Implementierung eines Generators für die Versant Datenbank

Jan Peter Riegel

Betreuer:
Joachim Altmeyer

AG VLSI Entwurf und Architektur
Prof. Dr. G. Zimmermann

Fachbereich Informatik
Universität Kaiserslautern

21.9.94

Inhaltsverzeichnis

Vorwort

Die vorliegende Projektarbeit befasst sich mit der Erweiterung des Software-Entwicklungswerkzeuges MOOSE um einen Generator für das Versant Datenbank-Management-System[1].

Das in der AG VLSI Entwurf und Architektur der Universität Kaiserslautern entwickelte Programm MOOSE dient zur graphischen Definition und Verwaltung von Datenschemata. Die Schemata beschreiben das Datenmodell eines (zu entwickelnden) Applikationsprogrammes, und können in Form von erweiterten Entity-Relationship Diagrammen (EER-Diagrammen) erzeugt und verändert werden. Ausgehend von diesen Schemata können mit Hilfe von Generatoren C oder C++-Programme erzeugt werden, welche die Datenhaltung im Hauptspeicher unterstützen. Die generierten Quelltexte stellen abstrakte Datentypen (ADTs) dar, die in beliebige (C oder C++) Programme eingebunden werden können. Somit erhält man automatisch eine Komponente zur Verwaltung der Daten in Applikationsprogrammen (Eine genaue Beschreibung des Schemaentwurfs und des C-Generators findet sich in [Sol93] und [Tre91], der C++-Generator ist in [Wer94] beschrieben).

Die vorliegende Arbeit beschreibt das Konzept und die Implementierung eines Generators für das Versant Datenbanksystem. Die Idee dabei ist, die erstellten Datenbankschemata direkt auf eine objektorientierte Datenbank zu übertragen. Dazu wird ein ADT erzeugt, der den Zugriff auf Objekte in einer Datenbank ermöglicht. Somit können von einem Applikationsprogramm nicht nur lokal im Hauptspeicher liegende, sondern auch dauerhaft in einer Datenbank abgelegte Objekte verwendet werden.

Die Schnittstelle die der Versant-ADT dem Programm dabei anbietet unterscheidet sich nur sehr gering von der des C++-ADTs. Dadurch können Programme leicht an einen der beiden ADTs angepasst werden.

Zusätzlich zu der Implementierung des Generators wurden Benchmarks für die generierten Quelltexte geschrieben um die Geschwindigkeit der Datenbank zu messen. Die Benchmarks wurden jeweils mit und ohne Datenbank laufen gelassen, um so Aussagen über den Aufwand zur Datenhaltung in einer Datenbank machen zu können.

1. Das Versant Datenbank-Management-System (DBMS) ist ein kommerzielles Produkt der Firma Versant Object Technologies, welches die Generierung und den Zugriff auf eine objektorientierte Datenbank ermöglicht.

Das zugrundeliegende Konzept des Versant-ADTs wird näher in der Einleitung (Kapitel 1) beschrieben. Kapitel 2 befaßt sich mit dem Versant-DBMS. Dort werden neben der allgemeinen Beschreibung des Datenbanksystems auch die Zeitmessungen erläutert. Details zur Implementierung des Generators und der von der Versant Bibliothek zur verfügung gestellten Programmierschnittstelle finden sich in Kapitel 3.
Zusätzlich zur reinen Datenhaltung wurde ein Mechanismus implementiert, mit dem Daten zwischen einem auf einer Datenbank beruhendem ADT und einem Hauptspeicher-ADT ausgetauscht werden können. Dieser Mechanismus wird in Kapitel 4 erläutert.
Erläuterungen und Beispielprogramme beruhen auf einem Datenbankschema aus dem VLSI-Entwurf. Dieses ist im Anhang A aufgeführt.

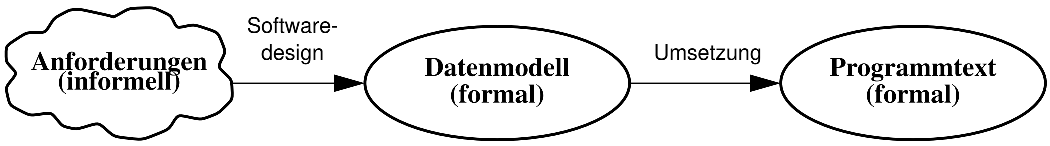
Für das Verständnis von Kapitel 3 und den Beispielprogrammen werden grundlegende Kenntnisse in objektorientierter Programmierung und der Programmiersprache C++ vorausgesetzt. Eine Einführung in C++ findet man z.B. in [Str92].

Kapitel 1

Einleitung

Eine wichtige Komponente im Software-Entwicklungsprozeß ist die korrekte und effiziente Modellierung der zur verwaltenden Daten. Dazu muß aus der meist informellen Problembeschreibung ein Datenmodell entwickelt werden, welches die einzelnen Objekte und deren Zusammenhänge aus der „realen Welt“ wiederspiegelt. Dieses Datenmodell dient zum einen zur Dokumentation des zu Grunde liegenden Problems und wird als Spezifikation der Datenbasis im späteren Programm benutzt.
Wie in Abb. 1 dargestellt, gliedert sich der Entwurf einer Softwarekomponente zur Datenverwaltung in zwei Teile. Zunächst muß aus den informellen Programmanforderungen (Software Requirements) durch Abstaktion ein geeignetes (d.h. korrektes, vollständiges und effizientes) Datenmodell entwickelt werden und anschließend wird aus diesem Modell eine Datenhaltungskomponente für das spätere Programm erzeugt.

Abb. 1 Datenmodellentwurf



Da es sich bei dem Datenmodell um eine formale Beschreibung des Problems handelt, liegt es nahe, die Umsetzung des Modells in ein Programm automatisch mit einem Computer durchzuführen. Um sowohl den Entwurf als auch die Umsetzung des Datenmodells zu unterstützen, wurde das Programm MOOSE[2] entwickelt. Mit diesem Programm können Datenbankschemata in graphischer Form erstellt und dokumentiert werden. Als Darstellungsform der Schemata

2. MOOSE: Modell Based Object Oriented Software Generation Environment. Das Programm ist hervorgegangen aus dem Datenbankeditor DBED (siehe [Tre91]) und ist die Folgeversion des graphischen Schemaeditors GSHED. Zur weiteren Dokumentation siehe [Sol93].

bedient sich das Programm erweiterten Entity-Relationship Diagrammen[3]. Dadurch können auf sehr einfache und direkte Art Objekte und ihre Zusammenhänge (Relationen) dargestellt werden.
Zur Umsetzung der Datenbankschemata in Programme existieren in MOOSE Software-Generatoren. Diese erzeugen aus den Datenbankschemata automatisch Programmteile zur Verwaltung der Daten im späteren Programm.

1.1 Anforderungen an CAD Systeme

Ein Schwerpunkt der AG VLSI Entwurf und Architektur ist die Entwicklung eines Frameworks zum Entwurf von VLSI-Chips. Die dabei zu verwaltenden Daten sind typisch für CAD Anwendungen. Ein Chip besteht aus vielen Teilschaltungen (Zellen), Anschlußpins und interner Verdrahtung. Die einzelnen Zellen sind wiederum hierarchisch in kleinere Zellen aufgeteilt. Um die Daten solch eines Chips im Computer abzuspeichern, benötigt man sehr viele Datensätze die untereinander stark vernetzt sind. Jeder einzelne Datensatz ist dabei meist relativ klein.
Speichert man alle Daten eines Chips in einer relationalen Datenbank ab, so entstehen sehr viele Tabellen mit wenig Spalten (d.h. Attributen) die allerdings viele Zeilen haben (also viele Datensätze enthalten). Um Teile des Chips anzusprechen, sind zahlreiche Zugriffe und Verknüpfungen einzelner Tabellen notwendig, wodurch die Datenbankanfrage viel Zeit benötigt. Eine effizientere Datenhaltung solcher Datensätze ist mit einem objektorientierten Ansatz möglich. Jeder Datensatz (z.B. cell, net, pin, …) entspricht hierbei einem Objekt. Verknüpfungen zwischen den Objekten werden über Zeiger modelliert, so daß sehr schnell auf Objekte zugegriffen werden kann, die mit dem Ausgangsobjekt zusammenhängen.
Der C++-Generator des MOOSE bildet Schemabeschreibungen direkt auf eine objektorientierte Datenstruktur ab. Dabei werden automatisch Funktionen zum Zugriff, Ändern und Vergleichen einzelner Objekte erzeugt. Eine genaue Beschreibung des C++-Generators findet sich in [Wer94].
Typisch für CAD-Datenbanken ist nicht nur der Aufbau einzelner Datensätze sondern auch die Menge aller vorhandenen Daten und die Häufigkeit mit der auf die Daten Zugegriffen wird. Beim Entwurf eines Chips wird dieser sukkzessiv in kleinere Teile unterteilt bis die einzelnen Teile so klein sind, daß sie als Standardteil in einer Datenbank abgelegt und später wiederverwendet werden können. Dabei wird die Struktur getrennt vom eigentlichen Aufbau der einzelnen Komponenten in der Datenbank abgelegt. Auf diese Art können verschiedene Realisierungen des gleichen Chips modelliert werden. Sowohl die Struktur als auch der Inhalt der Komponenten kann für einen späteren Entwurf wiederverwendet werden.
Für einen Speicherbaustein der aus tausenden von Flip-Flops besteht, braucht zum Beispiel nur ein einziges Flip-Flop komplett entworfen zu werden. Alle Flip-Flops des Chips sind dann nur Verweise auf dieses eine Standardteil. Die genaue Beschreibung eines Flip-Flops und der Gatter, aus denen es aufgebaut ist, können in einer Bibliothek abgelegt werden. So entstehen schnell große Bibliotheken von Standardobjekten, die beim Entwurf komplexer Schaltungen verwendet werden können. Umgekehrt werden beim Entwurf neuer Schaltungen nur wenig Objekte aus einer Bibliothek wirklich benötigt.

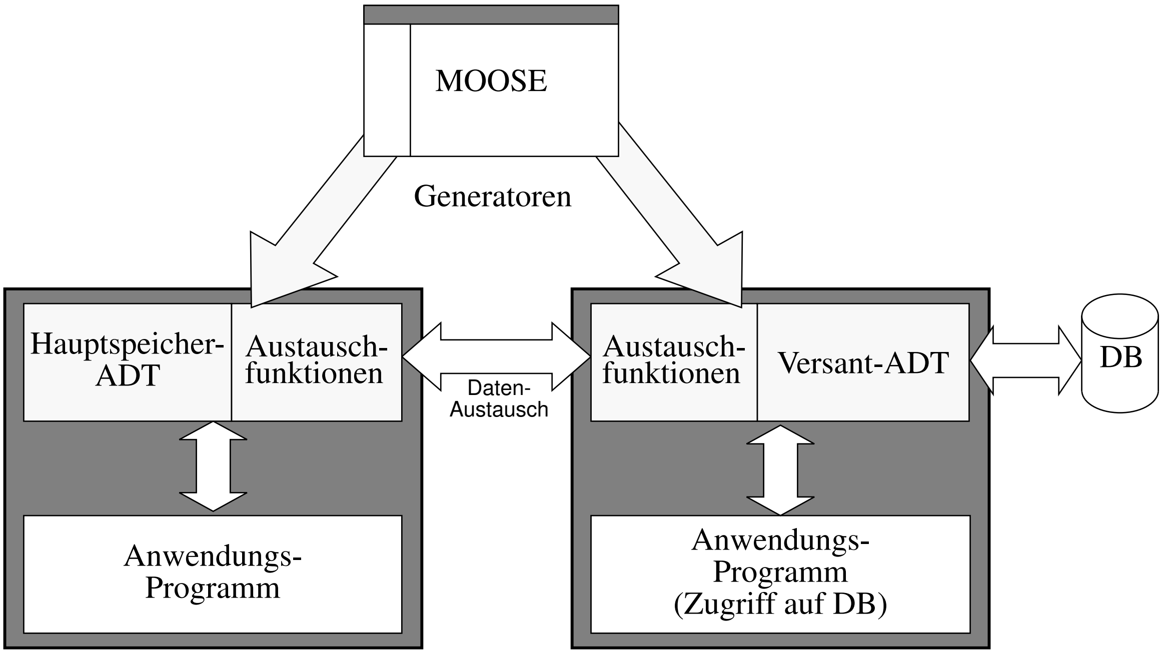
3. Entity-Relationship-Diagramme sind in [Che91] beschrieben. Erweitert wurden sie um die Konzepte der Vererbung und der Aggregation.

Bedingt durch die Größe dieser Bibliotheken ist eine Verwaltung sämtlicher Daten im Hauptspeicher nicht immer sinnvoll. Es bietet sich daher an, die Daten in einer objektorientierten Datenbank abzulegen. Zu diesem Zweck wurde in der vorliegenden Projektarbeit ein Generator für das Versant-DBMS implementiert. Neben dem Anlegen einer Datenbank werden von den Generatoren des MOOSE noch weitere grundlegende Softwarekomponenten zum vereinfachten Umgang mit Objekten erzeugt. Dazu gehören zum Beispiel Zugriffsfunktionen, Vergleichsoperatoren, Methoden zum Datenaustausch und weiteres.
Das Versant DBMS stellt sämtliche Funktionen zum Umgang mit Objekten in einer objektorientierte Datenbank zur Verfügung. Der Versant-Generator des MOOSE erstellt eine Bibliothek, die unter Zuhilfenahme dieser Zugriffsfunktionen eine Programmierschnittstelle zur Verfügung stellt, die fast genau so wie der C++ (Hauptspeicher-) ADT benutzt werden kann. Die anfallenden Daten werden dabei jedoch in einer Datenbank verwaltet. So können Programme leicht von dem Hauptspeicher-ADT auf den ADT für die Datenbank (oder umgekehrt) portiert werden.

1.2 Datenaustauschkonzept

Naturgemäß nimmt die Verwaltung von Daten in einer Datenbank einige Zeit in Anspruch (Zugriffe auf Sekundärspeicher, Mehrbenutzerverwaltung, ect.). Da viele Probleme beim Chip-Entwurf sehr komplex sind (und deren Lösungen somit lange Rechenzeiten benötigen), die dabei benutzten Daten im Vergleich zum Gesamtdatenaufkommen jedoch gering sind, scheint es sinnvoll diese Probleme basierend auf einem Hauptspeicher-ADT zu berechnen. Dadurch wird eine maximale Geschwindigkeit beim Datenzugriff erreicht.
Um nun solche Probleme effizient zu berechnen kann man in drei Schritten vorgehen: zunächst werden sämtliche für die Berechnung wichtigen Daten von einem auf dem Versant-ADT beruhenden Programm aus der Datenbank ausgelesen und in eine spezielle Datei geschrieben. Diese Datei wird von einem weiteren Programm eingelesen und die darin enthaltenen Datensätze werden im Hauptspeicher abgelegt. Die Berechnungen können jetzt auf diesen Daten ablaufen. Zuletzt werden die neu berechneten Werte wieder in eine Datei abgespeichert und zurück in die Datenbank geschrieben.
Die bei dieser Vorgehensweise benötigten Ein- und Ausgabefunktionen sowie das Datenaustauschformat wird dabei auf Wunsch automatisch von den Softwaregeneratoren des MOOSE mitgeneriert.

Abb. 2 Versant-ADT



Das gesamte Szenario ist noch einmal in Abb. 2 dargestellt. Ausgehend von einer Modellbeschreibung in MOOSE können verschiedene abstrakte Datentypen generiert werden. Diese übernehmen die Datenverwaltung für das geplante Anwendungsprogramm. Je nach Art der Problemstellung kann ein Anwendungsprogramm auf den C++ (Hauptspeicher-) ADT oder den Versant-ADT zurückgreifen. Die Verwaltung der anfallenden Daten kann grundsätzlich immer über die Datenbank erfolgen. Ist die Menge der Daten allerdings gering, so ist der Verwaltungsaufwand der Datenbank jedoch übertrieben hoch. Müssen im Applikationsprogramm nur wenig Daten schnell verwaltet werden, so bietet sich die Benutzung des Hauptspeicher-ADTs an. Bei großem Datenaufkommen können die Daten mit dem Versant-ADT auf einfache Art und Weise in einer Datenbank verwaltet werden. Soll ein Algorithmus möglichst schnell auf wenigen Datensätzen aus einer Datenbank arbeiten, so bietet sich es sich an, mehrere Programme zu schreiben. Ein relativ kleines Programm kümmert sich dabei um den Datenaustausch mit der Datenbank und legt auf Anfrage die vom Algorithmus benötigten Objekte in einer Datei ab. Die eigentliche Berechnung läuft dann im Hauptspeicher ab, nachdem die Daten aus der Datei eingelesen wurden. Nach der Berechnung werden die neuen Daten wieder über den Umweg einer Datei in die Datenbank zurückgeschrieben (siehe auch Beispiel 14).

Ein weiterer Fall, in dem der Versant-ADT benutzt werden kann, tritt ein, wenn Datensätze von mehreren Benutzern bearbeitet werden sollen. Während beim Hauptspeicher-ADT sämtliche Daten nur lokal vorhanden sind, besteht bei einer Datenbank die Möglichkeit, Datensätze mehreren Benutzern gleichzeitig zugänglich zu machen. Die dabei auftretenden Probleme bei konkurrenten Zugriffen auf gleiche Datensätze werden über Sperren (Locks) geregelt. So ist es z.B. möglich, daß ein Objekt von mehreren Benutzern gleichzeitig gelesen werden kann, ein schreibender Zugriff darf aber immer nur von einem Benutzer erfolgen.

Durch den Mechanismus der langen Transaktionen können auch einzelne Datensätze für eine längere Zeit vor dem Zugriff von anderen Benutzern gesperrt werden. Dies ist beispielsweise sinnvoll um neu entwickelte Objekte erst komplett zu erstellen und zu testen bevor sie in die Datenbank aufgenommen werden.
Es ist also auch möglich, Datensätze in der Datenbank zur Bearbeitung zu reservieren, diese mit einem, auf dem Hauptspeicher-ADT beruhenden, Programm zu verändern und sie zuletzt in der Datenbank zu aktualisieren und wieder freizugeben. Der Datenaustausch zwischen der Datenbank und dem Hauptspeicher-ADT kann dabei über eine Datei im SLANG-Format erfolgen. Eine lange Transaktion sorgt dafür, daß die Datenbank die ganze Zeit über in einem konsistenten Zustand bleibt.

