SDAI auf DBS implementieren und anwenden[1]

U. Nink
Fachbereich Informatik, Universität Kaiserslautern,
Postfach 3049, 67653 Kaiserslautern
e-mail: nink@informatik.uni-kl.de

Zusammenfassung: STEP erlaubt die elektronische Verwaltung aller während des gesamten Lebenszyklus eines Produktes anfallenden Daten. Mit SDAI wird in STEP eine Zugriffsschnittstelle standardisiert, die von konkreten Datenbanksystemen (DBS) abstrahiert. Wir stellen ein Projekt vor, das zum einen die Eignung von relationalen und objektorientierten Datenbanksystemen (RDBS bzw. OODBS) zur Implementierung von SDAI-Schnittstellen untersucht. Zum anderen soll die Eignung von SDAI zur Implementierung von DB-Anwendungen geprüft werden. Dabei identifizieren wir wichtige Aspekte aus Sicht des SDAI- und des Anwendungsprogrammierers. Während die Implementierung der SDAI-Schnittstelle auf einem OODBS aufgrund der ähnlichen Datenmodelle und der Unterstützung einfacher Zugriffsfunktionen nahtloser ist, bieten RDBS bessere Adaptions- und Optimierungsmöglichkeiten für verschiedenste Anwendungen.

1. Einleitung

STEP (Product Data Representation and Exchange, ISO 10303 [1]) ist eine Norm zum Austausch von Produktdaten, die die Abbildung sämtlicher Merkmale eines Produktes während seines gesamten Lebenszyklus erlaubt. Zur Modellierung der Produktdaten wird die objektorientierte Datenmodellierungssprache EXPRESS [14] als Teil 11 und zentraler Knotenpunkt des Standards vorgegeben. Ein weiterer wichtiger Teil dieses Standards ist die Zugriffsschnittstelle SDAI (Standard Data Access Interface [15], Teil 22), die den Zugriff auf durch EXPRESS beschriebene Daten unabhängig von der verwendeten Datenhaltung erlaubt. Neben dem Austausch der Produktdaten gewinnt in zunehmendem Maße auch deren Verwaltung in Datenbanksystemen (DBS) an Bedeutung, da dadurch typische DBS-Eigenschaften wie Persistenz, Konsistenz und Konkurrenz verfügbar werden. Aus diesem Grunde sind Bemühungen im Gange, SDAI auf DBS abzustimmen und aufzusetzen. In Abb. 1 ist eine typische Architektur für SDAI/DB-Anwendungen skizziert. Es sind zwei verschiedene DBS dargestellt, auf denen jeweils eine eigene SDAI-Schnittstelle liegt. Ausgehend von einem gemeinsamen EXPRESS-Schema werden DBS-abhängig die jeweiligen DB-Schemata erzeugt (schwarz durchgezogene Pfeile). Der Zugriff einer Applikation auf eine Datenbank (helle Pfeile) erfolgt über SDAI. Damit kann dieselbe Anwendung auf unterschiedlichen DBS, ob relational (RDBS) oder objektorientiert (OODBS [6]), ohne größere Anpassungsmaßnahmen laufen. Bislang erfolgt der Datenaustausch zwischen verschiedenen DBS über Dateien in einem in STEP standardisierten Format, wobei DBS-spezifische “Prozessoren” das Lesen und Schreiben solcher Dateien übernehmen (gestrichelte Pfeile). Alternativ dazu kann in jede SDAI-Schnittstelle eine entsprechende File-Schnittstelle integriert werden, so daß ein DBS-unabhängiger Prozessor implementiert werden kann.

Wir untersuchen in einem laufenden Projekt zwei grundlegende Fragen. Die erste beschäftigt sich mit den Auswirkungen der Wahl eines DBS auf die Implementierung der SDAI-Schnittstelle. Wir diskutieren dazu für objektorientierte und relationale DBS Modell-, Architektur- und Realisierungsaspekte. Die zweite Frage beschäftigt sich mit der Eignung von

1) Dieses Papier wurde veröffentlicht in Informatik aktuell, Tagungsband der GI-Fachtagung BTW’95, Datenbanksysteme in Büro, Technik und Wissenschaft, Springer, ISBN 3-540-59095-1, März 1995.

SDAI für die Implementierung von Datenbankanwendungen. Hierzu diskutieren wir im wesentlichen Mächtigkeit, Handhabung und Effizienz der SDAI-Schnittstelle. Wir haben aufbauend auf einer gegebenen Implementierung der SDAI-Schnittstelle [5] auf einem objektorientierten Datenbanksystem [11] die Erstellung eines experimentellen Prototyps einer Stücklistenverwaltung für den Automobilbau begonnen, wobei wir die Architektur in Abb. 1 anstreben. Da ein Produkt (Fahrzeug) oft in vielen Varianten (bzgl. Motor, Karosserie, Farbe, Sonderausstattung) erzeugt werden kann, ist es sinnvoll, diese Information in varianten Stücklisten zu repräsentieren. Die verschiedenen Variationen, die für ein Bauteil existieren, sind spezifizierbar. Auf oberster Ebene bedeutet dies, daß ein Kunde Wünsche äußert (Klimaanlage, Servolenkung). Auf tieferen Ebenen qualifiziert ein solcher Wunsch eine Menge von Bauteilen mit jeweils zugehöriger Variante für den Einbau. Bei der Ableitung ist zu beachten, daß die Qualifikation eines Bauteils (transitiv) entscheidend für die Qualifikation anderer Bauteile sein kann (britisches Exportmodell impliziert Rechtslenker). Wir verwenden für die Ableitungsvorschrift einen regelbasierten Ansatz. Diese Anwendung ist genügend komplex, um repräsentative Aussagen zu ermöglichen. Wir erläutern unsere bisherigen Erkenntnisse und Erfahrungen bei deren Umsetzung.

