTs nicht berücksichtigt.

4.6 Beispiel

Anhand des in Abb. 5, Abschnitt 4.3 skizzierten Beispiels soll der Kooperationsvorgang schrittweise (die Nummern der Schritte sind eingezeichnet) verdeutlicht werden. Dabei werden aus Gründen der Übersichtlichkeit nur die wichtigsten Rechte aufgeführt. Es wird die Notation `(Innenwirkung) / (Außenwirkung)` verwendet. Eine leere Klammer bedeutet, daß keine Rechte bestehen. Ein “all” bedeutet, daß alle Rechte bestehen.

(1) DT11 führt *check_out* von Version v zum Ändern aus, wobei eine lokale Version v1 entsteht. DT11 setzt folgende Rechte:

```
DT11:Version v  : (derive, decide) / (shared)
DT11:Version v1 : (update, decide) / ()
```

(2) DT11 verleiht die lokale Version v1 zum Schreiben durch *check_in* in den Pool von DT1. Hierfür wird eine temporäre, isolierte Version v2 erzeugt.

```
DT11:Version v1 : (shared, decide) / ()
DT11:Version v2 : (decide, return) / (borrow, update, decide)
```

(3) DT12 leiht sich die Version v2 aus, wodurch die lokale Version v3 entsteht. Die Außenwirkung muß *decide* und *return* enthalten, da DT11 diese Rechte behält.

```
DT12:Version v2 : (borrow, update, decide) / (decide, return)
DT12:Version v3 : (borrow, update, decide) / ()
```

(4) DT12 gibt die lokale Version v3 durch Überschreiben von v2 zurück und gibt durch Aufgabe des *decide*-Rechts ihre Änderungen frei. Dadurch liegt die Entscheidung über die Änderungen jetzt allein bei DT11.

```
DT12:Version v2 : Alle Rechte von DT12 werden entfernt.
DT12:Version v3 wird entfernt.
```

(5) DT11 erhält die temporäre Version v2 durch Überschreiben von v1 zurück.

```
DT11:Version v1 : (update, decide) / ()
DT11:Version v2 wird entfernt.
```

(6) DT11 gibt die Version v1 frei, indem sie eine neue Version v' von v ableitet. Sie behält sich aber eine Rücknahme vor.

```
DT11:Version v1 wird entfernt
DT11:Version v' : (decide) / (all)
DT11:Version v  : (decide) / (all)
```

(7) DT11 führt ein *commit* für die Erzeugung der Version v' durch. Danach werden alle Rechte für DT11 entfernt.

4.7 Rechte für Konsistenzzustände

Das konventionelle Transaktionsmodell unterscheidet implizit zwischen den beiden Zuständen "committed"[2] und "uncommitted", d.h. entweder ist ein Objekt stabil oder instabil. Entwurfsprozesse beinhalten jedoch die Definition anwendungsspezifischer Konsistenzzustände, z.B. beim Softwareentwurf *uncompiled*, *compiled*, *tested* etc. Eine DT sollte beim Anfordern einer Version in der Lage sein, festzustellen, in welchem Konsistenzzustand sich diese befindet. Umgekehrt sollte sie festlegen können, in welchem Konsistenzzustand sie eine Version freigibt, um zu vermeiden, daß andere DTs ihre Änderungen sehen können, bevor ein bestimmter Konsistenzzustand erreicht ist. Analoges gilt für den Austausch von Versionen durch Kooperation.

Die Konsistenzzustände können als objektbezogene Sperren (OR-Sperren) modelliert werden, wobei die zur Verfügung stehenden Zustände anwendungsabhängig definiert werden können (beispielsweise auf der Basis von *Features* [11]). Rechte werden nur gewährt, wenn der Zustand dies erlaubt.

Eine DT muß dann beim Anfordern einer Version zwei zusätzliche Rechte setzen:

- *oldstate* : gibt an, in welchem Zustand (oder in welcher Menge von Zuständen) sich die Version befinden muß.
- *newstate* : gibt an, in welchen Zustand (oder in welche Menge von Zuständen) die Version überführt werden darf.

Bei beiden Rechten können Operatoren wie "==", "!=", ">" und "<" eingesetzt werden, um den Konsistenzzustand zu spezifieren. Die letzteren beiden Operatoren sind sinnvoll, wenn eine Ordnung auf den Konsistenzzuständen definiert ist.

Wenn zum Beispiel eine Version den Konsistenzzustand *compiled* besitzt (als OR-Sperre), so wird das Recht *oldstate* > *uncompiled* gewährt, das Recht *oldstate* == *tested* jedoch nicht. Will eine DT die Version in den Konsistenzzustand *tested* überführen, so muß sie das Recht *newstate* == *tested* anfordern.

