Software-Technologie

Diplomarbeit:
Integration von Persistenzkonzepten in eine Prototyping-Sprache und Realisierung mit Hilfe eines Nicht-Standard-Datenbanksystems

Christian Kappert
Terkampstr. 5
45884 Gelsenkirchen

Gutachter: Prof. Dr. Ernst-Erich Doberkat
Prof. Dr. Udo Kelter

3. Mai 1995

UNI DO

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Aufgabenstellung der Diplomarbeit

Die mengentheoretisch-orientierte Prototyping-Sprache ProSet [DFG+92a] unterstützt die Modellierung von Programmen und von Daten. Hierzu stellt die Programmiersprache neben den üblichen primitiven Typen Mengen, Tupel, Funktionen und Module als grundlegende Datentypen zur Verfügung. Desweiteren bietet ProSet die Möglichkeit, Daten persistent zu machen, d.h. einmal berechnete Daten oder Programmeinheiten über eine Programmausführung hinweg zu erhalten und sie anderen Programmen zugänglich zu machen. Operationen auf persistenten Daten werden in ProSet im Rahmen von geschachtelten Transaktionen ausgeführt. Zudem werden Mittel zur expliziten Parallelprogrammierung zur Verfügung gestellt.

Prototyping ist Teil eines umfassenden Software-Entwicklungsprozesses, der durch eine offene, integrierte Software-Entwicklungsumgebung (SEU) unterstützt werden sollte. Als eine wesentliche Komponente einer solchen SEU soll deshalb eine Prototyping-Umgebung auf Basis der Prototyping-Sprache ProSet angeboten werden. Die Komponenten der SEU sollten in mehrfacher Hinsicht integriert sein. Der internationale Standard *Portable Common Tool Environment (PCTE)* definiert ein *Public Tool Interface (PTI)*, das die Integration der Komponenten einer SEU unterstützt[WJ93]. PCTE definiert u.a. ein *verteiltes, strukturell-objektorientiertes Objektbankmanagementsystem* zur Unterstützung der Datenintegration.

Innerhalb dieser Diplomarbeit sollen die in ProSet bereits integrierten Persistenzkonzepte mit Hilfe des von H-PCTE, einer partiellen Implementierung des PCTE-Standards, zur Verfügung gestellten Nicht-Standard-Datenbanksystems realisiert werden. Darüber hinaus soll eine Analyse der programmiersprachlichen Anforderungen an das Prototyping mit persistenten Daten vorgenommen werden. In diesem Rahmen sollen weitere Persistenzkonzepte betrachtet und im Hinblick auf ihre Eignung für das Prototyping eingeordnet werden. Auf Basis dieser Analyse sollen die bisherigen Persistenzkonzepte dann erweitert, gegebenenfalls in die Sprache integriert und anschließend mit Hilfe des Nicht-Standard-Datenbanksystems von H-PCTE realisiert werden. Dabei sollen auch Randbedingungen, die sich aus den Anforderungen an eine Integration des Systems in eine SEU ergeben, berücksichtigt werden.

Die Persistenzkonzepte sollen schwerpunktmäßig ein Transaktionskonzept für ProSet, ein Schutzkonzept für die persistenten ProSet-Daten, das auch den Mindestanforderungen einer SEU genügt, und ein Konzept für eine geeignete verteilte Speicherung der persistenten ProSet-Daten enthalten. Die Konzepte, insbesondere das Transaktionskonzept, beschränken sich auf sequentielle ProSet-Programme.

Diese Diplomarbeit steht in einer engen Beziehung zu der Diplomarbeit von Jörg Heinrich, in welcher Werkzeuge zur Verwaltung der persistenten Daten entwickelt werden [Hei95].