Kapitel 2

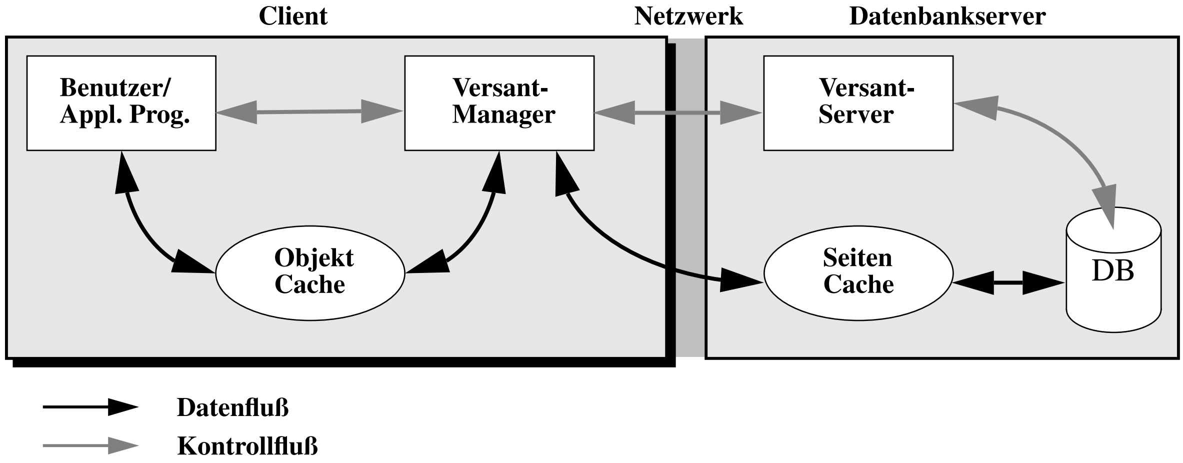
Konzept des Versant DBMS

An dieser Stelle soll zunächst auf das Konzept des Versant Datenbank Management Systems eingegangen werden. Die Benutzung des Versant-ADTs erfolgt im großen und ganzen genau wie die des C++-ADTs. Unterschiede und Erweiterungen ergeben sich hauptsächlich dadurch, daß zusätzlich eine (Mehrbenutzer-) Datenbank verwaltet werden muß.
Ansonsten beruhen sowohl der C++-ADT als auch die von Versant zur Verfügung gestellten Funktionen auf der NIHCL-Bibliothek. Dies ist eine Sammlung von nützlichen Programmteilen, mit denen zum Beispiel Felder oder sortierte Listen verwaltet werden können. Eine Beschreibung der NIHCL Klassen und Funktionen ist in [Gor90] und in [GOP91] zu finden.

2.1 Client/Server Konzept

Das Versant Datenbank Management System (DBMS) ist als Mehrbenutzersystem konzipiert. Dabei kann auf eine Datenbank von mehreren Benutzern netzwerkweit zugegriffen werden. Ein Benutzer (Applikationsprogramm) kann also Verbindungen zu einer Datenbank aufbauen, die sich physikalisch auf einem beliebigen Computer im lokalen Netzwerk befindet. Es können auch Verbindungen zu mehreren Datenbanken gleichzeitig aufgebaut werden. Die Verwaltung der Datenbankzugriffe und die Netzwerkkommunikation übernimmt dabei die von Versant zur Verfügung gestellte Bibliothek.
Um die verschiedenen Aufgaben des DBMS zu bewerkstelligen, ist die Verwaltung einer Datenbank in zwei Prozesse aufgeteilt. Der Versant Server regelt die direkten Zugriffe auf die Datenbank. Dabei werden einzelne Seiten der Datenbank in einem Cache abgelegt, um den mehrfachen Zugriff auf diese Daten zu beschleunigen. Für jede Datenbank auf die zugegriffen wird, läuft ein Versant Server. Mit diesem kommunizieren die Versant Manager. Jedem Prozeß ist ein Manager zugeordnet. Dieser kümmert sich um den Zugriff auf die einzelnen Objekte und regelt auch die Vergabe der Locks. Wird ein Objekt referenziert, so wird es in einem lokalen Cache abgelegt. Dadurch verringert sich die Zugriffszeit bei mehrfachen Zugriffen auf dasselbe Objekt.

Abb. 3 Versant Server/Manager



Der Zugriff auf eine Datenbank erfolgt also, wie in Abb. 3 gezeigt, in mehreren Stufen. Soll auf ein Objekt aus einer Datenbank zugegriffen werden, so reagiert automatisch der Versant-Manager um die geforderten Daten bereitzustellen. Befindet sich das angeforderte Objekt noch nicht im Objekt-Cache, so fordert es der Manager vom Versant-Server an. Dieser verwaltet ein auf Datenbankseiten beruhendes Cache. Eine Seite beinhaltet einen Ausschnitt fester Größe aus der Datenbank der mehrere Objekte umfaßt. Die Größe und Aufteilung (Clusterung) einer Seite kann beim Anlegen der Datenbank beeinflußt werden (siehe [Ver92b]).
Wenn sich das vom Manager angeforderte Objekt nicht bereits im Seiten-Cache befindet, wird die entsprechende Seite aus der Datenbank dorthin geladen und anschließend werden die benötigten Daten über das Netzwerk[4] an den Manager übergeben. Dieser legt sie im lokalen Cache ab und übergibt dem Applikationsprogramm einen Zeiger auf das Objekt im Cache. Jetzt kann auf das Objekt wie auf ein normales C++-Objekt zugegriffen werden. Zusätzlich vergibt der Versant-Manager Sperren für alle benutzten Objekte, so daß Konflikte im Mehrbenutzerbetrieb vermieden werden (siehe 3.3.9).
Sämtliche Aktionen des Versant Servers und Managers geschehen weitgehend automatisch. Um Objekte aus einer Datenbank zu lesen oder zu schreiben werden keine zusätzlichen Programmkonstrukte benötigt. Ein Applikationsprogramm muß lediglich die Verbindung zu einer Datenbank auf- und wieder abbauen und muß die einzelnen Programmblöcke in Transaktionen schachteln. Der prinzipielle Aufbau eines Applikationsprogrammes ist in Kapitel 3.3. beschrieben.

2.2 Benutzung des Versant DBMS

Die Benutzung des Datenbanksystems ist recht einfach gehalten. Es existieren UNIX Programme zum Erstellen und Warten von einzelnen Datenbanken. Die wichtigsten sind *makedb* zum Anlegen einer neuen Datenbank, *sch2db* um Schemabeschreibungen in die Datenbank aufzunehmen und *removedb* um eine Datenbank wieder zu löschen. Das Erstellen und Konfigurieren einer Daten-

4. Laufen Server und Manager lokal auf einem Computer, so unterbleibt die Kommunikation über das Netzwerk.

bank für einen generierten ADT übernimmt ein Shell-Script, welches beim Generieren eines ADTs automatisch mitgeneriert wird (siehe 3.2.3). Weiterhin existieren Programme um ein Backup einer Datenbank zu erstellen, einzelne Klassen oder Instanzen aus einer Datenbank zu löschen und ein graphischer Datenbankbrowser. Diese Werkzeuge werden im Versant Manual [Ver92a] genauer beschrieben.

2.2.1 Einbindung in C++-Programme

Der Zugriff und Umgang mit Objekten aus einer Datenbank erfolgt in C++-Programmen weitestgehend automatisch. Außer beim Anlegen von Objekten brauchen keine weiteren Spezialbehandlungen für Datenbankobjekte im Vergleich zu normalen C++-Objekten gemacht zu werden. Darüberhinaus existieren allerdings zusätzlich noch einige spezielle Funktionen für den effizienten Umgang mit Datenbankobjekten (z.B. zum Lesen oder Sperren von Mengen von Objekten).
Die Tranzparenz zwischen Datenbankobjekten und C++-Objekten ist möglich durch einen eigenen Präprozessor der die C++-Programme nach datenbankspezifischen Operationen durchsucht bevor der eigentliche Compiler aufgerufen wird. Einzelheiten zum Umgang mit einer Datenbank von C++-Programmen aus sind im Kapitel 3 beschrieben.

2.3 Zeitmessungen

Durch die Einführung des Versant-Generators in das Software Entwicklungswerkzeug MOOSE muß bei der Planung eines Anwenderprogrammes zusätzlich noch entschieden werden, ob auf eine Datenbank zugegriffen werden soll, oder nicht. Um diese Entwurfsentscheinung zu erleichtern und um die Geschwindigkeit des Versant-DBMS beurteilen zu können, sind einige Zeitmessungen an einer Versant Datenbank vorgenommen worden.
Ausführliche Geschwindigkeitsmessungen von verschiedenen objektorientierten Datenbanksystemen[5] sind z.B. von Carey, DeWitt und Naughton an der University of Wisconsin-Madison durchgeführt worden [CDN94]. Sie entwickelten einen Benchmark (OO7-Benchmark) zum Testen von Datenbanken. Ausgehend von diesem Benchmark wurde die Geschwindigkeit des Versant-DBMS im Vergleich zu einem Hauptspeicher-ADT gemessen. Ausgangspunkt war hierbei jeweils eine von MOOSE generierte Klassenhierarchie an die der OO7-Benchmark angepaßt wurde.

Der OO7 Benchmark besteht im wesentlichen aus zwei Teilen. In einem ersten Teil („Traversal“) wird hauptsächlich untersucht, wie schnell Relationen zwischen Objekten verfolgt werden können. Der zweite Teil des Benchmarks beschäftigt sich mit typischen Datenbankanfragen („Query“). Im folgenden Abschnitt werden allerdings zunächst einige allgemeine Vergleiche zwischen den beiden Konzepten unternommen.

5. In diesen Vergleich wurden die Datenbanksysteme Objectivity/DB, Ontos, Versant und E/Exodus miteinander verglichen.

2.3.1 Allgemeines

In Tabelle 1 sind einige Angaben über die vom Benchmark erzeugte Datenbasis gemacht. Die unterschiedliche Größe kommt daher zustande, daß Versant zum einen zu den eigentlichen Daten auch Verwaltungsstrukturen (z.B die Klassenhierarchie und den Aufbau der einzelnen Klassen) abspeichert, und zum anderen daran, daß bei Versant Indices mitangelegt werden.

Tabelle 1: Allgemeine Messungen

Messung	Versant	NIHCL
Größe der Datenbasis	ca. 13,6 MB	ca. 9,3 MB
Größe des Cache-Speichers	16 MB Objekt Cache	Daten werden direkt im Hauptspeicher verwaltet (65 MB)
Daten in der Datenbasis	12.459	
Zeit zum Aufbau der Datenbasis	63,41 s	58,96 s

Der Aufbau der Datenbasis im OO7-Benchmark spiegelt den typischen Aufbau von CAD-Entwurfswerkzeugen wieder. Sämtliche Daten sind hierarchisch strukturiert. Auf diese Weise besteht jedes Entwurfsobjekt (außer den Primitiven) aus ein oder mehreren Teilobjekten. Ein typischer Entwurfsschritt bewegt sich auf mehreren Hierarchieebenen um entweder top-down zu dem aktuellen Objekt die Teilobjekte zu entwerfen, oder es wird aus mehreren Teilobjekten ein übergeordnetes Objekt synthetisiert (bottom-up Entwurf). In dieser Hinsicht spiegelt die OO7-Datenbasis auch die in der AG VLSI Entwurf und Architektur benutzte PLAYOUT-Datenbank wieder. Unterschiedlich bei dieser Datenbank ist vor allem, daß hier zu einem Entwurfsobjekt auch mehrere Datenbankobjekte einer Hierarchieebene gehören (eine Zelle besteht z.B. aus einem Interface und ein oder mehreren Inhalten in evtl. verschiedenen Versionen). Dieser Sachverhalt wird von dem Benchmark nicht berücksichtigt, Zugriffe auf einer Hierarchiebene sind allerdings im Wesentlichen ähnlich zu denen auf mehreren Ebenen sobald sich alle benötigten Objekte im Cache befinden.

2.3.2 Traversal

Die Datenbank für den Benchmark beinhaltet hierarchisch strukturierte Datensätze mit denen komplexe Entwurfsobjekte beschrieben werden sollen (eine genauere Beschreibung der Datenbasis ist im Anhang A.1 und in [CDN94] zu finden). Die unter Traversal aufgeführten Tests verfolgen diese hierarchische Struktur von einem Basisobjekt zu den Blättern und führen auf den Blättern verschiedene Operationen durch.

Es wird jeweils gemessen, wieviel Zeit das Suchen der Blätter und die Durchführung dieser Operationen benötigt. Bei einer Datenbank treten hierbei typischerweise zwei unterschiedliche Zeiten auf. Wird ein spezieller Benchmark zum ersten mal auf einer Datenbank laufen gelassen, so werden sogenannte „kalt“ -Zeiten gemessen. Eine Datenbank befindet sich in einem kalten Zustand, wenn sämtliche Cache Speicher leer sind; zur Ausführung eines Programms müssen dann zunächst die benötigten Daten in das Cache

geladen werden bevor mit ihnen gearbeitet werden kann. Sind die Datenbankcaches bereits mit den richtigen Objekten geladen (bzw. mit einem Teil der benötigten Daten falls der für das Cache reservierte Speicherplatz zu klein ist), so spricht man von einem „heißen“ Zustand. Die heiß-Zeiten werden gemessen, indem man denselben Benchmark mehrmals hintereinander laufen läßt und anschließend den Durchschnitt der mittleren Läufe berechnet. Der erste Lauf dient dabei zum Füllen des Caches, die weiteren Läufe spiegeln die heiß-Zeiten wieder. Die zuletzt gemessene Zeit wird bei der Duchschnittsbildung nicht berücksichtigt, um den Overhead beim commit zu vernachlässigen.
Um die Geschwindigkeit der Datenbank mit der des Hauptspeicher-ADTs vergleichen zu können, ist es nicht sinvoll die „kalt“ Zeiten heranzuziehen, da bei diesen Messungen langsame Festplatten- und Netzwerkzugriffe dominieren. Es ist vier mehr interessant, die heiß Zeiten zum Vergleich heranzuziehen. Dadurch können Aussagen gemacht werden, wie Zeitaufwendig der Verwaltungsaufwand der Daten in einer Datenbank ist.

In Tabelle 2 sind sämtliche gemessenen Zeiten (in Sekunden) aufgeführt; davon werden im folgenden Abschnitt einige Messungen näher erleutert. Die Nummerierung der einzelnen Tests wurde vom original OO7 Benchmark übernommen und ist daher nicht durchgängig (einige Messungen, die keine neuen Einsichten erbrachten, wurden weggelassen).

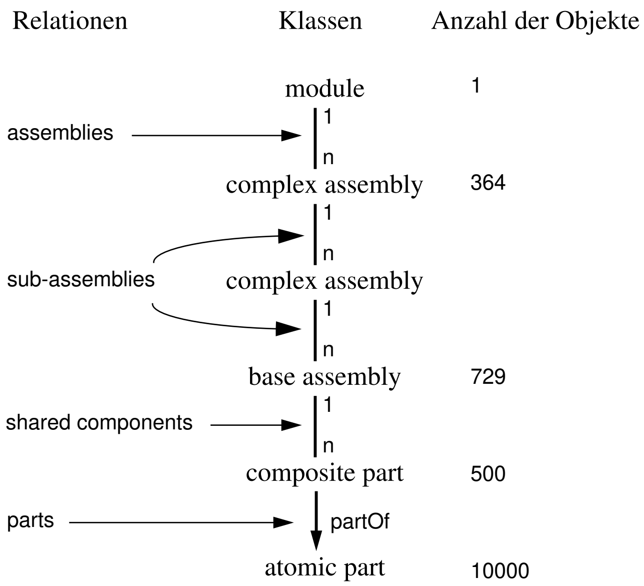
Tabelle 2: Traversal

Benchmark	Versant kalt	Versant heiß	NIHCL	Trade-off (Versant / NIHCL)
t1 one	25,9485	10,9501	0,2177	50,30
t1 many	31,1172	20,5568	0,2170	94,73
t2a	23,1140	11,0748	0,2161	51,25
t2b	22,5934	10,6712	0,2169	49,20
t2c	22,2804	10,8698	0,2213	49,12
t3a	22,2787	10,9116	0,2171	50,26
t3b	22,1206	10,8450	0,2169	50,00
t3c	22,5294	10,6151	0,2221	47,79
t6	7,4084	0,1547	0,1826	0,85
t8	0,4296	0,0334	0,0005	66,80
t9	0,2600	0,0009	0,0004	2,25

Benchmark t1 Beim Benchmark t1 wird die gesamte Datenbasis (s. Anhang A.1) top-down durchsucht. Ausgehend von den Modulen werden alle complex assemblies gesucht. Anschließend wird die dreistufige Assembly-Hierarchie durchsucht um zuletzt mit einer Tiefensuche sämtliche atomic parts zu finden (siehe Abb. 4). Die Gesamtzahl der gefundenen atomic parts wird am Schluß ausgegeben.

Bei diesem Geschwindigkeitstest wird eine Vielzahl von Objekten in ungeordneter Reihenfolge besucht. Gleichzeitig werden alle gefundenen Objekte mit einer Lesesperre versehen, sodaß sie nicht von einem anderen Programm verändert werden können.

Abb. 4 Daten-Traversal



Es zeigt sich, daß bei dieser Disziplin der Hauptspeicher-ADT wesentlch schneller als entsprechende Datenbankzugriffe ist (10,95 Sekunden zu 0,23), da hierbei zum Einen alle Objekte bereits im Hauptspeicher vorliegen und zum Anderen auf aufwendige Sperren verzichtet wird. Wird für jeden Testlauf bei der Datenbank eine eigene Transaktion gestartet (Test t1 many) so verdoppelt sich sogar noch die Laufzeit im Vergleich zu einer Transaktion (t1 one). Dies ist dadurch zu begründen, daß am Ende einer Transaktion sämtliche Cache-Speicher auf der Client Seite (Versant-Manager) gelöscht werden und anschließend erneut gefüllt werden müssen. Die Server-Caches bleiben allerdings erhalten und sind (im Falle einer heißen Datenbank) bereits mit den richtigen Daten gefüllt.