Andere DTs können wiederum Restriktionen für den Zugriff auf Versionen und für die Zustandsüberführungen festlegen.

4.8 Zusammenfassung der Rechte

In Abb. 6 werden die aufgeführten Rechte noch einmal systematisch aufgelistet. Die Rechte können grob unterschieden werden in:

[2]Bei geschachtelten Transaktionen bezieht sich der Begriff "committed" nur auf die jeweilige Hierarchieebene.

- Rechte für Zugriffe auf Versionen (*shared* etc.).
- Rechte für Kooperation (*return, borrow*).
- Rechte für *abort/commit* (*decide, independent*).
- Rechte für Zustände (*oldstate, newstate*).

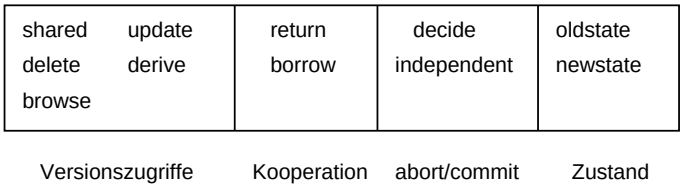

Abb. 6: Zusammenfassung der Rechte

5 Definition von Protokollen

Eine der Grundideen des Baukastens ist es, Transaktionen zu typisieren. Den Transaktionstypen werden u.a. Protokolle zugeordnet, die das Verhalten der entsprechenden Transaktionen beschreiben. Die in Kapitel 4 definierten Rechte erweitern diese Beschreibungsmöglichkeiten inbesondere um die Aspekte Kooperation und Recovery.

Um eine Version zu bearbeiten, muß eine DT bestimmte Rechte anfordern (entspricht der in Kap. 2 definierten Innenwirkung). Darüberhinaus kann sie festlegen, welche Rechte konkurrierende DTs gleichzeitig haben oder noch erwerben dürfen (entspricht der Außenwirkung). Voraussetzung für die Gewährung von Rechten ist, daß konkurrierende DTs diese Rechte nicht ausgeschlossen haben. Rechte können auch nachträglich geändert werden.

Ein Protokoll dient nun dazu, zu spezifizieren, welche Rechte eine DT anfordern bzw. freigeben und welche Rechte eine DT gewähren oder nicht gewähren darf. Es legt weiterhin fest, zu welchen Zeitpunkten bzw. in welchen Situationen dies geschehen darf. Auf diese Weise können die Eigenschaften und das Verhalten von DTs eines bestimmten DT-Typs genau definiert werden.

Im einzelnen definiert ein Protokoll folgende Eigenschaften:

- **Art des Rechts**:
  Es wird festgelegt, welche Rechte verwendet werden dürfen. Dabei kann unterschieden werden nach:
  - Anforderung von eigenen Rechten
    Ein Beispiel sind DTs, die nur Lese-, aber keine Schreibrechte erhalten dürfen.

Ein anderes Beispiel ist die Festlegung, ob eine DT Versionen entleihen darf (*borrow*-Recht).

- Freigabe von eigenen Rechten
  Ein Beispiel ist die Festlegung, ob eine DT das *decide*-Recht vor ihrem Ende aufgeben darf.
- Berücksichtigung von Rechten anderer DTs
  Hier wird z.B. festgelegt, ob eine DT auf Versionen zugreifen darf, auf die eine andere DT das *decide*-Recht gesetzt hat, ob sie auf verliehene Versionen zugreifen darf oder ob sie Versionen modifizieren darf, die parallel modifiziert werden.
- Gewährung von Rechten für andere DTs
  Hier kann z.B. definiert werden, ob eine DT das Entleihen erlauben darf oder ob sie parallele Modifikationen zulassen darf.

Es besteht also die Möglichkeit, die Vergabe und die Kompatibilität von Rechten für jeden DT-Typ individuell festzulegen.

- **Zeitpunkt**:
  Es wird festgelegt, zu welchen Zeitpunkten Rechte angefordert bzw. freigegeben werden können. Typische Einschränkungen sind:
  - *Preclaiming*
    Alle Rechte werden zu Beginn der DT angefordert. Hierdurch werden Deadlocks vermieden.
  - *Zweiphasigkeit*
    Die DT verläuft in zwei Phasen. In der ersten Phase dürfen Rechte nur angefordert, in der zweiten Phase nur freigegeben werden.
  - *Striktheit*
    Die Rechte dürfen erst bei Ende der DT freigegeben werden.