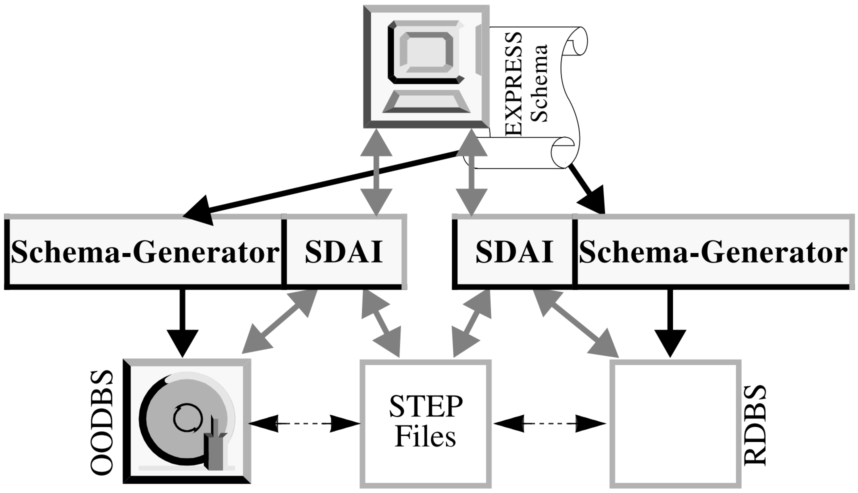

Abb. 1: Architektur für SDAI/DB-Anwendungen

In den folgenden beiden Abschnitten stellen wir zunächst Teil 11 (EXPRESS) und Teil 22 (SDAI) des STEP-Standards vor. Anschließend diskutieren wir die Implementierung von SDAI auf verschiedenen DBS und die Implementierung von DB-Anwendungen mit SDAI. Das Papier schließt mit einem Ausblick auf zukünftige Arbeiten.

2. Modellierung mit EXPRESS

Die wichtigste Voraussetzung für den Austausch von Daten zwischen verschiedenen Programmen auf unterschiedlichen DBS und Rechnerarchitekturen ist das Schaffen einer gemeinsamen Basis in Form eines einheitlichen abstrakten Datenmodells. Der zentrale Knotenpunkt des STEP-Standards ist deshalb die Modellierung der Produktdaten. Dies geschieht über eine Sprache (EXPRESS), die im folgenden kurz vorgestellt wird.

Vergleichbar mit einer Data Definition Language (DDL) in DBS werden einfache Datentypen und Konstrukte zum Aufbau komplexer Datentypen zur Verfügung gestellt. Wichtigster Baustein ist das *Entity*, das aus einer Menge von Attributen schon bekannter Datentypen aufgebaut wird. Da damit der Begriff Entity-Typ assoziiert ist, ersetzen wir im folgenden Entity durch Entity-Typ und bezeichnen dessen Ausprägungen als Instanzen. Neben einfachen, varianten, benannten, Enumerations- und Aggregattypen sind auch Entity-Typen als Attributtypen nutzbar. Letztere erlauben die Definition von (symmetrischen) *Beziehungen*; wir sprechen dann auch von entity-wertigen Attributen. Aggregierte entity-wertige Attribute ermöglichen die Modellierung von (1:n)- und (n:m)-Beziehungen, die zusätzlich über Kardinalitätsrestriktionen verfeinert werden können. Eine Menge von Typdefinitionen läßt sich modular zu einem *Schema* zusammenfassen. In Abb. 2 sind zwei Entity-Typen “Position” und “Variante” dargestellt, die zur Verwaltung von Einbaupositionen von Bauteilen in einem Produkt und den an jeweils einer Position einbaubaren Varianten eines Bauteils dienen. Position besitzt ein Attribut für textuelle Information (“info”). Seine restlichen Attribute repräsentieren folgende Beziehungen: (1) Es sollen bis zu zehn verschiedene Varianten an einer Position einbaubar sein (“moeglich”); (2) das eingesetzte Baukastenprinzip soll über eine Hierarchie von Positionen (“besteht_aus”) ausgedrückt werden. Durch das Schlüsselwort INVERSE wird für eine bestehende Beziehung (“FOR besteht_aus”) die Rückwärtsrichtung (“teil_von”) eingerichtet. Die Beziehung ist damit symmetrisch und gewährleistet referentielle Integrität.

```
SCHEMA Stueckliste;
  ENTITY Position;
      info: STRING;
      moeglich: LIST[0:9] OF Variante;
      besteht_aus: SET OF Position;
    INVERSE
      teil_von: Position FOR besteht_aus;
  END_ENTITY;
  ENTITY Variante;
      nr: INTEGER;
      info: STRING;
    WHERE
      Bereich: 1 <= SELF.nr <= 10;
  END_ENTITY;
END_SCHEMA;
```