Benchmark t2 Der Benchmark t2 entspricht t1 mit der Erweiterung, daß nun auch Daten verändert werden. t2a vertauscht die x und y Werte eines gefundenen atomic parts pro composite part. Bei 2b werden alle gefundenen atomic parts geändert und 2c ändert schließlich alle atomic Parts viermal hintereinander.

Benchmark t3 Der Benchmark t3 verändert genau wie t2 die gefundenen atomic Parts. Unterschiedlich ist hierbei nur, daß nicht die x und y Werte vertauscht werden, sondern das build date. Dieses Attribut ist bei der Versant-Datenbank mit einem Index (B-Baum) versehen. Gemessen wird bei diesem Test der Overhead um den Index zu aktualisieren. Da die Änderungen des build dates allerdings sehr gering sind (Änderungen um 1), muß der B-Baum auch nur an wenigen Stellen angepaßt werden. Daher sind die gemessenen Zeiten fast identisch zu denen vom Benchmark t2. Beim Hauptspeicher-ADT wurden keine Indices verwendet.

Benchmark t6 Ähnlich wie bei t1 wird auch bei diesem Test die hierarchische Objektstruktur verfolgt. Unterschiedlich ist nur, daß statt alle atomic parts zu suchen jeweils nur eines pro composite part besucht wird. Bei diesem Test fallen also wesentlich weniger Daten an, als bei t1 (ca. 700 Objekte). Auffallend ist, daß bei geringem Datenaufkommen kaum ein Unterschied zwischen Datenbank und Hauptspeicher-ADT gemessen werden kann. Sobald alle Cache-Speicher geladen sind und die Zugriffssperren auf die benutzten Objekte vergeben sind (wie im heiß-Zustand der Datenbank) kann Versant offensichtlich extrem schnell mit den Daten umgehen.

Benchmarks t8 und t9 Im Gegensatz zu den vorhergehenden Benchmarks wo viele kleine Objekte gesucht wurden, wird jetzt nur ein großes Objekt betrachtet. Sobald sich dieses im Objekt Cache befindet ist die Aufgabe für die Datenbank und den Hauptspeicher also nahezu identisch, wie die gemessenen Zeiten beweisen.
Bei t8 werden sämtliche Vorkommen des Buchstabens "I" im 100 Kilobyte großen Manualobjekt gezählt (Dabei werden 2632 Vorkommen gefunden). t9 testet, ob der Anfangsbuchstabe des Manuals mit dem Endbuchstaben übereinstimmt.

2.3.3 Queries

Die Queries beinhalten typische Datenbankanfragen die normalerweise in einer deklarativen Datenbanksprache ausgedrückt werden. Da die Datenbanksprache Object-SQL für das Versant DBMS noch nicht in einer endgültigen Version vorliegt, wurden hier diese Konstrukte konventionell mit C++ Funktionen simuliert.
Da bei allen hier aufgeführten Datenbankanfragen nur relativ wenig Objekte betrachtet werden, unterscheiden sich die gemessenen heiß-Zeiten der Datenbank nur gering von denen des Hauptspeicher-ADTs. Die folgende Tabelle zeigt alle gemessenen Zeiten; wesentliche neue Erkenntnisse können daraus jedoch nicht abgeleitet werden.

Tabelle 3: Queries

Benchmark	Versant kalt	Versant heiß	NIHCL	Trade-off (Versant / NIHCL)
q1	1,0793	0,0643	0,2617	0,26
q2	2,5490	0,0028	0,0005	5,60
q3	3,6593	0,0110	0,0023	4,78
q5	7,5709	0,0223	0,0270	0,83
q6	0,2542	0,0047	0,0398	0,12
q7	9,4740	0,0982	0,0184	5,34
i	0,2441	0,0040	0,0004	10,00
d	0,2442	0,0042	0,0004	10,50

Benchmark q1 Suche 10 zufällig ausgewählte atomic parts anhand ihrer id. Bei diesem Test ist die Versant Datenbank schneller, da das id-feld mit einem Index versehen ist (Hash-Tabelle) wohingegen beim Hauptspeicher-ADT eine lineare Suche durchgeführt wird.

Benchmarks q2, q3 und q7 Diese Datenbankanfragen sind sehr ähnlich. q2 erzeugt einen Zeitraum, sodaß das build date von etwa 1% aller atomic parts in diesen Zeitraum fallen. Query q2 verfährt ähnlich, nur wird hier ein Zeitraum betrachtet daß ca. 10% aller atomic parts gefunden werden. Query q7 schließlich sucht nach allen atomic parts. Bei all diesen Tests wird zu jedem gefundenen Objekt eine leere Funktion aufgerufen damit das Objekt auch wirklich referenziert wird.
Das Attribut build date ist mit einem Index versehen, sodaß die Datenbank zumindest bei den Anfragen q2 und q3 einen Vorteil hat.

Benchmark q5 q5 simuliert den Unix Befehl „make“ indem er base assemblies sucht, die nicht mehr „up to date“ sind. Verglichen wird dabei das build date des mit einem base assembly verbundenen composite part mit dem des base assemblies. Ist dieses größer als das des base assemblies so ist das base assembly ein Kandidat für „make“. Bei allen gefundenen Objekten wird eine leere Funktion aufgerufen damit die Objekte tatsächlich in das Objekt-Cache kopiert werden müssen. Die größere Geschwindigkeit der Datenbank beruht auf der Indizierung des build dates.

Benchmarks i und d Bei diesen Tests wird die Zeit zum Einfügen und Löschen von Objekten gemessen. Es werden jeweils 10 composite parts eingefügt bzw. gelöscht. Hierbei ist der Hauptspeicher-ADT etwa 10 mal schneller als die Datenbank.

2.3.4 Schlußfolgerungen

Der Vergleich der Geschwindigkeit einer Datenbank mit der eines Hauptspeicher-ADTs ist prinzipiell nicht einfach und führt häufig zu ungerechten Ergebnissen. Die hier gemessenen Zeiten können nur als grobe Anhaltspunkte aufgefasst werden, da sie von vielen Details in der Implementierung des Benchmarks abhängen die in realen Situationen eventuell anders gelöst werden.
Generell kann gesagt werden, daß bei einer Versant Datenbank der Erstzugriff auf Daten sehr viel Zeit in Anspruch nimmt. Die Zeiten für den ersten Zugriff auf Objekte ist nicht mit einem Hauptspeicher-ADT vergleichbar denn die bei diesem ADT benötigte Datenbasis muß ja auch erst erzeugt werden (durch Generierung oder Einlesen aus einer Datei). Dabei hängt es sehr stark von der Problemstellung ab, wie die Erzeugung der benötigten Daten im Hauptspeicher von statten geht.
Vergleichbar sind hingegen die Zeiten für den Zugriff auf Daten, die sich bereits im Hauptspeicher befinden. Hier scheint Versant eine sehr effektive Vorgehensweise anzuwenden, die den Datenbankoverhead klein hält. Werden allerdings viele Daten verwendet, so läuft das Objekt-Cache der Datenbank über und bei Zugriffen auf weitere Objekte müssen diese erst zeitaufwendig von der Festplatte nachgeladen werden. Dadurch laufen viele Programme die auf dem Hauptspeicher-ADT beruhen, vergleichsweise wesentlich schneller. Sollen also Datenintensivere Probleme gelöst werden, so ist es sicherlich sinnvoll, diese mit einem Hauptspeicher-ADT zu lösen. Die von dem Algorithmus benötigten Daten können vorher aus einer Datenbank ausgelesen werden und über eine Datei ausgetauscht werden.
Um Zugriffe auf Datenstrukturen in einer Datenbank zu beschleunigen ist es sinnvoll, Indices zu verwenden. Diese werden nach dem Anlegen automatisch, ohne weiteren Programmieraufwand, verwaltet.

2.4 GemStone

Es gibt verschiedene Ansätze um ein objektorientiertes Datenbankmodell umzusetzen und in eine Wirtssprache einzubetten. Neben den Prinzipien der Objektorientierung (Vererbung, Datenkapselung usw.) müssen grundlegende Datenbankkonzepte realisiert werden. Einen ausführlicheren Vergleich zwischen verschiedenen Datenbanksystemen (ObjectStore, Ontos, Gemstone und Versant) findet man in [KRR93].
In der folgenden Tabelle sind einige Aspekte des Versant DBMS im Vergleich zum GemStone[6] DBMS zusammengefaßt[7].

Tabelle 4: Vergleich Versant mit GemStone

	Versant	GemStone
Sprachinterface	C, C++, Smalltalk, OSQL	C, C++, Smalltalk
Vererbung	Mehrfach in C, C++ Einfach in Smalltalk	Einfach
Einbindung in Wirtssprache	Kein Unterschied zwischen persistenten und transistenten Objekten	Persistente Objekte werden speziell behandelt
Transaktionen	Lang, kurz Savepoint/undosavepoint	Kurz, nicht geschachtelt
Locks	Implizit, explizit Kurz, persistent	Explizit, Freigabe bei commit
Versionierung	Automatisch	n.V.

Sowohl auf Versant als auch auf GemStone Datenbanken kann von mehreren Wirtssprachen aus zugegriffen werden (siehe Tabelle). Betrachtet wurde hier insbesondere das Smalltalk Interface der GemStone Datenbank und bei Versant das C++ Interface. Ein wesentlicher Unterschied der beiden Datenbanksysteme ist der Umgang mit persistenten Objekten. Während bei GemStone persistente Objekte besonders gehandhabt werden müssen, kann bei Versant auf ein persistentes Objekt nach dem Anlegen genau wie auf ein normales C++ Objekt zugegriffen werden.
Martin Schütze beschreibt in seiner Diplomarbeit [Süt92] seine Erfahrungen mit dem Datenbanksystem GemStone. Insbesondere hat er GemStone daraufhin untersucht, ob es sich als Grundlage für eine VLSI-Entwurfsdatenbank eignet. Einige Probleme die beim Arbeiten mit GemStone auftraten lassen sich mit dem Versant-DBMS leichter lösen.
Ein Kritikpunkt an Gemstone war das unzureichende Transaktionsmanagement. Hier unterscheidet GemStone zwischen einem optimistischen Ansatz, bei dem keine Koordination zwischen des Transaktionen stattfindet, und einem pessimistischen Ansatz, bei dem sämtliche Objekte gesperrt werden. Versant verwendet beim Zugriff auf Objekte grundsätzlich Sperren, die allerdings in einer sehr feinen Granularität vergeben werden können (siehe 3.3.9). Soweit es die Semantik zuläßt, können mehrere Sperren auf dasselbe Objekt bestehen

6. GemStone ist ein kommerziell vertriebenes objektorientiertes Datenbanksystem der Firma Servio Logic.
7. Der Vergleich zwischen Versant und GemStone wurde ausgewählt, da die Einsatzmöglichkeiten von GemStone in der AG Zimmermann bereits untersucht wurden [Süt92].

(z.B. mehrere Lesesperren oder Snapshot-Locks). Außerdem werden grundsätzlich nur einzelne Objekte gesperrt, das heißt beim Sperren Einer Menge können weiterhin einzelne Objekte dieser Menge von anderen Programmen gesperrt werden.
Zusätzlich zu diesem einfachen Transaktionsmanagement bietet Versant noch lange Transaktionen an. Dadurch kann auf einzelne Objekte über einen langen Zeitraum und von verschiedenen Programmen aus zugegriffen werden. Dies ist insbesondere bei CAD-Anwendungen von Bedeutung, da diese häufig Datenbanken zum Entwurf von Ojekten benutzen. Mit Hilfe von langen Transaktionen kann ein Objekt erst vollständig entworfen werden, bevor es zum allgemeinen Zugriff freigegeben wird.
Lange Transaktionen können mit der automatischen Versionierung von Versant kombiniert werden, sodaß Objekte die nach einer langen Transaktion in die Datenbank zurückgeschrieben werden, eine neue Versionsnummer erhalten. So kann während der Bearbeitung des Objektes auch noch mit alten Versionen desselben weitergearbeitet werden.
Durch dieses flexible Transaktionsmanagement scheint Versant zum Einsatz als Entwurfsdatenbank gegenüber GemStone geeigneter zu sein.

Kapitel 3

Benutzung des Versant-ADTs

3.1 Allgemeines

Das folgende Kapitel enthält Einzelheiten über die Implementierung des Versant-ADTs und zeigt, wie dieser benutzt wird. Voraussetzung für die Anwendung des ADTs ist ein C++ Compiler (derzeit Objectcenter C++ Version 1.1.0) und das Versant-DBMS in der Version 1.7.3. Andere Versionen der Datenbank müßten auch mit dem ADT zusammenarbeiten, wurden jedoch nicht getestet. Ebenso lassen sich die Programme auch mit anderen Compilern übersetzen die kompatibel zu cfront 2.1 von AT&T sind. Die Datenbank läuft auf einem Computer der Marke SUN II unter dem Betriebssystem SunOS 4.1.3.

Nachdem zunächst auf die Erzeugung der ADTs und auf den generellen Aufbau von Benutzerprogrammen eingegangen wird, beschäftigt sich der zweite Teil des Kapitels mit einigen Konzepten des Versant-DBMS und wie diese von C++-Programmen aus genutzt werden können.

3.2 Benutzung des ADTs

3.2.1 Generierung eines ADTs

Die Generierung des Versant-ADTs erfolgt in gleicher Weise wie die des C++-ADTs (siehe [Wer94] Kapitel 6). Dazu wird im MOOSE der Menüpunkt *File*, *Define/Generate Application* aufgerufen. Daraufhin öffnet sich ein Fenster in dem die zu generierende Applikation ausgewählt oder angelegt werden kann. Neu hinzugekommen ist nur ein Knopf um auszuwählen, daß ein Versant-ADT generiert werden soll. Um zusätzlich auch noch Funktionen zum Datenaustausch (siehe Kapitel 4) zu erzeugen, muß der Knopf *generate I/O functions* angewählt werden. Zuletzt wird mit *Compile* der ADT erzeugt.
Es werden dann sämtliche benötigten Quelltexte generiert und zusätzlich werden ein Shell-Script *CALL_ME* zum Compilieren des ADTs und ein Shell-Script *UPDATE_DB* zum Anlegen bzw. Ändern einer Versant Datenbank erzeugt.

CALL_ME Das Shell-Script CALL_ME dient dazu, die generierten Quelltexte zu compilieren und in einer Bibliothek zusammenzubinden. Dabei werden auch Schemabeschreibungen für eine Versant Datenbank erzeugt.
Nach dem Generieren der Quelltexte muß dieses Shell-Script (im *sources* Verzeichnis des ADTs) aufgerufen werden um den ADT als Programmbibliothek nutzen zu können.

UPDATE_DB Das Shell-Script *UPDATE_DB* hilft nun beim Anlegen einer Datenbank. Als Parameter wird der Name einer Datenbank übergeben. Existiert die Datenbank noch nicht (im lokalen Netzwerk), so wird eine neue private Datenbank angelegt und richtig für die Benutzung des ADTs konfiguriert. Ansonsten werden nur die Änderungen in die schon bestehende Datenbank übernommen. Soll eine öffentlich zugängliche Datenbank (group database) erzeugt werden, so muß der Parameter -g mit an das Shell-Script übergeben werden. Beim Aufruf von *UPDATE_DB* ohne Parameter wird eine kurzer Hilfetext ausgegeben.

Schemaevolution Beim Ändern der Schemata einer bereits existierenden Datenbank wird zunächst versucht, die Datenbank inkrementell, ohne Verlust der darin enthaltenen Daten, zu erweitern. Gelingt dieses nicht[8] so werden nach einer Sicherheitsabfrage einzelne Klassenbeschreibungen sowie alle Instanzen dieser Klassen gelöscht und es wird eine neue Schemabeschreibung der Klassen angelegt.
Durch das Löschen einzelner Klassen kann die Datenbank in einen inkonsistenten Zustand gebracht werden da Referenzen auf nicht mehr existierende Objekte auftreten können! Deshalb sollte bei einer derartigen Änderung der Datenbankschemata der gesamte Inhalt der Datenbank gelöscht werden. Dieses kann durch den Befehl "dropinst -D <DBNAME> PVirtual" geschehen. <DBNAME> beschreibt dabei den Namen der geänderten Datenbank.

3.2.2 Konfiguration einer Datenbank

Nach dem Anlegen einer Datenbank kann diese noch an das zu erwartende Datenaufkommen angepasst werden. Für jede Datenbank werden Datenbankprofile angelegt. Diese sind als lesbarer Text abgespeichert und können mit einem Texteditor bearbeitet werden. Das globale Verhalten einer Datenbank ist in der Datei *profile.be* beschrieben. Diese befindet sich im gleichen Verzeichnis wie die Datenbank selbst[9]. In dieser Datei werden Maximalgrößen für Logdateien und Puffer eingestellt (siehe Beispiel 1).