  Derartige Eigenschaften findet man auch bei der Definition von Sperrprotokollen (z.B. (striktes) Zweiphasen-Sperrprotokoll).

- **Konsistenzzustand**:
  Der DT-Typ kann festlegen, welche Konsistenzzustände Versionen haben sollen, auf die zugegriffen werden darf. Beispielsweise können DT-Typen definiert werden, die nur auf Versionen im Konsistenzzustand *tested* zugreifen können (*oldstate*-Recht). Dies muß für jede Zugriffsart getrennt spezifiziert werden, damit eine DT z.B. Versionen eines bestimmten Konsistenzzustands zwar lesen, aber nicht verändern darf. Weiterhin kann über das *newstate*-Recht festgelegt werden, in welche Konsistenzzustände die DT Versionen überführen darf. Der DT-Typ kann auch festlegen, in

welchen Konsistenzzuständen Versionen freigegeben werden dürfen. Beispielsweise können DT-Typen definiert werden, die nur Versionen im Konsistenzzustand *tested* freigeben können.

Ein weiterer Aspekt bei der Definition von Protokollen, der hier aus Platzgründen nur kurz erwähnt werden kann, bezieht sich auf die Durchführung von *commit*- oder *abort*-Vorgängen. Es kann sinnvoll sein, bestimmte Restriktionen einzuführen, z.B. durch Festlegung einer *commit*-Reihenfolge oder durch Ausschluß des *commit* der beteiligten DTs während eines Kooperationsvorgangs. Auch muß festgelegt werden, wie sich DTs verhalten, die von einem *abort* einer anderen DT betroffen sind.

Protokolle erlauben die Definition unterschiedlicher Arten von kooperierenden DTs. Sie erlauben weiterhin, unterschiedliche Recoverystrategien zu verfolgen. Eine DT kann z.B. durch ein entsprechendes Protokoll vermeiden, daß sie auf inkonsistente Daten zugreift, die noch von einem *abort* betroffen sein können. Umgekehrt kann eine DT sicherstellen, daß sie keine inkonsistenten Daten freigibt. Es ist also möglich, "Barrieren" gegen die Ausbreitung von *aborts* zu bilden.

Durch die Definition dieser Aspekte können unterschiedliche Grade an Flexibilität und Sicherheit realisiert werden. DTs mit konventionellen Eigenschaften lassen sich realisieren, indem nur zweiphasige Protokolle und konventionelle Rechte mit der gängigen Kompatibilitätsdefinition verwendet werden. Kooperative DTs dagegen werden ein nichtzweiphasiges Protokoll einsetzen, Kooperation (z.B. Entleihen) und parallele Änderungen erlauben sowie Konsistenzzustände verwenden. Zwischen diesen beiden Extremen liegen eine Vielzahl von Möglichkeiten, DT-Typen anwendungsgerecht zu definieren.

Die Definition von Protokollen und die Anwendung von Rechten sind im Unterschied zum konventionellen Transaktionsmodell sehr komplex. Deshalb sollten dem Anwender auf Basis dieser Mechanismen Operationen zur Verfügung gestellt werden, die einen einfachen Umgang mit DTs ohne Kenntnis der internen Vorgänge erlauben.

6 Beispiel für eine DT-Hierarchie

Abb. 7 zeigt ein Beispiel aus dem Entwurfsbereich. Ein Projekt gliedert sich in die Bereiche *Entwicklung*, *Test* und *Kundenbetreuung*. Die Entwicklung zerfällt in zwei Subprojekte, an denen jeweils einige Entwickler arbeiten. Der Test wird von zwei Testern durchgeführt. Innerhalb dieser Hierarchie gelten prinzipiell unterschiedliche Verhaltensweisen, die durch Protokolle definiert werden müssen.

Beispiele hierfür sind:

- In den beiden Entwicklungs-Subprojekten kann kooperativ gearbeitet werden. Des-