1.2 Vorgehensweise und Gliederung der Diplomarbeit

Nach dieser Einleitung werden im zweiten Kapitel die Schwerpunktthemen erläutert, um dem Leser einen Überblick über den Kontext und den Inhalt der Diplomarbeit zu geben. Innerhalb dieses Kapitels werden

- die Prototyping-Sprache ProSet vorgestellt,
- der Persistenzbegriff erläutert und definiert,
- die Gründe für die Forderung nach der Integration von Persistenzkonzepten in ProSet beschrieben, aus denen später die ersten Anforderungen an die Persistenz von ProSet abgeleitet werden,
- überblicksartig die bereits bestehenden Persistenzkonzepte von ProSet dargelegt,
- sowie ausführlicher die Konzepte des Nicht-Standard-Datenbanksystems H-PCTE vorgestellt, die zur Realiserung der Persistenzkonzepte von ProSet benötigt werden.

Die folgenden Kapitel lehnen sich eng an die klassischen Phasen des Software-Entwicklungsprozesses an:

Im dritten Kapitel werden die Anforderungen an die Persistenz der Prototyping-Sprache ProSet analysiert. In den letzten Jahren wurden bereits einige grundlegende Persistenzkonzepte für ProSet entwickelt und prototypisch partiell auf Basis einer früheren H-PCTE-Version implementiert [DFKS93]. Aus diesem Grunde werden innerhalb der Anforderungsanalyse zum einen bereits in ProSet integrierte Konzepte erläutert und implizit Erfahrungen, die aus der prototypischen Implementierung gewonnen wurden, miteingebracht. Zum anderen wird eine nochmalige breitere Anforderungsanalyse durchgeführt, um notwendige bzw. wünschenswerte weitere Persistenzkonzepte für ProSet zu finden. Innerhalb dieser Analyse werden auch Randbereiche berücksichtigt, wie z.B. Anforderungen einer Prototyping-Umgebung, die innerhalb der Persistenzkonzepte von ProSet beachtet werden müssen. In diesem Kapitel wird bzw. werden deshalb

- die sich aus der Konzeption von ProSet ergebenden allgemeinen Anforderungen an die Persistenzkonzepte analysiert;
- eine Übersicht über Persistenzkonzepte in Programmiersprachen gegeben, um aus diesen die in ProSet benötigten bzw. wünschenswerten Konzepte abzuleiten und gleichzeitig die bereits bestehenden Persistenzkonzepte von ProSet in diesem Konzeptgerüst einzuordnen;
- Anforderungen analysiert, die sich aus dem Software-Entwicklungsprozeß mit ProSet bzw. aus dem Fernziel der Entwicklung einer Prototyping-Umgebung auf Basis von ProSet ergeben und die in den Persistenzkonzepten von ProSet berücksichtigt werden sollten;
- allgemeine Anforderungen an die Datenhaltung der persistenten ProSet-Werte aufgestellt, die im wesentlichen aus dem Datenbankbereich und dem Anwendungsbereich von ProSet stammen und berücksichtigt werden sollten, um eine Grundlage für Verwaltungswerkzeuge zu schaffen bzw. um den Anwendungsbereich der Prototyping-Sprache ProSet zu erweitern.

Da ein Tansaktionskonzept und ein Schutzkonzept zwei der Schwerpunkthemen dieser Diplomarbeit sind, werden diese in einem separaten Unterkapitel abgehandelt. In diesen Unterkapiteln werden auch Grundlageninformationen zu den beiden Themen gelierfert.

Im vierten Kapitel werden dann der Entwurf der Persistenzkonzepte und im fünften Kapitel die Implementierung der Konzepte präsentiert. Schließlich werden in einem sechsten Kapitel die Erfahrungen mit der Implementierung diskutiert und bewertet

2 Grundlagen

2.1 ProSet

2.1.1 Die Prototyping-Sprache ProSet

ProSet, ein Akronym für ***Pro****totyping with* ***Set****s*, ist eine imperative, *mengentheoretisch-orientierte Breitbandsprache* ([DFG+92a], [DFG+92b]), die in der Nachfolge von SETL ([SDDS86], [DF89]) steht.

ProSet ist eine *Breitbandsprache*, weil sie sowohl mächtige als auch primitive Datentypen und Operationen zur Verfügung stellt. Somit kann man sowohl auf einer sehr hohen Abstraktionsebene eng angelehnt an den konzeptionellen Ideen einer Problemlösung, als auch eher maschinennah auf einer niedrigeren Abstraktionsebene, vergleichbar mit der Programmierung in Ada oder C, programmieren. Wegen der Fähigkeit von ProSet, Programme auf einem hohen Abstraktionsniveau formulieren zu können, wird die Sprache auch zu den ***V****ery* ***H****igh* ***L****evel* ***L****anguages (VHLL)* gezählt.