Beispiel 1: `Datei profile.be der Beispiel-Datenbank oo7db (siehe Anhang A)`

```
# Init Section #
# This section will be processed by CREATEDB and the back end init process
# when locating the system volume.
sysvol                  20M     system
plogvol                 2M      physical.log
llogvol                 8M      logical.log
extent_size             4

# Configuration Section #
# This section will be processed by the back end init process to
# allocate resources.
```

8. Es ist nur möglich Schemabeschreibungen neuer Klassen hinzuzufügen und bei Vererbung einfache Änderungen an den Blättern der Klassenhierarchie durchzuführen.
9. Sämtliche Datenbanken werden in Unterverzeichnissen des Datenbank-Verzeichnisses abgelegt. Den Pfadnamen des Datenbank-Verzeichnisses erhält man mit dem Befehl *oscp -i*.

```
# Personal database may have these flags turned OFF to get speed.
# Group database should have these flags turned ON to maintain
# integrity.
llog_enabled on
plog_enabled on
locking on

llog_buf_size 512k
plog_buf_size 1024k

heap_size 1024k
max_page_buffs 256
commit_flush off
transaction 5
obj_cnt 2000
lock_wait_timeout 60
```

Lokale Einstellungen, die einzelne Anfragen an eine Datenbank betreffen, werden für jeden Benutzer einzeln eingestellt. Dazu wird im Heimverzeichnis des Benutzers ein Verzeichnis *.osc* angelegt, in dem für jede Datenbank eine Konfigurationsdatei mit dem Namen der Datenbank existiert. Dort können Einstellungen über (lokale) Cache-Größen und die maximale Anzahl von Objekten gemacht werden (siehe Beispiel 2)

Beispiel 2: `Konfigurationsdatei ~/.osc/oo7db`

```
# Configuration Section
#
# This section will be processed only when creating the ODA,
# i.e., the first time the front end is being brought up.

cache_size                  2048k
heap_size                   512k
max_objects                 10240

# Implementation Section
#
# This section will be processed every time a session is begun.
# The aliases will be kept throught out the entire session.
# The databases will be disconnected either by calling disconnect
# within the session or at end of the session.
# N.B., The access mode is not implemented in the first release.

#alias                      db1     db1@station1
#alias                      db2     db2@hostname
#alias                      db3     db3@osc
#alias                      localdb myDatabase

#connect                    db1     READ_WRITE
#connect                    db2     READ_ONLY
#connect                    db3     WRITE_ONLY
```

Nähere Angaben zu den Konfigurationsdateien befinden sich im Versant-Referenzhandbuch.

3.2.3 Übersetzen eigener Applikationsprogramme

Zum Compilieren eigener Programme die den Versant-ADT benutzen, muß ein C++ Compiler mit AT&T front end installiert sein (Derzeit wird der Object-Center C++-Compiler Version 1.1.0 unterstützt). Zunächst wird die Bibliothek mit Hilfe des Shell-Scripts CALL_ME übersetzt und installiert.
Um nun eigene Programme zu compilieren, muß die dadurch erzeugte Bibliothek (lib*SchemaName.Vers.Conf*_sun.a im Verzeichnis *SchemaName*/lib) beim Linken eingebunden werden. Zusätzlich müssen noch die Versant Bibliotheken (libvnih.a, libcxxcls.a und liboscfe.a im Verzeichnis /home/versant/lib) und die Bibliotheken zur Fehlerbehandlung (/zimmermann/tools/common_c/lib/libhandleerror_sun.a und /usr/local/lib/gcc-lib/sparc-sun-sunos4.1.2/2.3.3/libgcc.a) eingebunden werden. Im Anhang A ist ein Imakefile angegeben, welches als Schablone zum Übersetzen von Programmen dienen kann. Im Imakefile braucht nur noch angegeben zu werden, welchen ADT das zu übersetzende Programm benutzen will (Hauptspeicher oder Versant) und wie das Applikationsprogramm letztendlich heißt. Ein Aufruf des Unix Kommandos *xmkmf* generiert daraufhin ein passendes Makefile. Mit *make depend; make* wird dann das Programm compiliert.

3.3 Applikationsprogramme

3.3.1 Eigenschaften des Versant-ADTs

Prinzipiell verhält sich der Versant-ADT genau so wie der C++-ADT. Das bedeutet, daß (fast) alle Funktionen des C++-ADTs auch in ähnlicher Art und Weise beim Versant-ADT implementiert sind.
Konzeptionell bestehen jedoch zwischen der Speicherung der Daten im Hauptspeicher wie beim C++-ADT und der Datenhaltung in einer Datenbank einige Unterschiede. So unterscheidet Versant zwischen transistenten (also lokal im Hauptspeicher gehaltenen) und persistenten (in einer Datenbank dauerhaft abgelegten) Objekten. Persistente Objekte werden mit einem speziellen Befehl erzeugt (new Persistent, siehe 3.3.3) und können dann genau wie lokale Objekte benutzt werden. Aufpassen muß man dabei, daß sich die Datenbank am Ende einer Transaktion in einem konsistenten Zustand befindet. Das bedeutet insbesondere, daß ein Objekt in der Datenbank nur Referenzen auf andere persistente Objekte haben darf! Zeiger von persistenten Objekten auf lokale Objekte sind nicht zulässig.
Ein weiterer Unterschied zwischen dem C++-ADT und dem Versant-ADT besteht im Zugriff auf Objekte. In C++ wird im allgemeinen auf ein Objekt mit einem Zeiger (Pointer) zugegriffen. Dieser Zeiger zeigt auf die Speicheradesse, ab der das Objekt im Hauptspeicher abgelegt ist und ermöglicht so einen schnellen Zugriff auf das Objekt. Wenn ein Objekt aus einer Datenbank angesprochen werden soll, so ist ein Zugriff über Pointer nicht unbedingt möglich, da sich das Objekt eventuell gar nicht im Hauptspeicher befindet. Um jedoch in ähnlich einfacher Weise auf Objekte in einer Datenbank zuzugreifen, gibt es beim Versant-ADT sogenannte Links (siehe 3.3.4). Diese zeigen immer auf ein Objekt unabhängig davon, ob es sich in der Datenbank oder im Hauptspeicher befindet. Der Umgang mit Links ist ähnlich einfach wie der mit Pointern.

3.3.2 Aufbau der Applikationsprogramme

Zugriffe auf die Datenbank müssen innerhalb einer DB-Sitzung (session) geschehen. Dazu muß das Applikationsprogramm folgenden Aufbau haben:

Beispiel 3

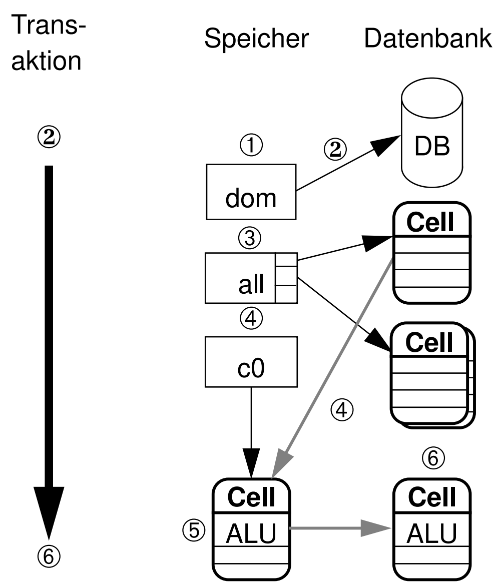
```
…
#include "OO7VersantADT.1.1/ins/allClasses.h"
#define DBNAME "oo7db"
#define SESSIONNAME NULL
…
main()
{
  …
  ::dom = new PDOM;
  ::dom->beginsession(DBNAME, SESSIONNAME);
  // beginsession(Datenbankname, Name der Sitzung)
  {
     … Datenbankzugriffe ect. …
     ::dom->commit();
  }
  ::dom->endsession();
  ::dom->exit(0);
}
```

Transaktionen und Änderungen der Datenbank werden mit *::dom->commit()* aktualisiert. Wird statt dessen ein *::dom->rollback()* ausgeführt, so werden alle Änderungen in der Datenbank, die während der aktuellen Transaktion stattgefunden haben, verworfen. Wird am Ende einer kurzen Sitzung weder commit noch rollback aufgerufen, so wird implizit ein rollback durchgeführt.
DBNAME ist der Name einer Versant-Datenbank; diese muß zuvor mit *makedb DBNAME* vorbereitet, und mit *createdb DBNAME* angelegt sein. Zusätzlich müssen mit sch2db die Datenbankschemata in die Datenbank eingetragen sein (Dieses wird automatisch von dem shell-script UPDATE_DB erledigt siehe 3.2.1).
SESSIONNAME bezeichnet den Namen der Sitzung. Dieser ist bei kurzen Transaktionen in der Regel NULL.

Abb. 5 Datenbanksitzung

Beispielprogramm:

```
  main()
① { ::dom = new PDOM;
②   ::dom->beginsession
       (DBNAME, SESSIONNAME);
③   { LinkVstr<Cell> all;
      all = Cell::allInstances();
④     Cell *c0 = all[0];
⑤     c0->setName("ALU");
⑥     ::dom->commit();
    }
    ::dom->endsession();
    ::dom->exit(0);
  }
```



In Abb. 5 wird beispielhaft eine Datenbanksitzung beschrieben. Um auf die Datenbank zugreifen zu können, muß zunächst eine Verbindung zu dieser Aufgebaut werden. Das geschieht mit Hilfe einer Sitzung (Session). In dem Beispiel wird in Zeile ② eine Sitzung gestartet. Dadurch wird eine Verbindung zu der Datenbank DBNAME aufgebaut. Gleichzeitig startet eine Transaktion (siehe 3.3.8). In der dritten Zeile erfolgt nun eine Datenbankanfrage. Diese liefert als Ergebnis eine Menge von Zeigern auf alle Objekte in der Datenbank die vom Typ Cell sind. Wird auf ein Objekt zugegriffen (Zeile ④) so wird dieses automatisch in den Hauptspeicher kopiert. Dort kann es dann verändert werden. Durch das commit in Zeile ⑥ stoppt die Transaktion und die Änderungen werden übernommen. Ein Aufruf der Funktion *endsession()* beendet die Verbindung zur Datenbank.

3.3.3 Anlegen neuer Objekte

Die Versant Datenbank unterscheidet zwischen persistenten (dauerhaften) und transistenten (temporären) Objekten. Grundsätzlich werden diese gleich angesprochen, nur mit dem Unterschied, daß Änderungen an persistenten Objekten (nach erfolgreichem commit) in der Datenbank sichtbar werden. Um ein Objekt persistent in einer Versant Datenbank anzulegen, muß es mit einem speziellem Befehl erzeugt werden. Statt dem bei C++-Objekten üblichen *new classname* wird ein neues Objekt mit *new Persistent classname* angelegt. Danach kann auf das Objekt wie auf ein ganz normales C++ Objekt zugegriffen werden. Objekte, die nach wie vor nur mit *new classname* angelegt werden, sind lokale Objekte und werden nicht in der Datenbank abgelegt. Durch den Befehl *persistentCopy()* kann eine persistente Kopie eines transistentes Objektes angelegt werden.
Referenzen von persistenten Objekten auf lokale Objekte sind nicht erlaubt und können zu Programmabstürzen führen!

Beispiel 4
```
// Neues Modul anlegen
ModuleClass *mod = new Persistent ModuleClass();
mod->setid(25);

// Neues Manual anlegen
ManualClass *man = new Persistent ManualClass();
man->settitle(new Persistent String40Class("Module_25"));

// Dem Modul mod wird das Manual man zugewiesen.
mod ->conManualOfModule(man);

// Transistentes Objekt
ModuleClass *mod2 = new ModuleClass();
// Persistent kopieren
ModuleClass *mod3 = (ModuleClass*) mod2->persistentCopy();
```

Versant speichert alle Objekte auf die momentan zugegriffen wird in einem Zwischenspeicher (Cache) ab. Dadurch ist es möglich, mit Pointern direkt auf diese Objekte zuzugreifen. Nach einem *commit()* oder *rollback()* wird dieser Cache Speicher automatisch gelöscht. Dadurch werden alle Pointer und Links auf Objekte aus der Datenbank ungültig! Das heißt, daß alle Objekte vor einem erneuten Zugriff erst wieder aus der Datenbank ins Cache geladen werden müssen.

3.3.4 Zugriff auf Objekte

Die Attribute der einzelnen Objekte werden beim Versant-ADT genau wie beim C++-ADT erzeugt. Ausnahme: komplexe Datentypen werden nicht mehr als Zeiger auf andere Objekte realisiert sondern als (Versant-)Links.

Beispiel 5
```
String40Class*_title;
wird (beim Versant-ADT) zu:
Link<String40Class> _title;
```

Auf Links kann in den meisten Fällen genau wie auf Pointer zugegriffen werden. Ansonsten müssen sie explizit mit einem cast zu einem Pointer verwandelt werden.

Beispiel 6
```
Link<String40Class> title = new String40ClassClass();
title->setLength(5);
title->setField ("Title");
// aber:
delete (String40Class *) title;
// oder: title->deleteobj();
```

Die Umwandlung von Pointern in Links ist notwendig, damit Referenzen auf Klassen und Objekte konsistent in der Datenbank abgespeichert werden können. Ein Link zeigt zunächst immer auf ein Objekt in der Datenbank. Wird auf dieses Objekt zugegriffen, so wird es in den Arbeitsspeicher geladen und der Link verweist nun auf die Kopie im Speicher. Dadurch wird einerseits gewährleistet, daß über Links die Datenbankobjekte immer angesprochen werden können, und andererseits kann auf die im Hauptspeicher befindlichen Objekte schnell zugegriffen werden.

3.3.5 BiLinks

Relationen werden beim Versant-ADT als BiLinks repräsentiert. Die Klasse BiLink beinhaltet eine Verbindung von einem Objekt zu einem anderen Objekt, das aus einer anderen Klasse sein kann. Dabei werden Veränderungen einer Variablen vom Typ BiLink automatisch auch in der verbundenen Instanzenvariablen des anderen Objektes durchgeführt[10]. n..1 Relationen werden auf *BiLink*s, und n..m Relationen auf *BiLinkVstr*s abgebildet. Ein BiLinkVstr verhält sich ähnlich wie ein NIHCL Set oder ein LinkVstr (siehe 3.3.6).

Beispiel 7:
```
class ModuleClass
{
  …
  public:
  Link<String10Class> _title;
  BiLinkVstr<AssemblyClass, _ModuleOfAssemby> _AssembliesOfModule;
  …
}

class AssemblyClass
{
```

10. Dadurch können beim Versant-ADT die Methoden *conIntern…()* und *dcoIntern…()* entfallen (siehe [Wer94], Kapitel 2.2).

```
  …
  public:
  BiLink<ModuleClass, _AssembliesOfModule> _ModuleOfAssembly;
  …
}

main()
{
  ModuleClass *mod1 = new Persistent ModuleClass();
  ModuleClass *mod2 = new Persistent ModuleClass();
  AssemblyClass *ass1 = new Persistent AssemblyClass();

  // ass1 mit mod1 verbinden. Die Rückverbindung von mod1 zu ass1
  // wird automatisch mitangelegt.
  ass1->conModuleOfAssembly(mod1);
  // entspricht ass1->_ModuleOfAssembly = mod1

  // Verbindung ass1↔mod1 löschen und ass1 mit mod2 verbinden
  ass1->conModuleOfAssembly(mod2);
  delete mod2; // Löscht automatisch Verbindung ass1↔mod2
}
```

Durch die Verwendung von BiLinks können Verweise auf andere Objekte konsistent in einer Datenbank abgelegt werden, was mit Pointern nicht möglich ist, da diese nach einem commit ihre Gültigkeit verlieren.

3.3.6 Set vs. Vstr

Bei Zugriffen auf eine Datenbank liefert der Versant-ADT eine Menge von Objekten in einer Variablen vom Typ Vstr zurück. Dieser Variablentyp wird von der Versant Klassenbibliothek vorgegeben wodurch ein schneller und einfacher Zugriff auf Objektmengen gewährleistet wird. Da der C++-ADT im Gegensatz hierzu Variablen vom Typ Set benutzt, sind im folgenden die Unterschiede zwischen Sets und Vstr aufgelistet.
Konzeptionell unterscheiden sich Sets von Vstrs dadurch, daß in einer Variablen vom Typ Set jedes Objekt nur einmal vorkommt. Ein Set hat also die Semantik wie eine (ungeordnete) Menge im mathematischen Sinn. Ein Vstr verhält sich dahingegen eher wie ein Feld (Array). Die folgende Tabelle (Tabelle 5) gibt darüber Aufschluß, wie auf Sets und Vstrs zugegriffen werden kann.

Tabelle 5: Vergleich von Sets mit Vstrs

Operation	Set	Vstr
Anlegen	m = new Set(unsigned size) Größe wird automatisch angepasst	m = new Vstr<Cell> (unsigned size) Größe wird automatisch angepasst
Abfragen der Größe	unsigned m.size()	int m.size()
Zugriff auf ein Objekt	CLObject *m.at(int index)	Cell& m[int index]
Objekt hinzufügen	m.add(CLObject&)	m.add(Cell&)
Sonstiges	Mengenoperationen &, |, -	append(Vstr&)
Iterator möglich	ja	nein (for-Schleife)

Vstrs sind als C++ Templates definiert was bedeutet, daß ein Vstr immer nur eine bestimmte Menge von Objekten beinhalten kann. Ein Vstr<Cell> beinhaltet also nur Objekte der Klasse Cell und deren abgeleiteten Klassen. Ein NIHCL-Set hingegen beinhaltet immer Objekte vom Typ CLObject, der Basisklasse der NIH-Bibliothek. Daher muß beim Zugriff auf ein Objekt in einem Set dieses explizit durch ein castdown in den gewünschten Typ umgewandelt werden.

Beispiel 8

```
NIHCL:
Set Zellen;
Cell c1, c2, c3;
Zellen.add(c1);
Zellen.add(c2);
c3 = Cell::castdown(Zellen.at(1));

Versant:
Vstr<Cell> Zellen;
Cell c1, c2, c3;
Zellen.add(c1);
Zellen.add(c2);
c3 = Zellen[1];
```

Um bei Versant ein Vstr zu erhalten, welches Objekte sämtlicher Klassen beinhalten kann, kann man ein Vstr<PHandle> verwenden. Der Datentyp PHandle dient in der Versant-Bibliothek als Zugriffsmöglichkeit auf beliebige Datenbankobjekte. Eine Variable vom Typ PHandle kann also auf jedes Datenbankobjekt zeigen. Soll auf dieses Objekt zugegriffen werden, so kann mit der Funktion PHandleToLink<*Klassenname*> ein Zeiger auf das Objekt erzeugt werden. *Klassenname* bezeichnet dabei die Klasse (oder eine Oberklasse) des entsprechenden Objektes.