Abb. 2: Ausschnitt aus einem EXPRESS-Schema

Weiter können *Regeln*, interpretierbar als Integritätsbedingungen, aufgestellt werden. Zu deren Definition sind aus Programmiersprachen bekannte Konstrukte (Zuweisung, Operationen, Variablen, Funktionen und Prozeduren) verwendbar. Da diese jedoch nicht zur Verhaltensbeschreibung von Entity-Typen herangezogen werden können, ist EXPRESS nicht voll objektorientiert, sondern “nur” strukturell objektorientiert. Im Beispiel schränkt die WHERE-Klausel den “Bereich” der gültigen Variantennummern ein. Zur Verfeinerung von Entity-Typen ist (multiple) *Vererbung* einsetzbar. Vererbt werden Attribute und Regeln. Abweichend von gängigen Vererbungskonzepten sind Vererbungsbeziehungen weiter einschränkbar, was wir an einem Beispiel erläutern. Ausgehend von einem Entity-Typ “Motor” gebe es die Subtypen “Turbo” und “Einspritzer”; Motoren können nun mit einem Turbolader, einer Einspritzanlage oder aber beidem ausgestattet sein. In einer ersten Umsetzung definieren wir zu Motor die Subtypen Turbo, Einspritzer und TurboEin - letzteren als Subtyp von Turbo und Einspritzer. In EXPRESS ist die Definition von TurboEin nicht notwendig, da mit “ENTITY Motor SUPERTYPE OF (Turbo ANDOR Einspritzer)” die aufgeführten Subtypen gemeinsame Instanzen haben dürfen. Anstelle von “ANDOR” erzwingt man mit “ONEOF” und “AND” die Zugehörigkeit einer Instanz zu genau einem bzw. zu allen der angegebenen Entity-Typen.

Aus diesem kurzen Überblick über EXPRESS wird deutlich, daß bei der Umsetzung auf das Datenmodell eines DBS einige Probleme auftreten. Wir kommen darauf noch zurück. Wie Instanzen erzeugt und manipuliert werden, beschreibt der Standard mit dem Teil 22, der Zugriffs- und Manipulationsschnittstelle SDAI.

3. Zugriff über SDAI

Ein weiterer wichtiger Punkt für die Unterstützung des Austauschs von Produktdaten ist neben einem einheitlichen Datenmodell eine einheitliche Zugriffsschnittstelle, die vom darunterliegenden DBS abstrahiert. Während EXPRESS das Datenmodell in STEP festlegt, beschreibt SDAI diese Schnittstelle. Für ihre Spracheinbettung werden verschiedene Einbettungsvorschriften (Language Bindings) für jede der vorgesehenen Zielsprachen C, Fortran und C++ standardisiert. Anbindungen an Sprachen wie C oder Fortran weisen aufgrund fehlender objektorientierter Konzepte einen deutlichen Reibungsverlust auf. Eine optimale Unterstützung für die SDAI-Schnittstelle bietet in unseren Augen nur die Anbindung an C++, auf die wir uns im folgenden beschränken. Es wird zwischen *Early* und *Late Binding* unterschieden, was nicht mit den Begriffen frühes und spätes Binden in objektorientierten Programmiersprachen verwechselt werden sollte, da keine Übereinstimmung, sondern nur eine Überlappung mit letzteren besteht. Zur besseren Abgrenzung ersetzen wir die SDAI-Begriffe im folgenden durch E- und L-Binding und erklären sie genauer. Im L-Binding sind alle SDAI-Operationen, die dem Programmierer zur Verfügung stehen, schemaunabhängig spezifiziert; alle Hinweise auf beteiligte EXPRESS-Konstrukte erfolgen über Eingabeparameter. Das E-Binding ist sehr viel spezieller; es stellt nur eine Teilmenge der Operationen (bspw. fehlt der Datenzugriff über Schemainformation) zur Verfügung; Hinweise auf EXPRESS-Konstrukte sind weitgehend in die Namen der Operationen integriert. So gibt es z.B. für ein Attribut einer Instanz “pos” des Entity-Typs Position zwei verschiedene Zugriffsmethoden: “pos->info()” und “pos->GetAttr(“info”)”. Die zweite Methode, die nur im L-Binding definiert ist, erlaubt im Gegensatz zur ersten auch den Zugriff auf erst zur Laufzeit bekannte Attribute. In C++ können aber beide Methoden sowohl früh als auch spät gebunden werden, das ist der jeweiligen Implementierung überlassen. Dies hat natürlich erheblichen Einfluß auf die Effizienz, zumal eine Reihe weiterer SDAI-Operationen hiervon betroffen ist.