Die Sprache ProSet ist *mengentheoretisch-orientiert*, weil sie u.a. mächtige Datentypen und Operationen für Mengen und Tupeln anbietet, so daß der Programmierer auf Basis der endlichen Mengenlehre Algorithmen formulieren kann.

ProSet ist in erster Linie als Unterstützung für das Software-Prototyping gedacht. Hierzu stellt ProSet ein schwaches Typkonzept und insbesondere neben den üblichen primitiven Datentypen – wie sie ebenfalls in anderen imperativen Programmiersprachen vorkommen – auch komplexe Datentypen für Mengen, Tupel, Funktionen und Module zur Verfügung.

Außerdem bietet ProSet ein mächtiges Konzept zur Ausnahmebehandlung an. Jeder auftretende Fehler löst eine Ausnahme aus, die von einem vom Programmierer zu definierenden Handler abgefangen und somit adäquat behandelt werden kann.

Ein weiteres wichtiges Merkmal von ProSet ist die Möglichkeit der expliziten Parallelprogrammierung [Has92]. Hierzu bietet die Sprache über ProSet-Linda Konstrukte zur Parallelprogrammierung an.

2.1.2 Der Übersetzungsvorgang eines ProSet-Programms

Der Übersetzungsvorgang eines ProSet-Programms in ausführbaren Code wird in zwei Schritten durchgeführt. Im ersten Schritt wird mit Hilfe des ProSet-Compilers aus dem ProSet-Programm ein ANSI-C-Programm generiert, das dann in einem zweiten Schritt mit Hilfe eines C-Compilers (z.Zt. Gnu-C) in Objekt-Code übersetzt wird. Anschließend wird dieser Objekt-Code mit der ProSet-Laufzeitbibliothek zu einem ausführbaren Programm gebunden. Die ProSet-Laufzeitbibliothek ist ebenfalls in C implementiert.

Der Übersetzungsvorgang vom ProSet-Programm über den „C-Zwischencode" zum ausführbaren Code ist an Hand der Abbildung 2-1 schematisch dargestellt. Eine genauere Beschreibung der Struktur des ProSet-Compilers findet man in [DFG+92b].

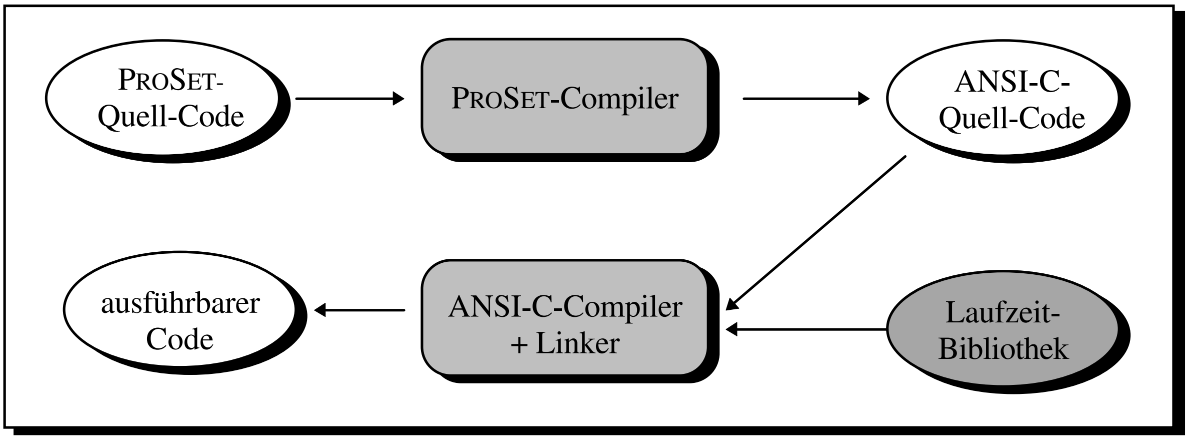


Abbildung 2-1: Übersetzung eines PROSET-Programms

2.2 Persistenz

Nach diesem Überblick über PROSET soll nun erläutert werden, was man eigentlich unter dem Begriff Persistenz, insbesondere im Rahmen von Programmiersprachen, versteht.