Bedingt durch die Unterschiedliche Implementierung muß bei der Programmierung von Applikationsprogrammen besondere Rücksicht beim Zugriff auf Mengen (insbesondere bei 1:n-Relationen) genommen werden. Unterschiedlich ist vor Allem der Zugriff auf einzelne Objekte einer Menge. Da eine Menge am häufigsten mit einem Iterator durchlaufen wird, stellen die vom MOOSE generierten Quelltexte eine vereinheitlichte Schnittstelle auf Iteratoren über Mengen zur Verfügung. Dazu werden die beiden Funktionen LOOP und NESTED_LOOP erzeugt, mit denen alle Objekte einer Menge durchlaufen werden können (siehe Beispiel 9). Die Funktion LOOP erhält ans Parameter die Menge, die durchlaufen werden soll, den Klassennamen der Objekte die in der Menge enthalten sind und einen Variablennamen. Daraufhin wird eine Variable angelegt, die sukzessive Zeiger auf die einzelnen Objekte in der Menge annimmt. Sollen Iteratoren geschachtelt werden, so muß die Funktion NESTED_LOOP verwendet werden. Sie erhält als zusätzliche Parameter einen weiteren Variablennamen (der nicht weiter verwendet werden kann) und eine Nummer für die Schachtlungstiefe.

Beispiel 9

```
// alle Module ausgeben
LOOP (ModuleClass::allInstances(), ModuleClass, mod)
    mod->printOn();
ENDLOOP

geschachtelte Iteratoren:
```

```
// alle Pins sämtlicher Zellen ausgeben
NESTED_LOOP(CellClass::allInstances(), CellClass, zelle,
i1, 1)
    zelle->printOn();
    NESTED_LOOP(zelle->pinsOfCell(), PinClass, pin, i2, 2)
       pin->printOn();
    NESTED_ENDLOOP
NESTED_ENDLOOP
```

3.3.7 Datenbankanfragen

Datenbankanfragen geschehen im allgemeinen mit dem select Befehl. Dieser ist in der Versant Klasse PClassObj definiert. Als Parameter erhält er die Datenbank, die durchsucht werden soll, ein Flag, ob die Suche rekursiv sein soll und ein Prädikat, das das Suchkriterium angibt.

Beispiel 10

```
// Suche alle Module mit id == 25 oder id > 50
PPredicate pred (PAttribute("ModuleClass::_id") == 25 ||
                 PAttribute("ModuleClass::_id") > 50);
// Suche beginnen
LinkVstr<ModuleClass> erg =
 PClassObj<ModuleClass>.select(
 NULL,      // standard Datenbank
 FALSE,     // nicht rekursiv
pred);

// alle gefundenen Module ausgeben
for (int i=0; i<erg.size(); i++)
 erg[i]->dumpOn();
```

Um Suchkriterien anzugeben, ist in der Versant-Bibliothek die Klasse PPredicate definiert. Damit können Suchprädikate in der Form "Instanzvariable Vergleichsoperator Wert" angegeben werden. Als Vergleichsoperator kann ==, !=, >, <, >= und <= angegeben werden. Prädikate lassen sich mit || (logisches oder) und && (logisches und) verknüpfen (siehe Beispiel 10 und [Ver92b]).

Komplexe Datenbankanfragen lassen sich durch mehrere select-Befehle und anschließende Mengenverknüpfungen realisieren. Eine weitere Möglichkeit komplexe Anfragen an eine Datenbank zu stellen, bietet Versant (ab der Version 2.0) in Form von OSQL, einer objektorientierten Erweiterung der Datenbanksprache SQL, an. Damit können, ausgehend von C++, sehr umfangreiche und komplizierte Datenbankanfragen formuliert werden. Die OSQL-Spracherweiterung liegt allerdings in der hier benutzten Version des Versant DBMS nur als Testversion vor und wurde daher nicht verwendet.

3.3.8 Transaktionskonzept

Sämtliche Operationen auf einer Datenbank werden in Transaktionen geschachtelt. Eine Transaktion gewährleistet, daß alle Aktionen innerhalb der Transaktion atomar sind, daß heißt es werden entweder alle Datenbankoperationen durchgeführt oder keine (siehe [Här86]).
Das Versant-DBMS unterstützt neben den, bei relationalen Datenbanksystemen üblichen, kurzen Transaktionen, auch lange Transaktionen die über mehrere Programme verteilt sein können. Dadurch ist es möglich, Objekte aus einer Datenbank über einen längeren Zeitraum hinweg zu bearbeiten und vor dem Zugriff von anderen Benutzern zu sperren.

Bevor ein Programm auf eine Datenbank zugreifen kann, muß zunächst eine Transaktion gestartet werden. Dies geschieht mit dem Befehl *beginsession* (siehe [Ver92b] S. 3 und Abb. 5). Wird dabei ein Name für die Transaktion mit angegeben, so startet automatisch eine lange Transaktion. Solange diese nicht explizit mit dem Befehl *endsession*[11] beendet wurde, kann sie von anderen Programmen unter Angabe des Namens wieder aufgenommen werden. Wenn am Ende einer Transaktion ein commit durchgeführt wird, so sind alle Änderungen, die während der Transaktion durchgeführt wurden, in der Datenbank nach außen sichtbar. Bei einem rollback wird die Datenbank wieder in den Zustand versetzt, den sie vor dem Start der Transaktion hatte.
Transaktionen können nicht geschachtelt werden. Das Konzept der geschachtelten Transaktionen kann allerdings mit Hilfe von savepoints realisiert werden. An jeder beliebigen Stelle innerhalb einer Transaktion kann mit dem Befehl savepoint der Datenbankzustand gesichert werden. Durch ein undosavepoint kann dann der zuletzt gesicherte Zustand wiederhergestellt werden. Erst am Ende einer Transaktion werden die gesicherten Datenbankzustände wieder gelöscht.

3.3.9 Sperren

Das Versant-DBMS ist als Mehrbenutzersystem konzipiert. Um Konflikte beim Zugriff von mehreren Benutzern auf dasselbe Objekt zu vermeiden werden Sperren (Locks) vergeben. Versant sperrt dabei individuell jedes einzelne Objekt je nach dem, welche Operation durchgeführt werden soll. Sperren werden in der Regel implizit vergeben, können aber auch explizit angefordert werden.

Versant vergibt vier unterschiedliche Arten von Sperren. Mit einem Read-Lock kann ein Datenbankobjekt gelesen werden. Dabei ist es auch erlaubt, daß mehrere Benutzer dasselbe Objekt gleichzeitig lesen.
Durch ein Update-Lock wird ausgedrückt, daß ein Objekt zunächst nur gelesen werden soll, später aber eventuell auch verändert wird. Ein Update Lock kann immer nur maximal ein Benutzer (für ein Objekt) bekommen; es ist allerdings möglich, daß noch andere Benutzer ein Read-Lock auf das Objekt vergeben haben. Ist ein Objekt mit einem Update-Lock gesperrt, so kann sehr schnell ein Write-Lock angefordert werden ohne daß das Objekt zwischenzeitlich von anderen Benutzern geändert werden kann (wird zuerst nur ein Read-Lock angefordert, so muß dieses bei Anforderung eines Write-Locks zunächst zurückgegeben werden um Deadlock-Situationen zu vermeiden).
Ist nun ein Objekt mit einem Write-Lock gesperrt, so kann es beliebig verändert werden. Ein Write-Lock wird nur vergeben, wenn keine anderen Read- oder Update-Locks das Objekt sperren.
Zusätzlich zu diesen Sperren gibt es noch ein Snapshot- oder No-Lock. Ein Snapshot-Lock kann jederzeit angefordert werden und berechtigt zum Lesen des Objektes. Der Unterschied zum Read-Lock besteht darin, daß nicht gewährleistet ist, ob das gelesene Objekt in einem aktuellen und konsistenten Zustand ist; es ist nämlich möglich, daß ein mit einem Snapshot-Lock belegtes Objekt gelesen wird während ein anderer Benutzer dasselbe Objekt verändert. Ein Snapshot-Lock eignet sich also nur für statistische Anfragen an eine Datenbank. Bei einer Datenbankanfrage der Art „Berechne die durchschnittliche Zahl der Anschlußpins aller Zellen“ ist es meist unerheblich, ob in dem Moment der Anfrage gerade bei einer Zelle die Zahl der Anschlußpins verän-

11. Mit entsprechenden Parametern, siehe Beispiel 14

dert wird. Stattdessen ist es wichtig, daß die Anfrage schnell bearbeitet wird. Würde man sämtliche Zellen mit einem Read-Lock zum Lesen anfordern, so erhält man zwar ein korrektes Ergebnis, muß dafür aber warten, bis alle Schreibvorgänge auf Zellen beendet sind. Mit einem Snapshot-Lock erhält man ein „annähernd“ richtiges Ergebnis ohne auf andere Benutzer warten zu müssen.

Tabelle 6: Blockierungen

	Snapshot Lock	Read Lock	Update Lock	Write Lock
Snapshot Lock	✔	✔	✔	✔
Read Lock	✔	✔	✔	✕
Update Lock	✔	✔	✕	✕
Write Lock	✔	✕	✕	✕

In Tabelle 6: ist aufgeführt, wie sich die verschiedenen Sperren gegenseitig blockieren. Ein "✔" bedeutet, daß sich zwei Sperren auf dasselbe Objekt nicht blockieren, ein "✕" zeigt an, daß die beiden entsprechenden Sperren nicht gleichzeitig existieren können. Wurde zum Beispiel bereits ein Update-Lock für ein bestimmtes Objekt vergeben, so kann kein Write-Lock mehr vergeben werden.

Sperren werden vom Versant-DBMS implizit oder explizit vergeben. Die implizite Vergabe wird über ein Standard Lock geregelt. Mit dem Befehl *setDefaultLock()* (siehe 3.4.3) kann eine Standardsperre für eine Datenbank vergeben werden. Jedes Objekt auf das zugegriffen wird, wird mit dieser Sperre belegt. Weiterhin werden bei Änderungen an Objekten automatisch Sperren vergeben. Der Versant-ADT ruft bei einer Änderung an einem Objekt die Funktion *dirty()* auf. Dadurch wird ein Write-Lock angefordert und gleichzeitig wird signalisiert, daß sich das Objekt verändert hat (der Zustand eines Objektes kann mit der Funktion *isdirty()* angefragt werden).
Zur expliziten Anforderung von Sperren können die Befehle *acquireslock()* und *acquireplock()* herangezogen werden (siehe 3.4.4). Unterschieden wird dabei zwischen kurzen Sperren (Short-Lock) und langen Sperren (Persistent-Lock). Kurze Sperren überdauern nur die Zeit bis zum nächsten commit oder rollback, wohingegen persistente Sperren während einer langen Transaktion (, die sich über mehrere Prozesse verteilen kann,) bestehen bleiben.

3.4 Unterschiede zum C++-ADT

Der Rest dieses Kapitels beschäftigt sich mit C++ Funktionen, die für die Programmierung von Applikationen mit dem Versant-ADT wichtig seien können. Für eine genauere Beschreibung dieser Befehle sei auf das Versant Manual [Ver92b] verwiesen.

3.4.1 Unterschiedliche Funktionen

allInstances () Die Methode *allInstances()*; liefert ein LinkVstr<…Class> zurück, welches aus einer Datenbankanfrage herrührt (Die Instanzvariable allInstances entfällt). Der LinkVstr ist eine Menge von Zeigern (Links) auf Objekte (siehe auch 3.3.6).

Beispiel 11
```
LinkVstr<ModuleClass> all = ModuleClass::allInstances();
for (int i=0; i<all.size(); i++)
{all[i]->printOn();
}
```

shallowCopy () Die Funktion *shallowCopy()* ist im Versant-ADT mit einer erweiterten Semantik implementiert. Wird sie auf ein transistentes Objekt angewendet, so wird auch nur eine transistente Kopie des Objektes angelegt; bei einem persistenten Objekt wird eine persistente Kopie erzeugt. Um aus einem transistenten Objekt ein persistentes zu machen existiert die Funktion *persistentCopy()* (siehe Beispiel 12).

3.4.2 Nicht implementierte Funktionen

deepCopy () Die Funktion *deepCopy()* wird von der Versant Bibliothek nicht unterstützt und ist folglich auch in dem generierten ADT nicht implementiert.

storeOn () Ebenso ist die NIHCL Funktion *storeOn()* nicht im Versant-ADT implementiert, da sie davon ausgeht, daß in jeder (Versant-) Klasse einige Funktionen zum Abspeichern und Einlesen von Objekten vorhanden sind; diese jedoch nicht in die Versant Klassen übernommen worden sind. Die automatisch generierten Funktionen zum Datenaustausch (siehe 4.3) machen diese Funktion zudem auch überflüssig.

3.4.3 Zusätzliche Versant Funktionen

isPersistent () Mit dieser Funktion kann festgestellt werden, ob ein Objekt persistent, also in einer Datenbank dauerhaft abgelegt, oder nur lokal im Hauptspeicher befindlich (transistent) ist (siehe Beispiel 12).

persistentCopy () Der Befehl persistentCopy legt eine persistente Kopie eines Objektes an. Dabei werden (ähnlich wie bei deepCopy) auch alle referenzierten Objekte kopiert.

Beispiel 12
```
#include "OO7VersantADT.1.1/ins/allClasses.h"

main()
{
  ::dom = new PDOM;
  ::dom->beginsession("oo7db", NULL);
  {
     ManualClass *temp, *pers;
     ModuleClass *mod;

     temp = new ManualClass(50);
     mod = new Persistent ModuleClass();
     mod->conManualOfModule(temp);
     if (temp->isPersistent())
        cout << "Manual ist persistent" << endl;
     if (mod->isPersistent())
```

```
         cout << "Modul ist persistent" << endl;

      pers = (ManualClass*) temp->persistentCopy();
      if (pers->isPersistent())
         cout << "Manual ist persistent" << endl;

      pers->spy();
   }
}
```

spy () Die Funktion *spy()* Gibt in ähnlicher Weise wie die Funktion *dumpOn()* Informationen über einzelne Objekte aus. Zusätzlich werden einige Datenbankspezifische Informationen (Datenbankschlüssel, Locks u.ä.) mit ausgegeben (siehe [Ver92b] Seiten 17ff).

3.4.4 Funktionen zum Datenbank Management

commit () / rollback () Die Befehle commit und rollback beeinflussen, ob eine Transaktion erfolgreich beendet wird (commit), oder ob alle Operationen während der aktuellen Transaktion verworfen werden sollen (rollback).

savepoint ()
undosavepoint () Mit den Funktionen *savepoint()* und *undosavepoint()* ist es möglich, den gesamten Zustand der Datenbank zu sichern bzw. wiederherzustellen. Dadurch können z.B. geschachtelte Transaktionen realisiert werden: dazu ruft man am Anfang der geschachtelten Transaktion *::dom->savepoint()* auf, um die Datenbank zu sichern. Schlägt die Transaktion später fehl, so kann die Datenbank durch ein *::dom->undosavepoint(*) wieder in den Zustand vor dem letzten Aufruf von *savepoint()* gebracht werden.

setDefaultLock () Mit dem Befehl *setDefaultLock()* kann eine Sperre für Objekte aus einer Datenbank vorgegeben werden. Beim Zugriff auf ein Objekt der Datenbank wird dieses dann automatisch mit dem entsprechenden Lock gesperrt. Mögliche Sperren sind: Snapshot-Lock (NOLOCK), Read-Lock (RLOCK), Write-Lock (WLOCK) und Update-Lock (ULOCK).

acquireslock () Der Befehl acquireslock fordert für ein bestimmtes Objekt eine kurze Sperre (Short-Lock) an. Als Parameter kann einer der folgenden übergeben werden:
NOLOCK (Snapshot-Lock), RLOCK (Read-Lock), ULOCK (Update-Lock), WLOCK (Write-Lock), IRLOCK (Intention Read-Lock), IWLOCK (Intention WriteLock) und IWRLOCK (Intention Write-Read-Lock).
Die Intention Locks speziell für Klassen reserviert. Mit ihnen kann ein Klassenobjekt verändert werden. Das IWRLOCK vergibt ein Write-Lock auf das Klassenobjekt und sperrt zusätzlich sämtliche Instanzen der Klasse mit einem Read-Lock (siehe auch [Ver92b] Seiten 99 und 173).
Kurze Sperren bleiben solange bestehen, bis sie entweder durch eine neue Sperre überschriegen werden, oder bis die aktuelle Transaktion beendet wird.

acquireplock () Um persistente Sperren zu vergeben gibt es den Befehl acquireplock (siehe [Ver92b] Seiten 98 und 168f). Mit diesem Befehl kann für eine Menge von Objekten (LinkVstr, siehe 3.3.6) eine dauerhafte Sperre angefordert werden.
Die Sperre bleibt solange bestehen, bis sie entweder explizit mit dem Befehl releaseplock aufgehoben wird, oder bis die (lange) Transaktion beendet wird. Als Parameter werden die Menge der zu sperrenden Objekte, die Art der Sperre und weitere Optionen übergeben.

Als Art der Sperre kann O_PL_READ, O_PL_WRITE und O_PL_NOLOCK angegeben werden. Zusätzlich kann angegeben werden, ob die Sperre hart (O_PL_HARD) ist, oder ob sie von anderen Sperren gebrochen werden kann (O_PL_SOFT). Als weiterer Parameter wird übergeben wie reagiert werden soll, wenn bereits eine Sperre für das angeforderte Objekt existiert. Hier stehen die Möglichkeiten O_PL_WAIT, O_PL_NO_BLOCK und O_PL_RESERVE zur Verfügung. Bei O_PL_WAIT wird solange gewartet, bis das Objekt freigegeben wird oder bis die Anfrage ein Timeout erhält. O_PL_NO_BLOCK liefert sofort eine Fehlermeldung, falls das angeforderte Objekt blockiert ist, und O_PL_RESERVE reserviert das Objekt und sendet eine Nachricht über email, wenn die Sperre vergeben wurde.
Als letzter Parameter kann angegeben werden, über welche Ereignisse man via email informiert werden will. Mögliche Optionen sind: O_PL_LOCK_BROKEN sendet eine Nachricht, falls ein Soft-Lock gebrochen wurde, O_PL_REQ_PENDING informiert darüber, daß ein anderer Benutzer eine Sperre für ein bereits gesperrtes Objekt anfordert und O_PL_OBJ_READY sendet eine email, wenn angeforderte Sperren erfolgreich vergeben wurden.