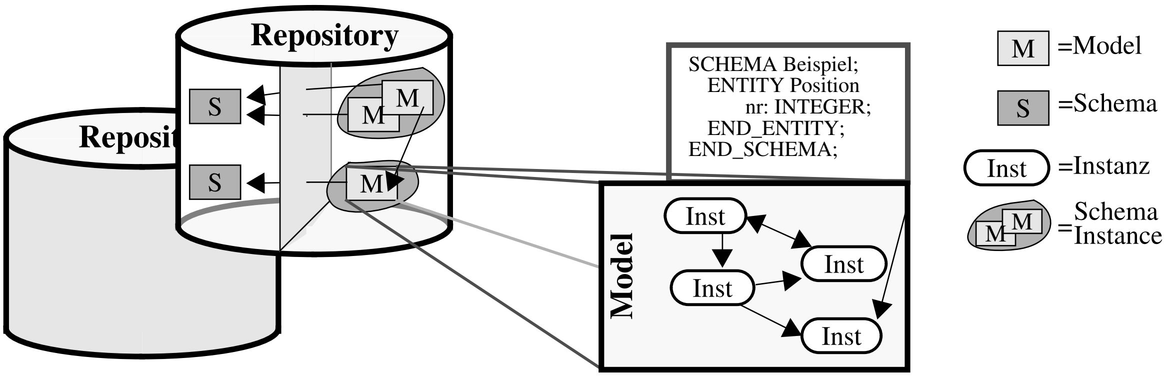

Abb. 3: Datenorganisation in SDAI

SDAI führt nun folgende Abstraktionen für Datengranulate und -operationen ein. Instanzen werden in gröberen Datenbehältern, *Model* genannt, gesammelt (Abb. 3). Ein Model ist einem *Schema* zugeordnet, das Metainformationen für Instanzen bereitstellt. Eine Menge von Schemata und Models wird zu einem *Repository* zusammengefaßt. Abweichend von der zum Zeitpunkt der Erstellung dieses Papiers offiziellen Norm, in der Transaktionen mit Repositories verwoben waren, gehen wir von einer orthogonalen Modellierung eines Transaktionskonzeptes wie in [15] aus. Auf dort ebenfalls neu vorgestellte Konzepte, wie *Schema Instance* (eine Menge von sich auf das gleiche Schema beziehende Models als Gültigkeitsbereich von Beziehungen und Regeln) oder *SDAI-Query* (erlaubt assoziative Extraktion einer Teilmenge eines Aggregats), konnten wir jedoch nicht eingehen. Die folgende Auflistung zeigt die wichtigsten Arten von Operationen (ohne Schemadienste) ihren verschiedenen Bereichen zugeordnet:

- Kontrollfluß (Open Session, Begin Transaction, ...)
- Repository (Open, Close)
- Model (Create, Delete, Get Entity Extent, Validate Global Rule, ...)

- Instanz (Get Attribute, Put Attribute, Get Type, ...)
- Aggregat (Add, Remove, Iteratoren, ...)

Nach dieser Einführung in die Modellierungssprache EXPRESS und die zugehörige Zugriffsschnittstelle SDAI gehen wir nun zur Diskussion der Implementierung von SDAI auf unterschiedlichen DBS über. Im darauf anschließenden Kapitel stellen wir unsere bisherigen Erkenntnisse in Bezug auf die Implementierung von DB-Anwendungen mit SDAI vor.

4. Implementierung von SDAI auf Datenbanksystemen

Im folgenden zeigen wir, welchen Einfluß verschiedene DBS auf die Implementierung einer SDAI-Schnittstelle haben. Die betrachteten Punkte sind unabhängig von der verwendeten SDAI-Implementierung gültig. Für die Diskussion schien uns die Einordnung der jeweils als wichtig empfundenen Aspekte in die Bereiche *Modell-* (objektorientiert, relational, SDAI), *Architektur-* (Page-, Object-, Query-Server [4, 7]) und *Realisierungsaspekte* (Synchronisation, Pufferung, Speicherungsstrukturen) sinnvoll.

4.1 Modellaspekte

Da EXPRESS das für SDAI zugrundeliegende Datenmodell ist - alle Strukturen der SDAI-Spezifikation liegen in EXPRESS vor - hängt die Implementierung von SDAI von der Abbildung von EXPRESS auf das Datenmodell des DBS ab. [13] beschreibt eine Abbildung von EXPRESS auf das *Relationenmodell*. Probleme bereitet die Umsetzung objektorientierter Modellierungskonzepte, wie komplexe Typen oder Vererbung (impedance mismatch). Im wesentlichen werden für jeden Entity-Typ und für jedes komplexe Attribut (aggregiert oder entity-wertig) eigene Relationen und Sichten erzeugt. Jede Instanz erhält eine eigene Identität (ID), über die die Beziehung zu komplexen Attributen und damit auch anderen Instanzen realisiert wird. Bei der Abbildung auf *objektorientierte* DBS gibt es weniger Reibungen. Wir legen im folgenden das Datenmodell von C++ zugrunde. Jeder Entity-Typ wird auf eine eigene Klasse abgebildet, wobei jedes seiner Attribute ein C++-Attribut, genannt "Element", der Klasse definiert. Einfache Attribute können ohne Indirektion integriert werden. Für komplexe Attribute werden Container-Klassen bereitgestellt. So können Beziehungen zu anderen Entity-Typen nahtlos auf instanz- bzw. zeiger-wertige Klassenelemente übertragen werden. Starre Abbildungsvorschriften können jedoch ein Nachteil sein. Gerade alternative Möglichkeiten bei der Umsetzung von z.B. symmetrischen Beziehungen (Suchfunktion oder Materialisierung der Rückwärtsrichtung) bieten Optimierungsmöglichkeiten. Für eine SDAI-Operation wie "FindEntityInstanceModel" (suche zur Instanz das zugehörige Model) bspw., die eher selten verwendet wird, ist die Bereitstellung einer Suchfunktion angebracht.