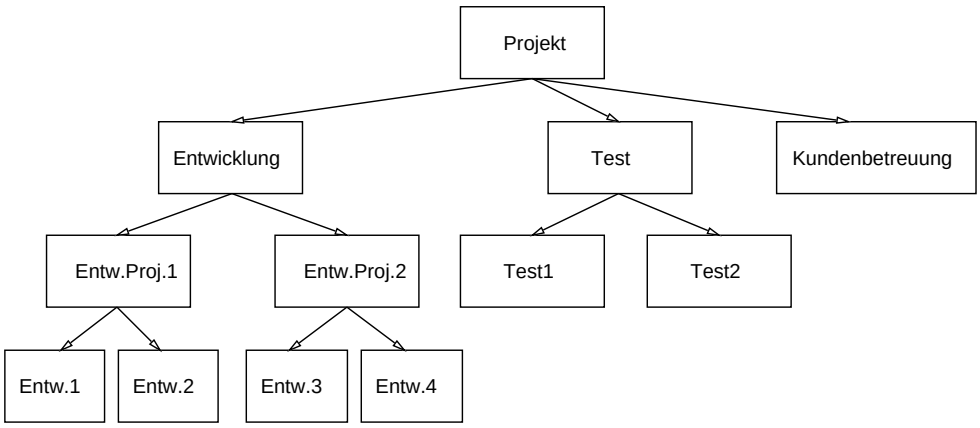

Abb. 7: Beispiel für eine DT-Hierarchie

halb werden Protokolle definiert, die parallele Modifikationen durch die Entwickler, Ausleihen von Versionen und frühzeitige Freigabe von noch inkonsistenten Versionen erlauben.

- Die Entwicklung verlangt, daß die Subprojekte nur solche Versionen untereinander austauschen dürfen, die wenigstens den Zustand *compiled* besitzen.
- Innerhalb der Entwicklung findet eine explizite Versionierung statt.
- Die Entwicklung darf nur solche Versionen freigeben, die einen Alpha-Test durchlaufen haben.
- Die Testgruppe greift nur auf Versionen zu, die mindestens einen Alpha-Test durchlaufen haben. Sie gibt nur solche Versionen frei, die einen Beta-Test durchlaufen haben.
- Innerhalb der Testgruppe besteht freie Kooperation zwischen den Testern.
- Die Kundenbetreuung greift nur auf beta-getestete Versionen zu.
- Innerhalb der Kundenbetreuung findet eine explizite Versionierung der vom Test freigegebenen Versionen statt.
- Beim *check_in* von Versionen in den Pool des Gesamtprojektes (z.B. von der Entwicklung) muß das *decide*-Recht aufgegeben werden.

Durch derartige Definitionen ist es z.B. ausgeschlossen, daß die Kundenbetreuung auf inkonsistente Daten zugreift und ggfs. von einem *abort* einer Entwicklungstransaktion betroffen ist. Jeder Knoten im Baum definiert, welche Informationen hinein- oder hinausfließen, und kann damit autonom Sicherheit und Flexibilität gegeneinander abwägen.

7 Vergleich mit anderen Ansätzen

Der beschriebene Ansatz basiert auf dem **Transaktionsbaukasten** aus [5]. Dieser Baukasten zeichnet sich insbesondere durch die Definition heterogener Transaktionshierarchien und durch die flexiblen Sperren aus. Wir erweitern diesen Ansatz um einen Rechtemechanismus zur Unterstützung von Kooperation und Recovery.

Die Spezifikation einer Hierarchie von **kooperativen Transaktionen** ist auch Ziel von [12]. Dort werden zwei Klassen von Transaktionen unterschieden, kooperierende Transaktionen, die den eigentlichen Entwurf durchführen, und Transaktionsgruppen, innerhalb derer Kooperation möglich ist. Der hier beschriebene Ansatz erlaubt eine flexiblere Definition und Schachtelung unterschiedlicher Typen von Transaktionen. Konsistenzbedingungen werden in [12] mit Hilfe von Grammatiken definiert. Dieses Verfahren ist geeignet zur statischen Beschreibung von Entwurfsprozessen (z.B. zur Definition von Phasenmodellen), während unser Ansatz Basismechanismen für den dynamischen Ablauf von Entwurfstransaktionen anbietet. Es ist denkbar, den Baukasten um Beschreibungsmöglichkeiten für Entwurfsabläufe zu erweitern.

Recovery für Nichtstandard-Transaktionen wird von den meisten Ansätzen über sog. **Kompensationstransaktionen** realisiert (z.B. [13, 14, 15, 16, 17]). Während in bestimmten Umgebungen Kompensationstransaktionen realistisch sind (z.B. die Stornierung einer Buchung), erscheint dieses Konzept im Entwurfsbereich eher fragwürdig, da Kompensation nur unter eingeschränkten Bedingungen eingesetzt werden kann und die explizite Programmierung von Kompensationstransaktionen voraussetzt. In unserem Ansatz werden Möglichkeiten geschaffen, die Recovery zu kontrollieren, z.B. durch Protokolle, die bestimmte Restriktionen definieren. Damit kann verhindert werden, daß Fehler sich beliebig ausbreiten bzw. daß beliebige Abhängigkeiten zwischen Transaktionen entstehen.