Beispiel 13

```
LinkVstr<ModuleClass> allmods =
 ModuleClass::allInstances();
PLOCK LockType(
 O_PL_ERITE,        // Write-Lock
 O_PL_HARD,         // Sperre kann nicht gebrochen werden
 O_PL_RESERVE,      // Objekt reservieren, Nachricht über
                    // email, wenn Sperre vergeben
 O_PL_OBJ_READY ||  // Nachricht, wenn Objekt gesperrt
 O_PL_REQ_PENDING);// Nachricht, wenn anderer Benutzer
                    // dasselbe Objekt anfordert
// Sperren anfordern
::dom->acquireplock(allmods, LockType);
…
// Sperren wieder freigeben
::dom->releaseplock(allmods);
```

releaseplock() Mit releaseplock können zuvor angeforderte Sperren wieder freigegeben werden (siehe Beispiel 13).

connectdb () Mit dem Befehl *::dom->connectdb(DBNAME)* kann eine Verbindung zu einer weiteren Datenbank aufgenommen werden. Dadurch ist es möglich, Objekte mittels *checkin()* und *checkout()* zwischen Datenbanken zu transferieren (siehe Beispiel 14).

checkin () / checkout () Die Funktionen checkin und checkout dienen dem Transfer von Objekten zwischen Datenbanken. Alle Objekte, die mittels checkout aus einer Datenbank gelesen werden, werden mit einer persistenten Sperre versehen. Beim Zurückschreiben der Objekte mit checkin wird die Sperre wieder aufgehoben. Das folgende Beispiel (Beispiel 14) verdeutlicht den checkout/checkin-Mechanismus. In dem ersten Programm (chekouts.C) werden sämtliche Objekte der Klasse Module aus der Datenbank ausgelesen und zum Schreiben gesperrt. Diese Objekte dürfen daraufhin von keinem Programm mehr verändert werden, welches nicht in derselben langen Transaktion läuft. Die ausgelesenen Daten können jetzt also zum Beispiel an ein anderes C++-Programm übergeben werden um sie dort ohne (ohne Datenbankoverhead) zu bearbeiten. Nach der Bearbeitung liest das Programm checkins.C die geänderten Daten ein und macht sie mit dem Funktionsaufruf checkin wieder für die Datenbank zugänglich.

Beispiel 14[12]

```
/*
 * chekouts.C - checkout, lange Transaktion starten
 */

#include <stream.h>
#include "OO7CCADT.1.1/ins/allClasses.h"

main()
{::dom=new PDOM;

 // starte lange Transaktion mit Names "LONGTRX1",
 // eine Sitzung und eine kurze Transaktion
 ::dom->beginsession("LONGTRX1","priv",NULL,(o_opts)0);
 {  ::dom->setDefaultLock(WLOCK);

    // Objekte aus der Datenbank oo7db auslesen und in die
    // Datenbank "priv" schreiben
    ::dom->connectdb("oo7db");
    PLock mylock(O_PL_WRITE,O_PL_SOFT,O_PL_WAIT,NULL);

    LinkVstr<Module> mod_ckout =
       PClassObj<Module>.pcheckout(
       "oo7db",        // Datenbankname
       FALSE,          // suche nicht rekursiv
       NULL_PREDICATE, // alle Instanzen
       mylock);        // Sperre mylock
    if (mod_ckout.size() == 0)
    {
       cerr << "Checkout hat kein Objekt gefunden!";
       cerr << endl;
       ::dom->exit(1);
    }
    cout << "Checkout hat " << emps_ckout.size();
    cout << " Objekte gefunden." << endl;

    // Daten in eine SLANG-Datei ausgeben
    SLANGout out("bspdaten.SLANG");
    out << mod_ckout;
    out.flush();
    // Die Daten können jetzt von einem anderen Programm
    // bearbeitet werden
 }
 // Beende kurze Transaktion und die Sitzung.
 // Die lange Transaktion bleibt bestehen!
 // commit() geschieht automatisch.
 ::dom->endsession();
}


/*
 * chekins.C - checkin, lange Transaktion beenden
```

12. Entnommen aus dem Versant Tutorial [Ver92a] und an die Datenbank aus Anhang A angepaßt.

```
 */
#include <stream.h>
#include "employee.h"
main()
{
 ::dom=new PDOM;

 // lange Transaktion weiterfuehren.
 ::dom->beginsession("LONGTRX1","priv","c2",((o_opts)0));
 {  // alle Module zurueckschreiben
    ::dom->connectdb("oo7db");
    SLANGin ("bspdaten.SLANG");

    LinkVstr<Module> mod_ckin = Module::allInstances();

    ::dom->checkin(mod_ckin,"oo7db",NULL_VSTR,NULL_VSTR);
    cout << mod_ckin.size();
    cout << " Module zurueckgeschrieben." << endl;
    mod.release();
 }

 // lange Transaktion beenden.
 // alle Objekte die nicht zurueckgeschrieben wurden
 // bleiben wie vorher vorhanden (rollback).
}
o_opts optionArray;
optionArray[0] = O_LXACT_ROLLBACK;
::dom->endsession(NULL,optionArray);
}
```

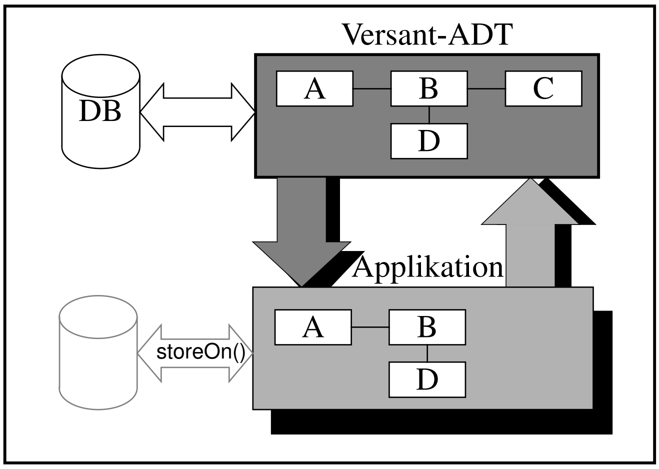
Kapitel 4

Datenaustausch

4.1 Anforderungen

Zusätzlich zur Implementierung des Versant-ADTs wurde im Rahmen der Projektarbeit ein einfacher Mechanismus zum Datenaustausch zwischen der Versant-Datenbank und einem C++-ADT implementiert. Dabei ist es möglich, einzelne Daten (Objekte) aus der Datenbank in eine SLANG-ähnliche Datei abzuspeichern, diese von einem C++-ADT einzulesen und die entsprechenden Daten zu bearbeiten. Nach der Bearbeitung soll der C++-ADT die Daten wieder auf den umgekehrten Wege in die Datenbank eintragen können. Auf diese Weise wird einem Applikationsprogramm, welches auf dem C++-ADT aufbaut, ermöglicht, in effizienter Weise auf Daten aus der Datenbank zu operieren. Das Applikationsprogramm braucht dabei nur auf Teile der Datenbank zuzugreifen und kann so die benötigten Daten effizient im Hauptspeicher verwalten.

Abb. 6 Datenaustausch zwischen Versant-DB und Applikation



In dem Beispiel (Abb. 6) sind in der Datenbank die Objektklassen A,B, C und D mit ihren Relationen abgelegt. Das Werkzeug (Applikation) benutzt allerdings nur die Objektklassen A, B und D. Sollen nun Daten für das Werkzeug aus der Datenbank abgespeichert werden, so brauchen nur die Instanzen der Klassen A, B und D in die (SLANG-) Datei abgelegt zu werden. Alle Verweise auf Objekte der Klasse C werden mit speziellen Schlüsseln versehen, damit beim Wiedereinlesen in die Datenbank keine Relationen verloren gehen.
Wenn eine Applikation die gespeicherten Objekte temporär auf einen Sekundärspeicher (Festplatte) ablegen will, so geschieht dieses über die NIHCL Funktion storeOn() (siehe [GOP91], Kapitel 7.4). Damit können auch Daten ohne Umweg über die Datenbank zwischen zwei Applikationen ausgetauscht werden. Der storeOn Befehl ist so programmiert, daß zu einem Objekt automatisch sämtliche von diesem Objekt referenzierten Objekte mit abgespeichert werden. Sind die Daten also stark untereinander vernetzt, so müssen viele Objekte abgespeichert werden obwohl eventuell nur wenige davon wirklich benötigt werden. Aus diesem Grund eignet sich das storeOn-Format nicht so sehr zur Datenübertragung zwischen einer Datenbank und einem Hauptspeicher-ADT. Vielmehr wird ein Verfahren benötigt, bei dem nur die Objekte übertragen werden, die der Hauptspeicher-ADT zu seinen Berechnungen braucht. Trotzdem muß gewährleistet sein, daß beim Zurücklesen der Objekte in die Datenbank sämtliche Referenzen auf Objekte die nicht übertragen wurden, wiederhergestellt werden. Zum Beispiel benutzt ein Algorithmus zum Plazieren von Zellen auf einem VLSI-Chip nur die Form der Zelle und nicht den eigentlichen Aufbau derselben. Mit dem storeOn Befehl der NIHCL-Bibliothek ist es nur möglich, die gesamte Zelle mit allen zugehörigen Informationen abzuspeichern. Durch das in dieser Projektarbeit implementierte Verfahren kann die Zelle nur mit ihrer Flächenausdehnung abgespeichert werden; alle anderen Informationen wie Struktur oder Inhalt der Zelle werden als Zeiger abgelegt. Beim Zurücklesen der Daten in die Datenbank können diese Zeiger dann richtig rekonstruiert werden.

Um einen Datenaustausch in dieser Weise zu ermöglichen, müssen einige Voraussetzungen an das Datenaustauschformat gestellt werden:

- Eindeutige Schlüssel innerhalb der Übertragungsdatei um Referenzen richtig aufbauen zu können.
- Objekttypen existieren auf der Datenbankseite und bei den Applikationsprogrammen evtl. in unterschiedlichen Versionen, d.h. sie können unterschiedliche Attribute haben.
  Werden also Attribute abgespeichert, die auf der Leseseite nicht vorhanden sind, so können diese überlesen werden. Umgekehrt müssen in der Übertragungsdatei nicht vorhandene Attribute beim Einlesen mit Standartwerden initialisiert werden.

Als Voraussetzung für einen solchen Datenaustausch muß also jedes Objekt in der Datenbasis über einen eindeutigen Schlüssel verfügen. Dazu werden die in MOOSE als Schlüsselattribut gekennzeichneten Attribute genommen (Sind keine Attribute als Schlüssel gekennzeichnet, muß künstlich ein Schlüssel erstellt werden; da die Versant Datenbank für jedes Objekt eindeutige Schlüssel vergibt, wird dann dieser genommen). Um möglichst flexibel bei der Vergabe der Schlüssel zu sein, werden diese hierarchisch geordnet. Ein Objekt kann also zu seinem lokalen Schlüssel zusätzlich Schlüsselattribute eines übergeordneten, also in der Vererbungshierarchie (isA-Relation) höher gestellten, Objektes erben. Über diesen globalen Schlüssel kann nun jedes Objekt in der Datenbank eindeutig identifiziert werden. Da diese Schlüssel eventuell sehr lang werden können, besteht die Möglichkeit, Objekte hierarchisch in der Übertragungsdatei zu schachteln. Dazu wird zunächst das Übergeordnete Objekt abgespeichert und danach werden geschachtelt alle Objekte abgelegt, die ihren (globalen) Schlüssel von diesem ersten Objekt erben. Durch diese Schachtelung wird die Übertragungsdatei verkürzt und es können immer noch alle Schlüssel eindeutig rekonstruiert werden. Beispiel 15 verdeutlicht den Unterschied zwischen geschachtelter und ungeschachtelter Abspeicherung.

Beispiel 15: `geschachtelte Objekte`

```
( CellClass
  < ( cellid 100 )
    ( pinsOfCell
    < ( id 1 ) >
    < ( id 2 ) >
    )
  >
)

( PinClass
  <
    ( id 1 )
    ( x 1 )
    ( y 5 )
    ( CellOfPin
      <
        ( cellid 100 )
      >
  >
  <
    ( id 2 )
    ( x 3 )
    ( y 5 )
    ( CellOfPin
      <
        ( cellid 100 )
      >
  >
)

Kann abgekürzt werden zu:

( CellClass
  < ( cellid 100 )
    ( pinsOfCell
      <
        ( id 1 )
        ( x 1 )
        ( y 5 )
      >
      <
        ( id 2 )
        ( x 3 )
```

```
        ( y 5 )
      >
    )
  >
)
```

4.2 Aufbau der Übertragungsdatei

Die Daten werden zwischen den ADTs über eine Datei im SLANG-Format ausgetauscht. Eine genaue Beschreibung des SLANG-Formates findet sich in [Sür88]. Das SLANG-Format besteht im wesentlichen aus ineinander geschachtelten Listen und Records. Listen unterscheiden sich dabei von Records durch ein vorgestelltes Schlüsselwort, welches die nachfolgenden Einträge charakterisiert.

Beim Datenaustausch zwischen den generierten ADTs sieht die Übertragungsdatei folgendermaßen aus:
Zunächst kommt eine Liste mit der Versionsnummer der SLANG Datei. Anhand der Versionsnummer ist es möglich, Änderungen im Übertragungsformat zu erkennen. Dadurch können Funktionen bereit gestellt werden, um bei Änderungen nach wie vor ältere Versionen einzulesen. Dazu muß nur vor dem Einlesen der eigentlichen Daten die Versionsnummer überprüft werden um dann die richtige Einleseroutine aufzurufen.
Nach der Versionsnummer folgt eine Liste mit dem Schemanamen und der Versionsnummer des generierten ADTs als Schlüsselwort. Anschließend folgt ein Record mit den abgespeicherten Objekten.
Die einzelnen Objekte (hier Instanzen von C++ Klassen) sind nun nach ihren Klassen gruppiert. Dazu wird eine Liste mit dem Klassennamen als Schlüsselwort angelegt; und die Instanzen werden als Records hinzugefügt. Eine Instanzvariable wird als Liste mit Variablennamen und Wert(en) abgespeichert.
Zusätzlich besteht die Möglichkeit, Objekte verschiedener Klassen geschachtelt abzuspeichern. Dieses ist sinnvoll bei Objekten, die Schlüsselattribute von einer anderen Klasse erben (siehe 4.2.2). Wird solch ein Objekt geschachtelt abgespeichert, so braucht statt dem kompletten globalen Schlüssel nur noch der lokale Teil des Schlüssel ausgegeben werden.

Abb. 7 Beschreibung des übertragungsformates in erweiterter Backus-Naur Form (EBNF):

Slang_File	::= Version_list Data_list
Version_list	::= '(' SLANG-Version Versionno ')'
Data_list	::= '(' Schemaname Data_record ')'
Data_record	::= '<' Class_list { Class_list } '>'
Class_list	::= '(' classname Instance_record { Instance_record } ')'
Instance_record	::= '<' Attributes | Class_list '>'
Attributes	::= Attribut_list { Attribut_list }
Attribut_list	::= '(' Attributname Attributvalue ')'

Versionno beschreibt die Version des Austauschformates um kompatibilität zu früheren Versionen zu ermöglichen. Sie sollte aus einer reellen Zahl mit (maximal) zwei Nachkommastellen bestehen.
Beispiel (Version_list): (SLANGVersion 1.00)

Classname ist der Klassenname der abgespeicherten Objekte wie er in C++ vergeben wurde. Ebenso sind *Attributname* und *Attributvalue* die Namen bzw. Werte der Instanzenvariablen des Objektes. Das Format, wie Variablenwerte abgelegt werden, ist unten beschrieben.

4.2.1 Speicherung der Attribute

Für jedes Objekt werden zunächst die Schlüsselattribute und anschließend die restlichen Attribute abgespeichert. Sind keine Attribute als Primärschlüssel markiert, so wird künstlich ein Schüssel eingeführt und entsprechend gekennzeichnet (siehe 4.2.2). Die verschiedenen Typen von Attributen werden folgendermaßen abgespeichert:

- **Einfache Attribute**
  Werden direkt durch den in C++ standardmäßig definierten << Operator ausgegeben.
- **Arrays**
  Bei Arrays wird zwischen Zeichenketten (Basistyp char) und allgemeinen Feldern (jeder andere Basistyp) unterschieden. Zur besseren Lesbarkeit werden Zeichenketten als normaler Text in Anführungszeichen geschachtelt gespeichert. Bei allen anderen Felder werden die einzelnen Komponenten durch Leerzeichen getrennt abgelegt.
  Beispiele: (title "Text"), (numbers 1 2 3 4 5).
- **Struct**
  Bei einem Struct werden ähnlich wie bei einem Array die einzelnen Komponenten durch Leerzeichen getrennt. Beispiel: (position 25 10)
- **Enum**
  Es wird die Repäsentation als ganze Zahl abgelegt
  Beispiel: enum {rot, blau, grün} -> blau wird als 1 abgespeichert
  (colour 1)
- **1_zu_1 Relationen**
  Es wird der Schlüssel des Objektes abgespeichert, auf daß verwiesen wird. Da der referenzierte Schlüssel aus mehreren Attributen (d.h. mehreren Listen) bestehen kann, werden alle diese Schlüsselattribute in einem Record zuammengefaßt: (ManualOfModule < (id 100) >)
- **1_zu_n Relationen**
  Wird als Record mit Listen der Schlüssel aller Objekte, auf die verwiesen wird, abgelegt: (AssembliesOfModule < (id 10) > < (id 11) > < (id 12) >)

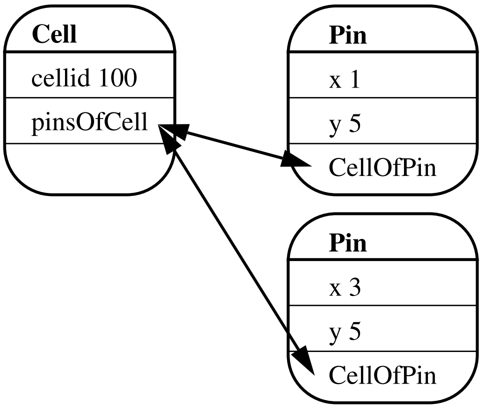
4.2.2 Schlüssel

Um beim Einlesen Objekte, die in der Datenbank bereits vorhanden sind, nicht neu anzulegen und um Relationen richtig aufbauen zu können, erhält jedes Objekt einen eindeutigen Schlüssel. Der (lokale) Schlüssel eines Objektes besteht aus den Attributen, die im MOOSE als Primärschlüssel gekennzeichnet wurden. Wird beim Einlesen ein solches Attribut gefunden, so wird es automatisch als Schlüsselattribut behandelt. Eine besondere Unterscheidung in der Übertragungsdatei ist also nicht notwendig.