Einige Punkte bereiten sowohl RDBS als auch OODBS Probleme. Eine nahtlose Abbildung des Vererbungskonzeptes von EXPRESS erfordert, daß die Zugehörigkeit einer Instanz zu mehreren Entity-Typen instanzabhängig gesteuert werden kann. In den Datenmodellen von C++ und anderen kommerziellen Systemen ist dies nicht möglich. Im C++-Binding von SDAI wird dafür multiple Vererbung herangezogen, jedoch muß jede (in EXPRESS implizite) Kombination von Entity-Typen explizit als eigene Klasse modelliert werden. Da die Zahl der Kombinationen sehr stark mit n wächst (bei n mit ANDOR verknüpften Entity-Typen $O(2^n)$), ist eine automatische Generierung aller Klassen nicht sinnvoll. Man nutzt stattdessen den Template-Mechanismus von C++, mit dem die benötigten Klassen über Instantiierung erzeugt werden. Damit muß aber im Anwendungprogramm codiert werden, welche Kombinationen auftreten dürfen. Das bedeutet, daß schon bei der Erzeugung einer Instanz entschieden wird, zu welcher Kombination sie gehört; jede nachträgliche Änderung der Zugehörigkeit impliziert zumindest die Migration der Instanz in eine andere Klasse. Ein weiteres Problem stellen die in EXPRESS definierbaren Regeln dar, die die Gültigkeit von Instanzen festlegen; wir verstehen sie als Integritätsbedingungen. Die Verantwortung für deren Einhaltung überträgt SDAI dem Anwendungsprogrammierer. Da aber in vielen DBS Integritätsbedingungen spezifiziert und automatisch überwacht werden können, sollte eine Möglichkeit zur Nutzung dieser Fähigkeiten angestrebt werden. Bei automatischer Überwachung ist getrennt von der EXPRESS-Schemabeschreibung eine Einteilung der Bedingungen in *direkt* oder *verzögert auszuwerten* effizienzsteigernd einsetzbar. Dies führt zu einem eigenen Integritätssubsystem. Auch die Gleichheitssemantik in SDAI hat Auswirkungen auf die Effizienz. Es wird zwischen *IsSame* und *IsEqual* unterschieden. Bei ersterem werden die Identitäten zweier Instanzen, bei letzterem jeweils die Werte der einfachen Attribute der Instanzen und rekursiv ihrer referenzierten Instanzen verglichen. Ein spezieller Algorithmus fängt im zweiten Fall etwaige Zykel ab. Da dieser "tiefe" Vergleich mitunter sehr zeitaufwendig werden kann und ohnehin oft nicht der Gleichheitssemantik der Benutzer entspricht, wären hier benutzerdefinierte Vergleichsvorschriften, die schon bei der Datenmodellierung definiert werden können, vorzuziehen. So ließen sich z.B. für den Vergleich zweier Varianten die textuelle Information (info) ihrer zugehörigen Positionen heranziehen, ohne die Attribute der Varianten berücksichtigen zu müssen.

4.2 Architekturaspekte

Um architekturunabhängig zu bleiben, kann SDAI dem Anwendungsprogrammierer kaum direkte Mittel zur Optimierung bieten. Dies muß damit eine wesentliche Aufgabe der Schnittstellenimplementierung sein. Optimierungsspielraum sehen wir insbesondere bei der Wahl des Anforderungs- bzw. Nachladegranulats und in der Cluster-Bildung. Mit der Einführung von einfachen Anfragen [15] kann man darüberhinaus auf Anfrageoptimierung zurückgreifen. Zur Verdeutlichung der Diskussion zeigt Abb. 4 die Architektur von SDAI-Schnittstellen auf unterschiedlichen Workstation/Server-DBS-Architekturen [4, 7]. Diese DBS-Architekturen erhalten ihre Bezeichnung Page-, Object- bzw. Query-Server nach dem Datenaustauschgranulat zwischen Server- und Workstation-Komponente (S- bzw. WS-DBS). Gestrichelte Pfeile repräsentieren Schnittstellenaufrufe, durchgezogene Pfeile, deren Dicke die Menge der auf eine Anforderung hin übertragenen Daten suggerieren soll, den Datenfluß.

Der Aufruf einer SDAI-Operation erfordert für Page- und Object-Server den Aufruf einer oder weniger Operationen der WS-DBS-Schnittstelle. Die Anforderung einer Instanz eines Entity-Typs erzwingt evtl. die Anforderung einer Seite bzw. eines Objektes aus der Datenbank. Das gilt auch für die Navigation von Instanz zu Instanz. Diese Nachladevorgänge geschehen automatisch, und angeforderte Seiten bzw. Objekte werden in einem vom WS-DBS verwalteten Cache auf der Workstation gepuffert. Mit ObjectStore [11] als Page-Server ist es weiter möglich, eine Seitenmenge, die mindestens eine und maximal alle Seiten eines Segmentes enthält, auf einmal übertragen zu lassen. Mit Versant [17] als Object-Server kann man Objektmengen übertragen, im Extremfall das gesamte Model auf einmal. Ontos [2] erlaubt als adaptierbares System, die Übertragung objekt-, seitenbezogen oder sogar gemischt zu gestalten. In allen Architekturen ist die Größe des Transfergranulates anpaßbar.

Während für Page- bzw. Object-Server das Transfergranulat eine Seite bzw. ein Objekt ist, sollte es beim Query-Server zur Minimierung der Kommunikation ein Model oder ein größerer Teil davon sein. Bei relationalen Systemen, die wir als Spezialfall des Query-Server auch als Relationen-Server bezeichnen, wären es mehrere Instanzenmengen (abhängig von der Zahl