2.2.1 Der Persistenz-Begriff

Der Begriff der Persistenz wird im Zusammenhang mit der Lebensdauer von Daten verwendet. Persistenz kann als eine Art Abstraktion des Speichers angesehen werden, in der die Unterscheidung zwischen dem Hauptspeicher und dem externen (stabilen) Speicher aufgehoben wird und man nur definieren muß, wie lange ein Datenobjekt aufrechterhalten bleiben soll ([Heu92], Kap. 9.2).

Persistenz: Nach Atkinson [Atk91] bezeichnet *Persistenz* die Fähigkeit der Daten eine längere Lebensdauer anzunehmen als das Programm, das sie erzeugt hat, d.h. die Daten existieren auch noch nach der Beendigung ihres zugehörigen Programms. Diese Eigenschaft steht im Gegensatz zu flüchtigen (*transienten*) Daten, die nur während der Ausführung des Programms existieren.

Die Lebensdauer von Daten entspricht normalerweise der Ausführungszeit ihres Programms. Besteht jedoch die Möglichkeit, diese Daten persistent zu machen, so kann mit dem gleichen oder einem anderen Programm auch zu einem späteren Zeitpunkt mit den gleichen Daten weitergearbeitet werden.

Gewöhnlich werden Daten, die eine Persistenzeigenschaft besitzen, *persistente Daten* genannt. Klassische Beispiele für Arten von (grobgranularen) persistenten Daten aus dem Bereich des Software-Engineering sind Quell-Code, bindefähiger Objekt-Code und ausführbarer Code. Derartige Daten wurden ursprünglich innerhalb von Dateien abgelegt und somit über ein Dateisystem verwaltet. Jedoch ist es gerade im Bereich der Software-Entwicklung auch erforderlich, Daten von feinerer Granularität persistent abzulegen. Darunter fallen neben den typischen in einer Entwicklungsumgebung benötigten Informationen, wie Symboltabellen und attributierten abstrakten Syntaxbäumen, ebenfalls die innerhalb eines Programms berechneten Daten.

Da im weiteren Verlauf der Diplomarbeit zwischen den von einem PROSET-Programm erzeugten persistenten und transienten Daten und den in der Objektbank verwalteten Daten zur Realisierung der Persistenz unterschieden werden soll, seien folgende Begriffe hiermit definiert:

- *persistente Datenobjekte* bezeichne allgemein Daten, die im Rahmen der Persistenz in Programmiersprachen anfallen. D.h. hierzu zählen nicht nur die von einem Programm erzeugten persistenten Daten, sondern auch die in der Objektbank zur Realisierung der Persistenz in Programmiersprachen unterhaltenen Hilfstrukturen, wie z.B. Datencontainer.
- *ProSet-Werte* bezeichne die von einem ProSet-Programm erzeugten – transienten und persistenten – Werte.
- *transiente ProSet-Werte* bezeichne die von einem ProSet-Programm erzeugten transienten Werte.
- *persistente ProSet-Werte* bezeichne die von einem ProSet-Programm erzeugten persistenten Werte.

2.2.2 Motivation für eine Integration von Persistenzkonzepten in ProSet

In diesem Abschnitt sollen nun die wichtigsten Gründe für eine Integration von Persistenz in ProSet erläutert werden. Die Gründe stammen hauptsächlich aus den allgemeinen Anforderungen an die Software-Konstruktion und insbesondere aus den Anforderungen an das Prototyping als Teil eines umfassenden Software-Entwicklungsprozesses.