Werden statt lokaler Schlüssel globale (hierarchisch vererbte) Schlüssel verwendet, so können zusätzlich zum lokalen Schlüssel einfach die geerbten Schlüsselattribute verwendet werden. Dabei muß allerdings vorausgesetzt werden, daß die Namen der lokalen und der geerbten Schlüsselattribute unterschiedlich sind. Das heißt, daß zum Beispiel die Klasse PinClass mir dem Schlüsselattribut *id* nicht noch ein weiteres Schlüsselattribut *id* von einer anderen Klasse erben kann[13].

Das Konzept der hierarchischen Schlüssel wurde eingeführt, damit bei einzelnen Objekten auch doppelte lokale Schlüssel auftreten können. Die Unterscheidung solcher Objekte erfolgt dann über zusätzliche, von anderen Objekten geerbte, Schlüsselattribute. Diese Vorgehensweise ist sinnvoll, wenn die Eindeutigkeit eines lokalen Schlüssels nicht gewährleistet ist und nicht ein zusätzliches künstliches Schlüsselattribut eingefügt werden soll. Bei einem Datenmodell zum Chip-Entwurf kann zum Beispiel als Schlüssel für einen Anschlußpin einer Zelle seine Position gewählt werden. Dieser Schlüssel ist jedoch nur im Zusammenhang mit einer bestimmten Zelle eindeutig. Daher können die Schlüsselattribute der Zelle an seine Anschlußpins vererbt werden um jeden Pin eindeutig ansprechen zu können.

Abb. 8 Hierarchische Schlüssel



In dem Beisiel in Abb. 8 ergeben sich für die beiden Pins die Schlüssel „(cellid 100) (x 1) (y 5)“ und „(cellid 100) (x 3) (y 5)“. Ein anderer Anschlußpin an der Position (1, 5) kann nur zu einer anderen Zelle gehören und erbt daher eine andere cellid.

Um Objekte bei der Datenübertragung eindeutig identifizieren zu können, muß immer der komplette hierarchische Schlüssel angegeben werden. Die Angabe des geerbten Schlüssels kann allerdings entfallen, wenn das Objekt in geschachtelter Form abgelegt wird.

Beispiel 16

```
( CellClass
  < ( cellid 100 )
    ( pinsOfCell
      < ( x 1 )
        ( y 5 )
      >
      < ( x 3 )
```

13. Um dieses zu vermeiden, müßte bei den geerbten Attributen zusätzlich noch der Klassenname der Klasse angegeben werden, von dem der Schlüssel geerbt wurde.

```
          ( y 5 )
        >
      )
    >
  )
```

Ein Spezialfall tritt auf, wenn im MOOSE kein Attribut als Schlüssel gekennzeichnet wurde. Um Referenzen in der Übertragungsdatei dabei noch eindeutig zu halten, muß in solch einem Fall ein künstlicher Schlüssel erzeugt werden. Prinzipiell unterscheiden sich hier nun der C++-ADT vom Versant-ADT.
Die Versant Datenbank vergibt automatisch für jedes Objekt, welches persistent angelegt wird, einen netzwerkweit eindeutigen Schlüssel. Mit diesem Schlüssel kann ein Objekt aus einer Datenbank wiedergefunden werden, da jeder Schlüssel nur einmal vergeben wird (selbst wenn Objekte gelöscht wurden wird der benutzte Schlüssel nicht neu vergeben). Deshalb kann dieser Schlüssel auch als Schlüssel für die Übertragungsdatei verwendet werden. Wird beim Einlesen in den C++-ADT ein solcher Schlüssel erkannt (gekennzeichnet durch das Wort „VKEY“), so muß die Verknüpfung zwischen dem Schlüssel und dem eingelesenen C++ Objekt explizit gespeichert werden, damit beim erneuten Abspeichern des Objektes wieder derselbe Schlüssel verwendet werden kann.
Die NIHCL-Bibliothek vergibt keine solchen Schlüssel, da die anfallenden Daten allein im Hauptspeicher verwaltet werden. Daher muß beim Abspeichern von Objekten aus dem C++-ADT anders vorgegangen werden. Hier wird als Schlüssel der Zeiger auf das zu speichernde Objekt verwendet. Dadurch wird garantiert, daß alle Schlüssel innerhalb der Übertragungsdatei eindeutig sind. Beim Einlesen eines Objektes mit solch einem Schlüssel (in der Übertragungsdatei gekennzeichnet durch das Wort „KEY“) wird grundsätzlich ein neues Objekt erzeugt. Anschließend können Relationen auf dieses Objekt eingetragen werden.

4.3 Anwendung

Die Benutzung der neuen Ein-/Ausgabefunktionen ist angelehnt an die C++ Klasse *iostream*. Einzelne Objekte oder Mengen von Objekten können mit dem << Operator in eine Ausgabedatei geschrieben werden. Alle Objekte, die ausgegeben werden sollen, werden zunächst nur gesammelt und erst durch den Befehl *flush()* in die Datei geschrieben. Daraufhin werden alle angegebenen Objekte abgespeichert. Objekte, die zwar von anderen Objekten referenziert werden, jedoch nicht abgespeichert werden sollen, werden in der Übertragungsdatei nur mit ihren Schlüsselattributen gespeichert. Dadurch ist eine korrekte Zuordnung der Referenzen nach dem Wiedereinlesen möglich.

partOf Eine besondere Semantik bei der Ausgabe von Objekten hat die partOf Relation. Soll ein Objekte abgespeichert werden, welches aus anderen Teilen besteht (d.h. es ist mit einer partOf Beziehung mit anderen Objekten verbunden), so wird nicht nur das Objekt selbst, sondern auch seine Bestandteile abgespeichert. Zusätzlich werden Schlüsselattribute entlang einer partOf-Relation vererbt. Das heißt zum Schlüssel eines Teilobjektes gehört automatisch auch der Schlüssel des Vaters.
Zum Einlesen einer Datei muß nur der Name der Datei an die Klasse SLANGin übergeben werden. Daraufhin wird die gesamte Datei eingelesen. Einzelheiten über die Benutzung der Ein-/Ausgabefunktionen sind dem Beispiel in Kapitel 4.3.2 zu entnehmen.

4.3.1 Neue Funktionen in der generierten Klassenhierarchie

Um einen Datenaustausch in oben genannter Form zu ermöglichen, müssen in der generierten Klassenhierarchie einige Funktionen vorhanden sein.

I. Ausgabefunktionen

Jedes Objekt muß seinen lokalen und globalen Schlüssel im SLANG-Format ausgeben können. Dazu werden die Funktionen

getKey () void getKey (ostream& file, char *praefix = " ") und

getGlobalKey () void getGlobalKey (ostream& file, char *praefix = " ") benutzt. Sie geben den entsprechenden Schlüssel in den ostream „file“ aus. Der zweite Parameter „praefix“ wird benutzt, um die Ausgabe zu formatieren. Bevor ein Schlüsselattribut ausgegeben wird, wird zunächst die Zeichenkette praefix ausgegeben. Besteht praefix aus Leerzeichen, so wird die Ausgabe entsprechend eingerückt.

Nachdem der Schlüssel eines Objektes ausgegeben wurde, werden diejenigen Attribute ausgeben, die nicht zum Schlüssel gehören. Dazu wird die Funktion

getAttributes () void getAttributes (ostream& file, char *praefix = " ") generiert. Die Parameter entsprechen denen von getKey() und getGlobalKey().

Um ein Objekt abzuspeichern müssen Methoden existieren, sodaß jedes Attribut ausgegeben werden kann. Dazu wird der Operator << benutzt:

operator << () ostream& operator << (ostream& out, type Value).

Für einfache Attribute ist der Operator bereits vorhanden, bei komplexen Attributtypen (z.B. Array oder Struct) werden spezielle Ausgabefunktionen generiert. Diese lassen sich genauso ansprechen wie die für einfache Attribute.

II. Einlesefunktionen

Beim Einlesen von Daten aus der Übertragungsdatei werden zunächst „leere“ Objekte erzeugt. Dazu gibt es die Funktion

createObject() static BaseClass *SLANGin::createObject (String& classname). Bei Angabe des Klassennamens gibt sie ein neues Objekt dieser Klasse zurück. Wird ein falscher Klassenname übergeben, so wird NULL zurückgegeben.

Um festzustellen, ob ein Objekt bereits eingelesen wurde, wird vom Generator die Funktion

findObject () static Object *findObject (char *key) erzeugt. Sie durchsucht die gesamte Datenbasis nach einem Objekt mit dem Schlüssel key. Wird kein Objekt gefunden, so wird NULL zurückgegeben.

Zum Eintragen der Attribute in ein leeres Objekt wird die Funktion

readAttributes () void readAttributes (char *Attr) benutzt. Sie Trägt die Attribut-Werte, die in der Zeichenkette Attr stehen, in die entsprechenden Instanzenvariablen des Objektes ein.

III. Verwaltung

Die Verwaltung der Ein/Ausgabe erfolgt in einer eigenen Klasse SLANGClass. SLANGClass ist die Oberklasse für zwei weitere Klassen SLANGin und SLANGout.
Ein Objekt vom Typ SLANGout regelt dabei die Ausgabe in eine Datei oder einen ostream. Auf Variablen vom Typ SLANGout kann (fast) genau wie auf einen ostream zugegriffen werden.

Beispiel 17

```
SLANGout out ("test.SLANG") // Variable out deklarieren;
// Ausgabe erfolgt auf Datei test
out << Objekt1 << Objekt2;
out.flush(); // Abspeichern
```

Funktionesdeklarationen der Klasse **SLANGout**:

Zunächst muß die Ausgabe in eine Datei vorbereiten werden. Das geschieht mit dem Konstruktor von SLANGout. Durch den Aufruf von

SLANGout () SLANGout (char *filename) wird eine Datei mit dem übergebenen Namen angelegt.

Die Ausgabe einzelner Objekte erfolgt mit dem Operator <<.

operator << () SLANGout& operator << (SchemaBaseClass& obj) oder
SLANGout& operator << (SchemaBaseClass* obj)

Wird anstelle eines einzelnen Objektes eine Menge von Objekten übergeben, so werden alle in dieser Menge enthaltenen Objekte abgespeichert.
SLANGout& operator << (LinkVstr<SchmaBaseClass> objs)
oder beim C++-ADT:
SLANGout& operator << (Set* objs)

Soll der gesamte Datenbestand gespeichert werden, so kann direkt die Funktion

addall () void SLANGout::addall() aufgerufen werden.

Sämtliche Objekte die abgespeichert werden sollen werden zunächst nur gesammelt und erst beim Aufruf der Funktion

flush () SLANGout& flush() wirklich gespeichert. Wird der Befehl flush nicht explizit benutzt, so werden die Daten mit dem Aufruf des Destruktors des SLANGout Objektes gesichert. Dieser wird spätestens am Ende des laufenden Programms aufgerufen.

Zum Einlesen von Objekten aus einer Datei dient die Klasse SLANGin.

SLANGin () Der Konstruktor von SLANGin liest sofort die gesamte Datei ein.

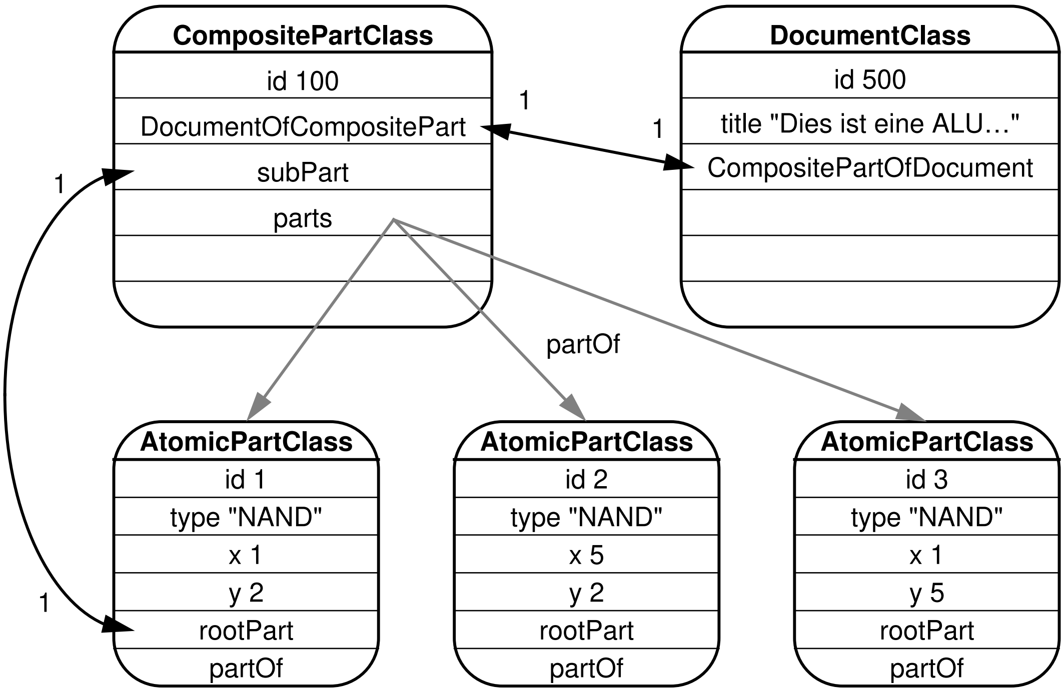
Beispiel 18

```
SLANGin ("test.SLANG")
```

4.3.2 Beispiel

Um den Datenaustausch an einem Beispiel zu verdeutlichen, wird im folgenden ein kleiner Ausschnitt aus der OO7 Datenbank (siehe Anhang A) betrachtet.
Es sollen einige Teile aus dem design library abgespeichert werden. Die angespeicherten Objekte sind in Abb. 9 graphisch dargestellt.

Abb. 9 Beispiel Datenbasis



Es wird also ein Composite Part mit zugehörigem Dokument in die Übertragungsdatei abgelegt. Das Composite Part besteht aus drei Atomic Parts. Da die Atomic Parts mit einer partOf Beziehung mit dem Composite Part zusammenhängen, werden sie automatisch bei der Speicherung des Composite Parts mit abgespeichert. Die dabei entstehenden Listen in der Übertragungsdatei werden so ineinander geschachtelt, daß aus der Struktur eindeutig hervorgeht, daß die Atomic Parts zu dem Compisite Part gehören. Dadurch kann die Angabe der Zeiger vom Composite Part zu den Atomic Parts und zurück entfallen.
Die entsprechende Übertragungsdatei sieht folgendermaßen aus:

Beispiel 19
```
( SLANGVersion 1.00 )
( OO7CCADT.1.1
  <
    ( CompositePartClass
      <
        ( id 100 )
        ( DocumentOfCompositePart
          <
            ( id 500 )
          >
        )
        ( subPart
          <
            ( id 1 )
          >
        )
        ( parts
          <
            ( AtomicPartClass
              <
```

```
                  ( id 1 )
                  ( type „NAND‟ )
                  ( x 1 )
                  ( y 2 )
                  ( rootPart
                    <
                       ( id 100 )
                    >
                  )
                >
                <
                  ( id 2 )
                  ( type „NAND‟ )
                  ( x 5 )
                  ( y 2 )
                >
                <
                  ( id 3 )
                  ( type „NAND‟ )
                  ( x 1 )
                  ( y 5 )
               >
             )
              >
            )
          >
        )
        ( DocumentClass
          <
            ( id 500 )
            ( title "Dies ist eine ALU aus 3 NAND-Gattern" )
            ( CompositePartOfDocument
            <
              ( id 100 )
            >
          )
        >
      )
  >
)
```

Das folgende Beispiel zeigt das entsprechende C++ Programm zum Anlegen und Abspeichern der Daten in der Versant Datenbank “oo7db":

Beispiel 20

```
// Beispiel 16
// Datenaustausch ueber eine SLANG-Datei

#include "OO7CCADT.1.1/ins/allClasses.h"

#define PERSISTENT

main()
{
 // ::dom = new PDOM;
 // ::dom->beginsession("oo7db", NULL);
 {
    CompositePartClass *alu;
```

```
    DocumentClass *doc;
    AtomicPartClass *nand1, *nand2, *nand3;

    // Objekte anlegen
    nand1 = new PERSISTENT AtomicPartClass();
    nand1->setid(1);
    nand1->setx(1);
    nand1->sety(2);
    nand1->settype(new PERSISTENT String10Class
       ("NAND", 4));

    nand2 = new PERSISTENT AtomicPartClass();
    nand2->setid(2);
    nand2->setx(5);
    nand2->sety(2);
    nand2->settype(new PERSISTENT String10Class
       ("NAND", 4));

    nand3 = new PERSISTENT AtomicPartClass();
    nand3->setid(3);
    nand3->setx(1);
    nand3->sety(5);
    nand3->settype(new PERSISTENT String10Class
    ("NAND", 4));

    doc = new PERSISTENT DocumentClass();
    doc->settitle(new String40Class
       ("Dies ist eine ALU aus 3 NAND-Gattern"));
    doc->setid(100);

    alu = new PERSISTENT CompositePartClass();
    alu->setid(100);
    alu->conDocumentOfCompositePart(doc);
    alu->conparts(nand1);
    alu->conparts(nand2);
    alu->conparts(nand3);
    alu->consubPart(nand1); // nand1 ist root part

    // Datenbestand abspeichern
    SLANGout out("daten.SLANG");
    out << alu << doc;
    // AtomicParts werden automatisch mit abgespeichert
    // (partOf)
    out.flush();
  }
}
```

Kapitel 5

Ausblick

Die vom Versant-Gererator erzeugten Quelltexte ermöglichen einen vereiheitlichten Zugriff auf Datenbankobjekte. Somit ist es recht einfach Applikationsprogramme, die auf einem generierten C++-ADT beruhen, auf einen Versant-ADT zu portieren. Interessant wäre es, bereits auf einem C++-ADT basierende Programme für eine Datenbank umzuschreiben, um anschließend die Geschwindigkeit zu vergleichen, und Vor- und Nachteile durch die zu Grunde liegende Datenbank zu untersuchen.
Auch ist es interessant, die Einsetzbarkeit eines verteilten Ansatzes, bei dem die Daten in einer (Versant-) Datenbank verwaltet werden und die Berechnungen ohne Datenbank durchgeführt werden (wie in Kapitel 4.1 beschrieben), an größeren Beispielen zu testen.