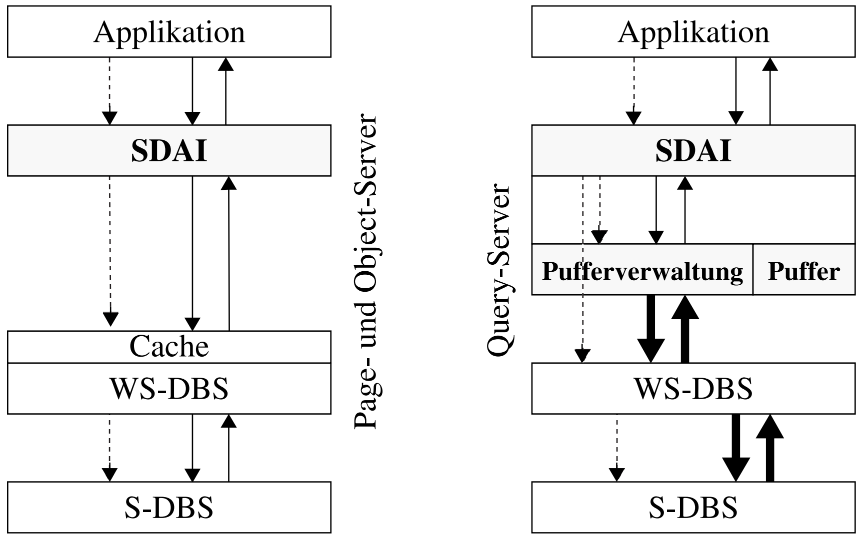

Abb. 4: Architektur von SDAI-Schnittstellen auf WS/S-DBS-Architekturen

der Entity-Typen im Model). Wir unterstützen dies, indem wir einen Puffer verwalten (siehe rechten Teil der Abbildung) und beim Öffnen eines Model eine Anfrage ausführen lassen, die den Inhalt des Model oder einen Teil davon in diesen Puffer lädt. Anschließende SDAI-Operationen, die sich auf das im Puffer befindliche Model beziehen, können dann ohne den Zugriff über das DBS allein auf dem Puffer abgewickelt werden. Da der Zugriff auf ein entitywertiges Attribut (Dereferenzierung) eine Instanz aus einem anderen Model liefern kann, ist u.U. auch hier ein Nachladen notwendig. Dies muß explizit in der SDAI-Schnittstelle kodiert werden, was jedoch im Gegenzug einen großen Optimierungsspielraum bietet, da das Transfergranulat kontextabhängig von der einzelnen Instanz bis hin zum kompletten Model reicht. Eine solche kontextabhängige Entscheidung kann über die SDAI-Operation "Get_Attribute" realisiert werden, da jede Dereferenzierung über diese Operation erfolgen muß. Geschieht das Nachladen über die Anfrageschnittstelle des DBS, so sind einzelne Objekte als Transfergranulat zu vermeiden; vielmehr ist wieder das gesamte Model oder ein Teil davon nachzuladen.

Für *Page-Server* ist ein Model auf ein Cluster, also auf eine zusammengehörige Menge von Seiten, z.B. ein Segment, abzubilden. Models, die aufgrund vieler model-übergreifenden Referenzen ihrer Instanzen eine starke indirekte Verbindung aufweisen, sind möglichst in das gleiche Segment abzulegen, um Verwaltungsaufwand oder nachfolgende Zugriffe einzusparen bzw. zu optimieren. Allgemein gilt, daß die Wahl der Cluster-Bildung größten Einfluß auf die Effizienz eines Page-Server hat. Wir können dies in der erforderlichen Kürze nicht ausreichend erklären und verweisen deshalb auf [7]. Für *Object-Server* kann Cluster-Bildung eingesetzt werden, um auf Server-Seite eine Einsparung von Transfers vom Server-Seitenpuffer zur Platte und zurück und eine bessere Server-Pufferausnutzung zu erzielen. Diese Verbesserung gilt auch für *Query-Server*. Hier gilt weiter, daß die Kommunikation zwischen Server und Workstation durch mengenweises Zusammenstellen von Instanzen zur Übertragung minimiert wird, und zwar unabhängig von jeglicher Cluster-Bildung.

Der für unser Projekt verwendete SDAI-Prototyp setzt auf ObjectStore, einem Page-Server, auf und folgt der Architektur im linken Teil der Abb. 4. Ein Model ist dabei genau einem Segment zugeordnet, und das Laden von Daten geschieht seitenweise. Ursprünglich beabsichtigten wir, das kommerzielle Produkt ST-Developer 1.3 von Step Tools Inc. [16] einzusetzen. Dieses erwies sich jedoch als ungeeignet, da es keine SDAI-Schnittstelle für ObjectStore bereitstellte. Die Alternative, Daten zunächst aus der Datenbank in eine Datei auszulesen und dann über SDAI (gegeben als L-Binding für C) zu bearbeiten, schien uns kaum sinnvoll.

4.3 Realisierungsaspekte

Die im folgenden betrachteten Aspekte Synchronisation, Pufferung und Speicherungsstrukturen sind eher unabhängig von der jeweiligen Architektur. Bei einem Page-Server ist im Sinne der *Synchronisation* die Abbildung jedes Model auf ein eigenes Cluster in Erwägung zu ziehen, da so Instanzen verschiedener Models nicht in der gleichen Seite liegen können, was den konkurrierenden Zugriff auf Instanzen unterstützt. Bei Einsatz adaptiver Locking- und Callback-Mechanismen [3] zur Unterstützung variabler Sperr- und Pufferaustauschgranulate benötigt man diese vorbeugende Maßnahme nicht. Dies gilt somit auch für Object- und Query-Server, denn diese können Sperren auf Objekt- bzw. Tupelebene halten. Zur Minimierung der Sperrverwaltung und der Anzahl der Sperranforderungen spielen über Repräsentanten sperrbare komplexe Objekte eine große Rolle. Der Idealfall ist dann die Abbildung eines Model auf ein komplexes Objekt. Um komplexe Objekte oder allgemein Anfrageergebnisse eines Query-Server auf Seite der Anwendung bereitzustellen, ist dort das Einrichten eines *Puffers* sinnvoll. Zusätzlich zu den hierzu in 4.2 erfolgten Überlegungen betrachten wir einige weitere Aspekte. Die Pufferung ist durch Quelltextanalyse oder Programmiererangaben steuerbar; ersteres ist sehr aufwendig, letzteres einfach und zumutbar bspw. über SQL. Diese Flexibilität erfordert die Unterstützung verschiedener Austauschgranulate, welche von einzelnen Instanzen über Mengen von Instanzen eines Entity-Typs bis hin zu vollständigen Models reichen können. Bei Page- und Object-Server sind deren implizite Caching-Mechanismen nutzbar. So muß zwar eine auf das jeweilige System angepaßte Abbildung erfolgen, die Implementierung der SDAI-Schnittstelle ist jedoch einfacher als auf einem Query-Server, da sie keinen Puffer explizit verwalten muß. Die Implementierungen sollten unterschiedliche *Speicherungsstrukturen* für Daten auf Platte und Hauptspeicher zur Verfügung stellen [12], da beim Zugriff auf Externspeicher der I/O-Overhead und im Hauptspeicher der Rechenaufwand wesentlich die Effizienz bestimmen. Da EXPRESS lediglich abstrakte "Container" vorgibt, ist dies leicht zu verbergen. Applikationsabhängige Anpassungen sollten möglich sein, um ein ggf. bekanntes Zugriffsverhalten auszunutzen. Ebenso kann im Hauptspeicher Pointer-Swizzling [8], das im wesentlichen eine Transformation der DB-Referenzen auf Hauptspeicheradressen darstellt, entscheidende Effizienzgewinne bringen.