Wie bereits erwähnt, ist als Einsatzbereich für ProSet das *Prototyping* mit seinen verschiedenen Ausprägungen vorgesehen (siehe [DF89] und [BKKZ92]). Den diversen Prototyping-Arten gemeinsam ist das systematische Ausprobieren und Entwickeln von algorithmischen Lösungen. Daraus folgt, daß ProSet insbesondere die Konstruktion von ausführbaren Modellen unterstützen soll. Gerade das Kriterium der Ablauffähigkeit ist hier von Bedeutung, da an diesen Modellen die Eigenschaften von Algorithmen und Programmen studiert werden sollen, um so zu einem Feed-Back für die Anforderungsdefinition zu gelangen.

Die primären Ziele eines jeden Software-Entwicklungsprozesses sind das Produktionsprogramm möglichst schnell (und somit kostengünstig) und korrekt zu konstruieren. Bei Anwendung des Prototypings konzentriert sich das Interesse jedoch auf die schnelle und korrekte Konstruktion (Extremform: *Rapid Prototyping*) eines Prototypen, der alle wesentlichen Merkmale des endgültigen Produkts aufweist und somit als Kommunikationsbasis zwischen Anwender und Programmierer dienen soll. Mit Hilfe des Prototyps können somit in Zusammenarbeit mit dem Anwender Anforderungen konkretisiert und modifiziert werden. Aus diesem Grunde muß eine Programmiersprache außerdem eine schnelle Modifikation des Prototypen unterstützen.

Um die Effizienz der Konstruktion eines Prototypen oder auch eines Produktionsprogramms zu steigern, sind zwei wesentliche Aspekte zu beachten ([DFKS93], [ES89]):

- Die *Wiederverwendbarkeit von Software-Bausteinen*, z.B. aus der unternehmensinternen Produktion, muß durch eine entsprechende Speicherung innerhalb eines Repositorys unterstützt werden. Nur so kann man auf bereits bestehende Problemlösungen zurückgreifen und aus diesen dann im optimalen Fall bereits durch entsprechende Kombination der Bausteine den Prototypen konstruieren.
- Es sollten möglichst *mächtige Sprachkonstrukte* über die Prototyping-Sprache zur Verfügung gestellt werden, damit der Programmieraufwand zur Erstellung von noch nicht

existierenden Problemlösungen für spezielle Anwendungsbereiche reduziert werden kann (Konstruktion von projektspezifischen Modulen).

Gerade der zweite Aspekt erfordert natürlich auch eine in möglichst vielen Anwendungsbereichen einsetzbare Prototyping-Sprache. Neben den mächtigen Datentypen und Operationen, die ProSet bereits zur Reduzierung des Programmieraufwands anbietet, werden deshalb auch Konzepte benötigt, um die Software-Bausteine, zu denen in diesem Zusammenhang berechnete Daten, konstruierte Datenstrukturen und konstruierte Algorithmen bzw. Programmeinheiten gezählt werden sollen, persistent zu speichern und somit wiederverwenden zu können.

So kann man mit Hilfe der Persistenz diese Software-Bausteine dem Programmierer prinzipiell zu jedem Zeitpunkt des Software-Entwicklungsprozesses verfügbar machen und außerdem den Anwendungsbereich der Sprache erweitern (insbesondere in Richtung Datenbankanwendung).

Ein weiterer wichtiger Gesichtspunkt zur Verminderung des Konstruktionsaufwandes ist die Erleichterung des Datentransportes. Nach Atkinson [Atk91] besteht ein Drittel des Codes der meisten kommerziellen Anwendungen aus Operationen zum Datentransport. Dies ist im *Impedance Mismatch*, also Bruch zwischen den Datenstrukturen der Programmiersprache einerseits und dem Datenmodell der Datenbank andererseits, begründet. In traditionellen Programmiersprachen muß der Programmierer selbst für diese Überwindung des Impedance Mismatch sorgen. Dieser Aufwand kann durch Einführung der Persistenz als eine geeignete Form der Abstraktion zwischen Hauptspeicher und dem stabilen externen Speicher reduziert werden, da so zumindest für den Benutzer der Programmiersprache kein Impedance Mismatch mehr besteht (Abbildung 2-2).