Weiterhin kann die Funktionalität des Versant-Generators noch erweitert werden.
Eine sinnvolle Erweiterung ist es, Teile der Datenbankkonfiguration bereits im MOOSE durchführen zu können. Bisher wird beim generieren des ADTs eine Datenbank mit Standardeinstellungen erstellt, die dann von Hand nachkonfiguriert werden muß.
Damit größere Programme effektiv auf eine Datenbank zugreifen können, muß diese vernünftig konfiguriert sein. Dazu gehört die Vergabe von Indices, die Verteilung der einzelnen Objekte im Speicher (Clusterung), die Einstellung der Cache-Größen und mehr.
Wünchenswert wäre die Möglichkeit, einige dieser Einstellungen bereits beim Entwurf des Modelles im MOOSE mit angeben zu können. Dazu gehört insbesondere die Vergabe von Indices; diese könnten dann auch für einen Hauptspeicher-ADT generiert werden.

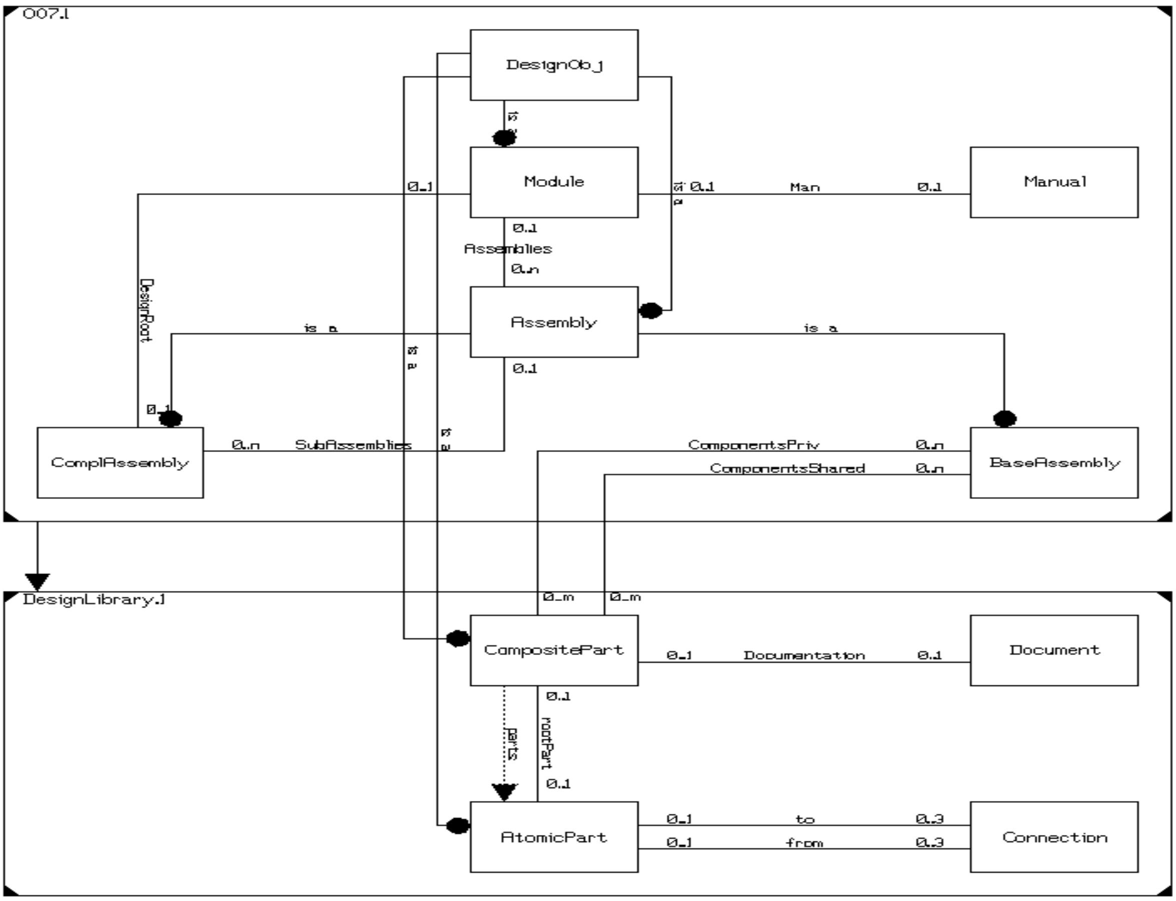
Anhang A

Beispiel

A.1 Beschreibung

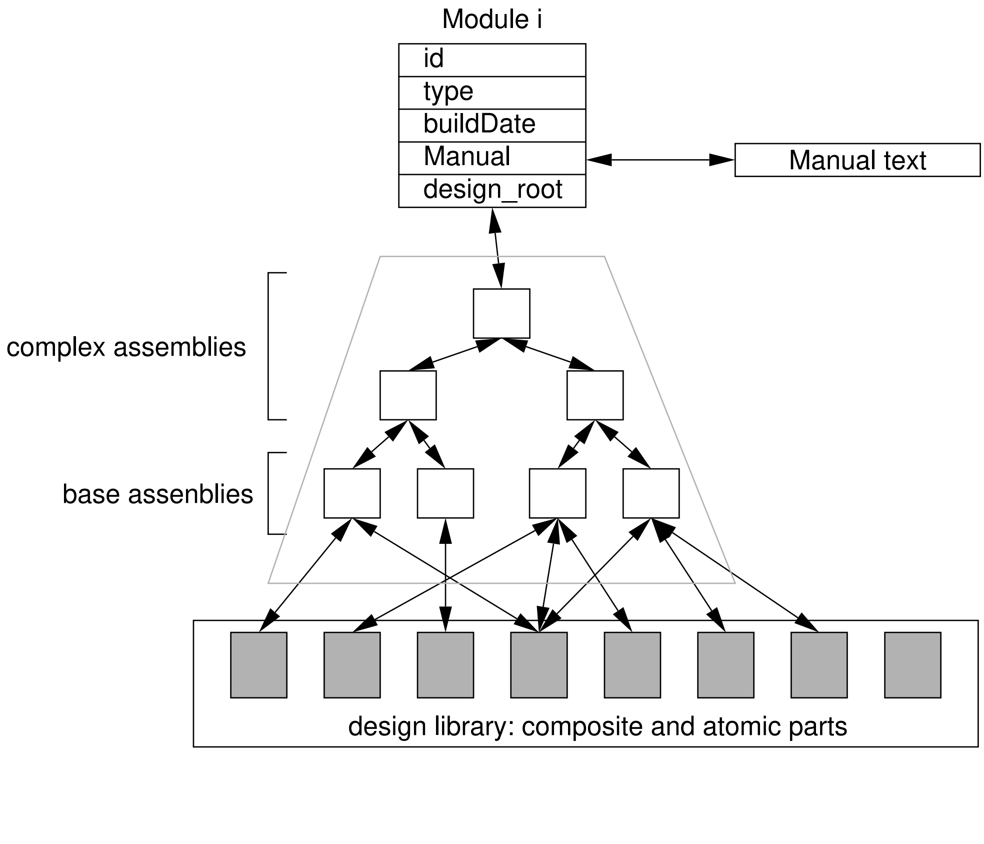
Sämtliche Beispiele in diesem Text beziehen sich auf das folgende Beispiel. es ist aus dem OO7-Benchmark der University of Winsconsin-Madison entnommen (siehe [CDN94]) und stellt eine typische Datenbasis für eine CAD Architektur dar.

Abb. 10 Datenbasis des OO7 Benchmarks



Die unterste Ebene der Datenbasis für den OO7-Benchmark besteht aus dem design library. Darin werden zusammengesetzte Teile (composite parts) abgelegt. Ein zusammengesetztes Teil kann zum Beispiel eine Zelle aus einer VLSI-Datenbank sein. Als Attribute hat es eine Identifikationsnummer (id) und ein Entstehungsdatum (buildDate). Ebenfalls ist jedem composite part ein Dokument zugeordnet, welches eine kurze Beschreibung des Teils enthalten soll.
Alle zusammengesetzten Teile bestehen aus atomaren Teilen (atomic parts). Sämtliche atomic parts sind untereinander verbunden. Diese Verknüpfung der atomaren Teile geschieht mit Hilfe eines speziellen Objektes (Connection), damit für jede Verbindung auch Attribute (type und length) gespeichert werden können.
Das design library macht den Großteil der Datenbasis beim OO7-Benchmark aus. Um jedoch auch komplexere Objekte modellieren zu können, wurde noch das Konstrukt der assemblies eingeführt. Ein assembly kann z.B. eine ALU beschreiben. Jedes assembly ist entweder komplex (complex assembly), das heißt es besteht aus anderen assemblies; oder es ist ein base assembly welches aus komplexen Teilen aus dem design library besteht. So können auf der Ebene der assemblies hierarchische Strukturen aufgebaut werden. Jede dieser Hierarchien wird in einem Modul zusammengefaßt. Jedem Modul ist ein Manual als Dokumentation zugeordnet.
Das Übersichtsbild in Abb. 11 beschreibt noch einmal den Aufbau der Datenbasis.

Abb. 11 Struktur eines Moduls[14]



14. vgl. Fig. 3 in [CDN94]

A.2 Imakefile zum Compilieren der Beispiele

Das folgende Imakefile kann als Schablone zum compilieren der Beispielprogramme oder auch anderer, auf den generierten (C++ oder Versant) ADTs beruhenden Programmen, dienen.
In den ersten Zeilen wird der Name und die Versionsnummer des verwendeten Schemas angegeben. Anschließend muß mittels *#define/#undef* festgelegt werden, ob das Applikationsprogramm auf der NIH- oder auf der Versant-Bibliothek aufbauen soll. Im Mittelteil der Imake-Datei werden die Parameter zum compilieren und linken der Applikation eingestellt. Dabei wird auch die aktuelle Rechnerplattform (HP oder Sun) berücksichtigt. Änderungen in diesem Teil sollten nicht nötig sein. Am Ende der Datei wird die zu übersetzende Applikation angegeben.
Zum Übersetzen des Applikationsprogrammes muß zunächst mit dem UNIX-Kommando *xmkmf* ein passendes Makefile erzeugt werden. Mit *make depend* werden dann Abhängigkeiten der Applikation von anderen Programmen (bzw. Bibliotheken) in das Makefile aufgenommen. Zuletzt kann mit dem Aufruf von *make* die Applikation übersetzt werden.

```
# Imakefile for applicationprograms using C++- or Versant-ADT's
#

ADT_NAME=  OO7
ADT_VER=   1
ADT_CONF=  1
# ADT to use
#define NIHCL
#undef VERSANT

################################################################################

#ifdef NIHCL
 ADT_NAME_LONG = $(ADT_NAME)CCADT.$(ADT_VER).$(ADT_CONF)
#endif
#ifdef VERSANT
 ADT_NAME_LONG = $(ADT_NAME)VersantADT.$(ADT_VER).$(ADT_CONF)
#endif
#ifdef HPArchitecture
 CXX = /usr/bin/CC
 EXTRA_DEFINES = -DCC
 INCLUDES = \
     -I /usr/include/X11R5 \
     -I /usr/include/CC \
     -I $(ADT_NAME_LONG)/ins \
     -I . \
     -I /zimmermann/tools \
 -I /usr/include/Motif1.2
 CXXFLAGS = +a1 -g
 DEPLIBS = $(ADT_NAME_LONG)/lib/lib$(ADT_NAME_LONG)_hp.a
 EXTRA_LIBRARIES = \
     -L /zimmermann/tools/common_c/lib \
     -L /zimmermann/tools/nihcl/lib \
     -L /usr/local/lib/gcc-lib/sparc-sun-sunos4.1.2/2.3.3 \
     -L $(ADT_NAME_LONG)/lib \
     $(DEPLIBS) -l nihcl -l handleerror_hp
#endif
#ifdef ApolloArchitecture
 CXX = /bin/CC
```

```
 ADT_CXXFLAGS = -g -I.. -I/zimmermann/tools
#endif
#ifdef SunArchitecture
 DEPLIBS = $(ADT_NAME_LONG)/lib/lib$(ADT_NAME_LONG)_sun.a
#ifdef NIHCL
 CXX = /tools/Saber/bin/CC
 EXTRA_LIBRARIES = \
     -L /zimmermann/tools/common_c/lib \
     -L /zimmermann/tools/nihcl/lib \
     -L /usr/local/lib/gcc-lib/sparc-sun-sunos4.1.2/2.3.3 \
     -L $(ADT_NAME_LONG)/lib \
     $(DEPLIBS) -lhandleerror_sun -lC /zimmermann/tools/nihcl/lib/libnihcl_SUN.a /
zimmermann/tools/nihcl/lib/nihclerror_SUN.a -lgcc
 EXTRA_DEFINES = -DCC
#endif
#ifdef VERSANT
 CXX= cxxcenter
 EXTRA_LIBRARIES = \
     -L /home/versant/lib/center.2.1/ \
     -L /home/versant/lib \
     -L /zimmermann/tools/common_c/lib \
     -L /usr/local/lib/gcc-lib/sparc-sun-sunos4.1.2/2.3.3 \
     -L $(ADT_NAME_LONG)/lib \
     $(DEPLIBS) -lvnih -lcxxcls -loscfe \
     -lhandleerror_sun -lgcc
 EXTRA_DEFINES = -DVERSANT -D_NO_X_APPLICATION
#endif
 CXXFLAGS = -g
 INCLUDES = \
     -I/usr/include/X11 \
     -I /home/versant/h \
     -I $(ADT_NAME_LONG)/ins \
     -I . \
     -I /tools/Saber/c++_1.1.0-r1.0/sun4-40/include \
     -I /zimmermann/tools
#endif

CC = $(CXX)

.SUFFIXES:$(.SUFFIXES) .C

.C.o:
     $(CXX) $(CXXFLAGS) $(INCLUDES) $(EXTRA_DEFINES) -c $*.C

################################################################################

SRCS = prog.C
OBJS = prog.o

ComplexProgramTarget(prog)
DependTarget()
```

Anhang B

Literaturverzeichnis

[CDN94] M. J. Carwy, D. J. DeWitt, J. F. Naughton: “The OO7 Benchmark“, University of Wisconsin-Madison, computer Science Department, 1994 (zugänglich über ftp: ftp.cs.wis.edu im Verzeichnis /pub/OO7)[Die92]U. Diel: “Abbildung Konzeptioneller Datenmodelle des Schemaeditors ‘DBED’ auf persistente Repräsentationsformen”, Diplomarbeit, Universität Kaiserslautern, 1992

[Che91] P.P. Chen, “The Entity-Relationship Model - Toward a Unified View of Data“, ACM TODS, Vol. 1, No. 1, 1976, S. 9-36

[GOP91] K. E. Gorlem, S. M. Orlow, P. S. Plexico: “Data Abstarction and Object-Oriented Programming in C++”, Teubner Verlag, Stuttgart, 1991

[Gor90] K. E. Gorlen: “NIH Class Library Reference Manual“, National Institute of Health, Bethesda, Maryland, 1990

[Här86] T. Härder, “Datenbanksysteme 1“, Skriptum zur Vorlesung, FB Informatik Universität Kaiserslautern, 1986

[Hit92] M. Hitz: “C++ Grundlagen und Programmierung”, Springer Verlag, Wien, 1992

[Kho93] S. Khoshafian: “Object Oriented Databases“, John Wiley & Sons Inc., New York, 1993

[KRR93] G. Kappel, S. Rausch-Schott, W. Retschitzegger, M. Schrefl, U. Schreier, M. Stumptner, S. Vieweg: “Object-Orineted Database Management Systems - An Evaluation“, Universität Wien, Mai 1993

[Sol93] Joachim Scholz: ”Ein graphischer Editor zur Eingabe von Datenbankschemata als Grundlage eines allgemeinen Schemabaukastens”, Diplomarbeit, Universität Kaiserslautern, 1993

[Str92] Bjarne Stroustrup, “C++ Prorgramming Language“, Second Edition, Addison-Wesley Publishing Company, Inc. , April 1992

[Sür88] B. Schürmann: “SLANG (Simple Data Description Language)”, SFB 124, Bericht 2288, Universität Kaiserslautern, 1988

[Süt92] Martin Schütze, “Abbildung des PLAYOUT-Datenmodells auf GemStone”, Diplomarbeit, AG VLSI Entwurf und Architektur, Universität Kaiserslautern, 1992

[Tre91] Peter Trebing, “Entwurf und Implementierung eines Umsetzers von einem Informationsmodell auf Hauptspeicherstrukturen zur Verwendung im PLAYOUT-Entwurfssystem“, Diplomarbeit, Universität Kaiserslautern, 1991

[Ver92a] Versant Object Technology, “C++/VERSANT Tutorial“, VERSANT Version 1.7, June 1992

[Ver92b] Versant Object Technology, “C++/VERSANT Reference“, Version 1.7, June 1992

[Wer94] C. Werner, “Ein Generator zur Umsetzung von Objektschemata in C++ Klassen”, Projektarbeit, AGVLSI Architektur und Entwurf, Universität Kaiserslautern, 1994

Anhang C

Index

A

B

C

D

F

G

T

U

V

W