5. Implementierung von DB-Anwendungen mit SDAI

Eine leichte Portierbarkeit einer Anwendung auf unterschiedliche DBS erfordert, daß sich der Programmierer an die Möglichkeiten von SDAI hält. Wir wollen dazu im folgenden unsere Erfahrungen mit der Implementierung der Stücklistenanwendung berichten. Wir diskutieren, welche Möglichkeiten SDAI zur Anwendungsprogrammierung bietet, welche Einschränkungen sich ergeben und welche Erweiterungen sinnvoll sind. Wir beschränken uns darauf, daß die Anwendung auf nur einem Repository arbeitet - diese Sichtweise führt zu einer einfachen Handhabung, ohne die wesentlichen Punkte einzuschränken. Die abgeleiteten Aussagen sind meist allgemeingültig und auch in anderen Anwendungen nachvollziehbar; sprachliche Aspekte sind auf Sprachen mit ähnlichen Eigenschaften wie C++ übertragbar.

Abbildung der Anwendungsdaten auf Models Zunächst liegt die Abbildung einer kompletten Stückliste auf ein Model nahe. Dagegen spricht die Existenz von Abhängigkeiten zwischen Stücklisten in der Art, daß in der Regel gemeinsame Teile (Motor oder andere Konstruktionsgruppen) auftreten, die oft logische Einheiten für sich darstellen. Diese sind zu identifizieren und getrennt abzubilden. Weiter soll nach Möglichkeit der konkurrierende Zugriff minimal behindert werden. Grundsätzlich ist die Wahl des Sperrgranulats aber Aufgabe des zugrundeliegenden DBS, und Models sollten für den Anwender lediglich eine abstrakte Strukturierungshilfe für die Ablage, die Verwaltung und das Anfordern von Daten sein.

Mächtigkeit der SDAI-Sprachschnittstelle SDAI ist bislang vom Aufgabenumfang der einzelnen Funktionen eine recht einfache Schnittstelle, die dem Anwender die ganze Last der Vorgehensweise bei der Bearbeitung der Daten eines Repository aufbürdet. Läßt man die Schemadienste außen vor, so gibt es grob gesprochen drei Arten von Verarbeitungsschritten beim "Durchstöbern" eines Repository:

a) Finden von Einstiegspunkten über einen vorher vergebenen Namen (GetModel).
b) Navigieren von Instanz zu Instanz (GetAttr).
c) Iterieren über eine Menge von Instanzen (Beginning, Next, Previous, End).

Ob diese Operationen für eine effiziente Verarbeitung ausreichen, ist von der jeweiligen Anwendung abhängig. Da STEP wohl insbesondere in großen heterogenen Systemen zum Einsatz kommen soll, werden aber sehr verschiedenartige Anwendungen beteiligt sein. Aus diesem Grund muß die SDAI-Schnittstelle möglichst allgemein sein. So sind selbst einfache Erweiterungen nützlich. Bspw. war schon die Erweiterung der Iteration um "Previous" für Stücklistenanwendungen effizienzsteigernd. Im wesentlichen genügten die von SDAI bereitgestellten Möglichkeiten zur Realisierung unseres Projektes. Vermißt wurden jedoch assoziative Zugriffsmöglichkeiten, wie z.B. auf Aggregatelemente über die Spezifikation eines Attributwertes. Die jüngst eingeführten SDAI-Queries befriedigen prinzipiell diese Anforderung. Jedoch ist eine Leistungssteigerung erst mit dem Einsatz von Indizes (nicht von EXPRESS und SDAI abgedeckt) zu erwarten. Kommt ein Puffer zum Einsatz, so daß die Anfrageverarbeitung auch auf dem Hauptspeicher abläuft, ist zu beachten, daß gegenüber der externspeicherbezogenen Verarbeitung die Effizienz durch ganz andere Faktoren bestimmt wird (siehe 4.3). Jeweils zugeschnittene Speicherungs- und Zugriffspfadstrukturen und auch Satzformate können hier wesentliche Effizienzgewinne bringen.

Benutzung der Entity-Typen in der Anwendung Wir haben für die Implementierung das E-Binding für C++ benutzt. Die Klassen, die die Entity-Typen modellieren, möchte man in der Regel um anwendungsbezogene Funktionalität erweitern. Dies wird von EXPRESS nicht direkt unterstützt (es gibt jedoch auch hier anderweitige Bemühungen [9]). In SDAI erhält man stattdessen für jeden Entity-Typ eine C++-Klasse mit der in SDAI allgemein festgeschriebenen Basisfunktionalität. Eine naheliegende Lösung zur Erweiterung der Funktionalität ist der Einsatz von Vererbung: es werden bspw. nur um Methoden erweiterte Klassen von den generierten Klassen abgeleitet. So kann ein wichtiger Aspekt für Sprachintegration erfüllt werden: möglichst natürlich zusätzliche Fähigkeiten in eine Sprache zu integrieren.