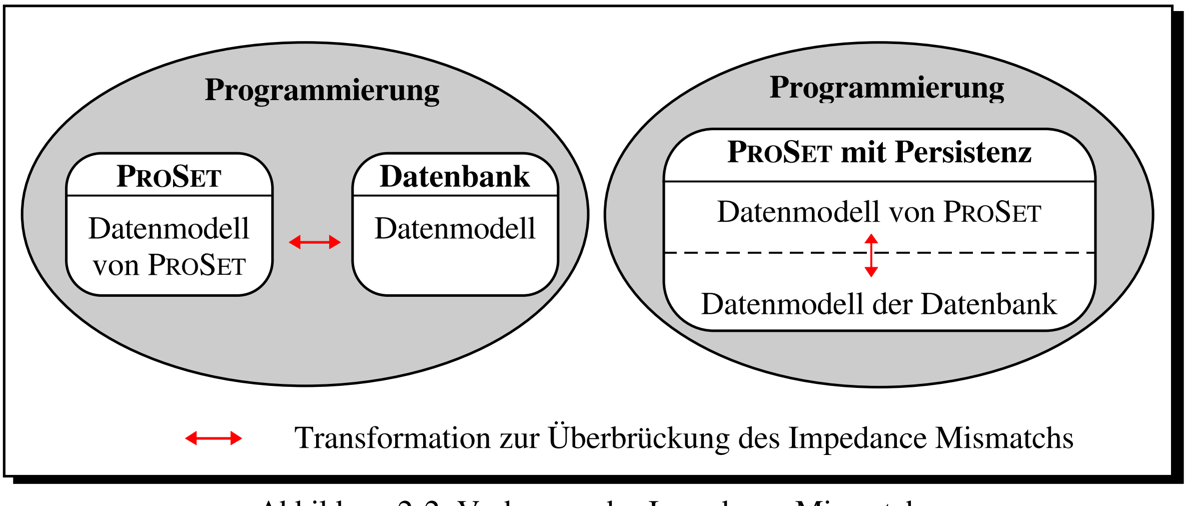


Abbildung 2-2: Verbergen des Impedance Mismatch

Ein weiterer Grund für eine Forderung nach Persistenz in ProSet leitet sich aus einem eher methodischen Ansatz des Prototypings mit ProSet ab: Insbesondere im Rahmen des evolutionären Prototypings soll nämlich ein *transformationeller Ansatz* beim Prototyping mit ProSet unterstützt werden.

Der transformationelle Ansatz von ProSet besteht darin, daß die Problemlösungen, die mit Hilfe der mächtigen Sprachkonstrukte von ProSet auf einem hohen semantischen Niveau formuliert wurden, auf Sprachebenen mit einem niedrigeren semantischen Niveau transformiert werden können. Innerhalb dieses Transformationsprozesses werden so die Vorteile beim Gebrauch einer Very-High-Level-Language, wie z.B. die schnelle Konstruktion des Prototypen, mit den Vorteilen einer maschinennahen Programmierung, wie z.B. die Erzeugung eines effizienteren Codes vereint. So soll aus dem ersten Prototyp, der zwar schnell erstellt werden konnte, aber aufgrund des Gebrauchs von mächtigen Sprachkonstrukten im allgemeinen aus ineffizientem Code besteht (Mächtigkeit der Konstrukte erfordert Generalität, Generalität einer Operation verursacht im Vergleich zu speziellen Operationen ineffizienten Code), das Endprodukt mit einem effizienteren Code durch geeignete Transformationsschritte konstruiert werden.

Dies bedingt wiederum, daß man nicht nur die verwendeten Algorithmen iterativ verfeinern kann, sondern auch – wie beim klassischen Ansatz des Datenbankentwurfs das Datenmodell – die definierten ProSet-Datenstrukturen iterativ verfeinern kann. Hierzu müssen sowohl Datenstrukturen als auch Algorithmen persistent gespeichert werden können.

Als Ergebnis dieser aus den verschiedenen Blickwinkeln betrachteten Anforderungen ergibt sich, daß die *Wiederverwendbarkeit* von berechneten Daten und bereits konstruierten Datenstrukturen und Algorithmen durch eine entsprechende persistente Speicherung sichergestellt werden muß. Hierbei ist speziell die Wiederverwendbarkeit von Algorithmen multilingual zu sehen, da prinzipiell jeder Software-Baustein, gleichgültig in welcher Sprache er programmiert wurde, von ProSet aus zugänglich sein sollte.