Das **ACTA-Framework** [18] bietet Konzepte zur formalen Spezifikation von Transaktionsmodellen und hat damit auch den Charakter eines (theoretischen) Baukastens. Unser Baukasten ist jedoch konkreter auf die Anforderungen von Entwurfsumgebungen abgestimmt, indem geeignete Primitive (z.B. Ausleihen) angeboten werden. Auch berücksichtigt ACTA nicht Aspekte wie die Konsistenzzustände von Objekten oder die Heterogenität der Transaktionshierarchie. Das ACTA-Konzept der *responsability* besitzt eine Ähnlichkeit mit dem hier eingeführten *decide*-Recht.

8 Zusammenfassung und Ausblick

In diesem Papier wird ein Verfahren vorgestellt, das über die Definition von Rechten Unterstützung für Kooperation und Recovery bietet. Das Verfahren besitzt insbesondere

folgende Eigenschaften:

- Modellierung typischer Vorgänge in Entwurfsumgebungen (z.B. Ausleihen).
- Berücksichtigung von Versionierung.
- Unterstützung von Konsistenzzuständen, die anwendungsspezifisch definiert werden können.
- Definition unterschiedlicher Typen von Transaktionen mit verschiedenen Graden an Sicherheit und Flexibilität.
- Konfiguration von heterogenen, anwendungsspezifischen Transaktions-Hierarchien.

Durch das Baukastenprinzip existieren weitreichende Möglichkeiten zur Definition von Transaktionen für Entwurfsumgebungen.

Eine detaillierte Betrachtung von Recoverymechanismen sowie die Umsetzung der Konzepte (z.B. durch Sperren oder Regeln) sollen in späteren Arbeiten abgehandelt werden. Auch wird es notwendig sein, die Struktur der Entwurfsdaten (Datenschema) einzubeziehen. Weiterhin wird z.Zt. im *CADLAB* auf Basis des *JESSI Common Frameworks* [19] ein Prototyp zur Validierung der Konzepte erstellt.

Literatur

[1] Rammig, F.; Steinmüller, B.: Frameworks und Entwurfsumgebungen. *Informatik Spektrum* 15 (1992), S. 33–43.

[2] Bernstein, P.; Hadzilacos, V.; Goodman, N.: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1992.

[4] Härder, T.; Reuter, A.: Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys* 15 (1983), Nr. 4, S. 287–317.

[5] Unland, R.; Schlageter, G.: A Transaction Manager Development Facility for Non Standard Database Systems. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 399–466, Morgan Kaufmann, 1992.

[6] Elmagarmid, A.: *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[7] Moss, J.: *Nested Transactions - An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[8] Tichy, W.: RCS – A System for Version Control. *Software Practice and Experience* 15 (1985), Nr. 7, S. 637–654.

[9] Barbian, G.; Schlageter, G.: CODA – a GROUPBASE-System for Cooperative Design Applications. In: *Proc. ICICIS*, May 1993.

[10] Dayal, U.; Hsu, M.; Ladin, R.: Organizing Long-Running Activities with Triggers and Transactions. In: *Proc. SIGMOD Conf. on Management of Data*, S. 204–214, May 1990.

[11] Kaefer, W.: A Framework for Version-based Cooperation Control. In: *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA)*, April 1991.

[12] Nodine, M.; Ramaswamy, S.; Zdonik, S.: A Cooperative Transaction Model for Design Databases. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 53–85, Morgan Kaufmann, 1992.

[13] Korth, H.; Levy, E.; Silberschatz, A.: A Formal Approach to Recovery by Compensating Transactions. In: *Proc. Conf. on Very Large Data Bases*, S. 95–106, August 1990.

[14] Buchmann, A.; Özsu, M.; Hornick, M.; Georgakopoulos, D.; Manola, F.: A Transaction Model for Active Distributed Object Systems. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 123–158, Morgan Kaufmann, 1992.

[15] Muth, P.; Rakow, T.; Klas, W.; Neuhold, E.: A Transaction Model for an Open Publication Environment. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 159–218, Morgan Kaufmann, 1992.

[16] Wächter, H.; Reuter, A.: The ConTract Model. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 219–263, Morgan Kaufmann, 1992.

[17] Weikum, G.; Schek, H.: Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 515–553, Morgan Kaufmann, 1992.

[18] Chrysanthis, P.; Ramamritham, K.: ACTA: The SAGA Continues. In: *Database Transaction Models for Advanced Applications* (Elmagarmid, A., ed.), S. 349–397, Morgan Kaufmann, 1992.

[19] Steinmüller, B.: The JESSI-COMMON-FRAME Project - A Project Overview. In: *Proc. 3rd IFIP Workshop on Electronic Design Automation Frameworks*, S. 227–238, March 1992.