Ausnutzung von Verarbeitungskontexten Das Wissen um Verarbeitungskontexte ist für Optimierungen und Vereinfachungen ausnutzbar. Oberhalb der SDAI-Schnittstelle kann dies nur durch die Abbildung der Anwendungsdaten auf SDAI-Models geschehen. Da sich aber das Zugriffsverhalten innerhalb eines Model und auch die Verarbeitungskontexte verschiedener Anwendungen oft sehr unterscheiden, muß man bei der Abbildung Kompromisse eingehen. In SDAI integrierte Anfragemöglichkeiten wären zur dynamischen Kontextspezifikation nutzbar, wodurch eine Abbildung auf Models prinzipiell überflüssig würde.

6. Ausblick

Neben der Mächtigkeit der SDAI-Schnittstelle ist die Einschätzung ihrer Performanz aus Anwendersicht besonders wichtig. Offensichtlich muß man bei der Verwendung von SDAI mit Effizienzeinbußen rechnen. Die Frage ist, ob diese vertretbar bzw. zu verkraften sind. Mit der Prototypimplementierung sollen die Voraussetzungen geschaffen werden, detaillierte Messungen in dieser Richtung zu unternehmen. Gleichzeitig soll auch die Umsetzung von Benchmarks, die gegenüber unserer Beispielanwendung anwendungsunabhängig sind oder sich zumindest auf einen größeren Anwendungsbereich beziehen, Aufschluß über die Effizienz von SDAI geben. Wir erhoffen uns durch den direkten Vergleich einer Anwendung oder eines Benchmarks mit und ohne Nutzung der SDAI-Schnittstelle Resultate, die uns zeigen, wie man mit geschickter Implementierung von SDAI etwaige Effizienzverluste weitgehend ausschalten oder diejenigen Konzepte erkennen kann, die für diese Verluste verantwortlich sind. Weitere Einbußen werden erkauft, wenn der Zugriff auf Instanzen über Schemainformation erfolgt. Interessant wäre es zu sehen, welche Einbußen allein durch das Ersetzen der E-Binding-Funktionen durch die äquivalenten L-Binding-Funktionen in einer Anwendung anfallen und in welchem Verhältnis diese Einbußen zum Overhead von SDAI stehen.

Wir wollen ebenfalls unsere SDAI/DB-Anwendung auf unterschiedlichen Datenhaltungssystemen realisieren. Hierbei interessieren wir uns nicht nur für Laufzeitunterschiede, sondern auch für evtl. auftretende Schwierigkeiten bei der Portierung. Erst die Umsetzung auf ein anderes System wird zeigen, wo der Standard noch Probleme aufweist. Der nächste und letzte Schritt in diese Richtung ist dann, eine Applikation über eine SDAI-Schnittstelle auf zwei oder mehr unterschiedlichen DBS laufen zu lassen, so daß innerhalb eines Prozesses Daten aus beiden Systemen manipuliert und ausgetauscht werden.

7. Literatur

[1] R. Anderl: STEP - Grundlagen der Produktmodelltechnologie. In W. Stucky, A. Oberweis (Ed.), GI-Fachtagung für Büro, Technik und Wissenschaft, 1993, pp. 33-53.

[2] M. J. Carey, D. J. DeWitt, J. F. Naughton: The OO7 Benchmark. In ACM SIGMOD, 1993, pp. 12-21.

[3] M. J. Carey, M. J. Franklin, M. Zaharioudakis: Fine-Grained Sharing in a Page Server OODBMS. In ACM SIGMOD 1994, pp. 359-370.

[4] D. J. DeWitt, P. Futtersack, D. Maier, F. Velez: A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. In VLDB 1990.

[5] A. Herbst: Long-Term Database Support for EXPRESS Data. In Proc. 7th Int. Working Conf. on Scientific and Statistical Database Management, Charlottesville, Virginia, 1994.

[6] A. Heuer: Objektorientierte Datenbanken: Konzepte, Modelle, Systeme. Addison-Wesley, 1992.

[7] T. Härder, B. Mitschang, U. Nink, N. Ritter: Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen. Informatik Forschung und Entwicklung, 1995.

[8] A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases. Proc. Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 155-162.

[9] M. Koethe, A. Nieva, F. Schönefeld: Product Data Exchange in Open Systems: The PISA Approach. In STAK, Ilmenau, Deutschland, März 1994, pp. 101-119.

[10] F. Leymann: Towards The STEP Neutral Repository. In Proc. of the 4th Int. Conference on CALS and Information Management in Europe, Berlin, Deutschland, 1993.

[11] C. Lamb, G. Landis, J. Orenstein, D. Weinreb: The ObjectStore Database System. Communications of the ACM, 34(10), Oktober 1991, pp. 50-63.

[12] T. J. Lehman, E. J. Shekita, L.-F. Cabrera: An Evaluation of Starburst's Memory-Resident Storage Component. Report RJ 8919, IBM Almaden Research Center, 1992.

[13] K. C. Morris: Translating Express to SQL: A Users's Guide. National Institute of Standards and Technology, U.S. Department of Commerce, 1990.

[14] ISO TC184/SC4/WG5: Product Data Representation and Exchange - Part 11: EXPRESS Language Reference Manual, ISO TC184/SC4/WG5 N55, Januar 1994.

[15] ISO TC184/SC4/WG7: Product Data Representation and Exchange - Part 22: Standard Data Access Interface, ISO TC184/SC4/WG7 N370 Committee Draft, November 1994.

[16] ST-Developer 1.3 Reference Manuals. Step Tools Inc. 1994.

[17] VERSANT Release 2 System Reference Manuals. July 1993.