2.2.3 Die Persistenz in ProSet

Im folgenden werden die Persistenzkonzepte von ProSet und der Entwicklungsstand bei der Integration dieser Konzepte in ProSet vorgestellt.

2.2.3.1 Übersicht über die Persistenzkonzepte von ProSet

Die Persistenz in ProSet ist orthogonal zu anderen Eigenschaften von ProSet-Werten. In ProSet kann jeder Wert mit Bürgerrechten erster Klasse persistent gemacht werden. *Bürgerrechte erster Klasse* bedeutet, daß die ProSet-Werte eine *Identität* besitzen und zugewiesen werden können.

Die *Datentypen erster Klasse* werden in ProSet wie folgt klassifiziert:

1. *primitive Datentypen*: `boolean`, `integer`, `real`, `string` und `atom`, wobei über einen `atom`-Wert ein weltweit eindeutiger Wert beschrieben wird.
2. *zusammengesetzte Datentypen*: `set` und `tuple`, über die heterogene ungeordnete und geordnete Mengen beschrieben werden können.
3. *Datentypen höherer Ordnung*: `function`, `modtype` und `instance`. Werte von diesem Typ werden nachfolgend näher beschrieben.

Außerdem werden `procedure`, `lambda`, `handler` und `module` zu den *Datentypen zweiter Klasse* gezählt. Detailliertere Informationen zu den verschiedenen Datentypen findet man in [DFG+92a]. Ein persistenter ProSet-Wert kann nur benutzt werden, nachdem er in einer *Programmeinheit* als `persistent` deklariert wurde.

Folgende Typen bilden in ProSet eine Programmeinheit:

- über `function`, `procedure` und `lambda` werden benannte und unbenannte Prozeduren in ProSet primär zur Unterstützung der *Programmierung im Kleinen (PiK)* gebildet,

- über `handler` werden Ausnahmebehandlungsroutinen definiert (dazu später mehr) und
- über `module`, `modtype` und `instance` werden Module zur Unterstützung der *Programmierung im Großen (PiG)* gebildet.

Da Werte vom Typ `function`, `modtype` und `instance` persistent gemacht werden können, ermöglicht ProSet über seine Persistenzkonzepte einen Mechanismus zur separaten Compilierung. Ein Modul kann z.B. dann, nachdem es persistent gemacht wurde, wieder benutzt werden, indem man es vom persistenten Speicher lädt.

In ProSet wird ein abstrakter Datentyp *P-File* angeboten, der als Container für persistente ProSet-Werte dient. In einem P-File kann wiederum ein P-File geschachtelt sein. Ein in einem P-File geschachteltes P-File wird *Sub-P-File* genannt. Somit kann eine hierarchische Containerstruktur modelliert werden, wobei jeder P-File bzw. Sub-P-File einen eigenen Namensraum für persistente ProSet-Werte bietet. Auf die persistenten ProSet-Werte eines solchen P-Files bzw. Sub-P-Files können dann ProSet-Programme oder auch Werkzeuge einer späteren Prototyping-Umgebung zugreifen. In Erweiterung bzgl. einer Datenintegration von Daten der Werkzeuge einer Prototyping-Umgebung können in einem solchen P-File auch spezielle Daten der Werkzeuge gespeichert werden. Die Abbildung 2-3 zeigt ein Beispiel-Szenario für einen Zugriff auf P-Files von verschiedenen ProSet-Programmen und Werkzeugen aus. Detailliertere Informationen zu den Persistenzkonzepten werden innerhalb der in Kapitel 3 ab Seite 24 vorgestellten Anforderungsanalyse gegeben.

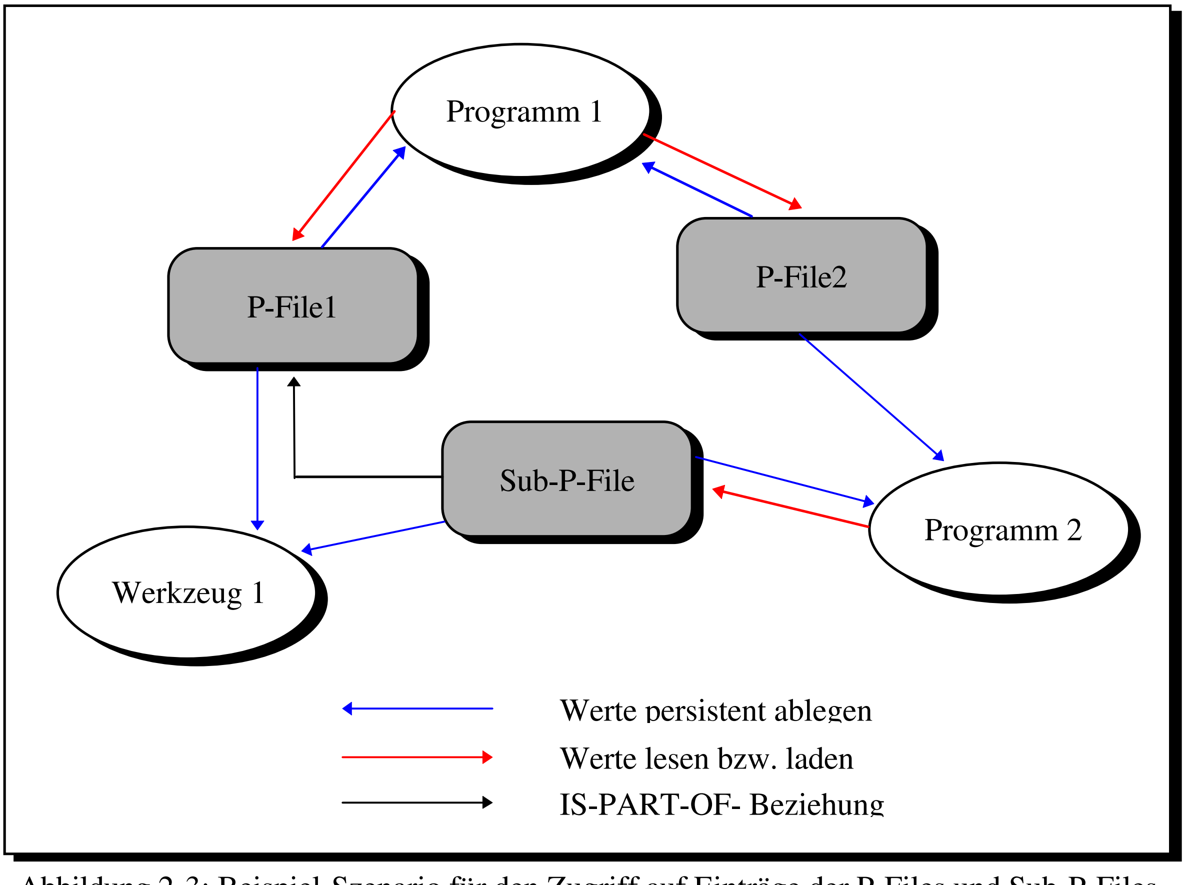


Abbildung 2-3: Beispiel-Szenario für den Zugriff auf Einträge der P-Files und Sub-P-Files

Ein im Zusammenhang mit der Persistenz von ProSet ebenfalls wichtiges Konzept stellt die *Ausnahmebehandlung* von ProSet dar: Die Ausnahmebehandlung ermöglicht die Behandlung von Situationen, die zur Laufzeit eines Programms auftreten, und deren Nichtbehandlung die Zuverlässigkeit des Programms beeinträchtigt. Um darüber hinaus die Strukturierung und Modellierung unterstützen zu können, wurde die Ausnahmesituation von deren Behandlung durch einen *Handler* strikt getrennt. Nach der Behandlung der Ausnahme kann die Programmeinheit, die diese Ausnahme ausgelöst hat, wieder aufgenommen oder beendet werden. Nähere Informationen zur Ausnahmebehandlung findet man in [DFG+92b], Kapitel 8.

2.2.3.2 Entwicklungsstand

Herr Prof. Dr. Ernst-Erich Doberkat hat bereits 1990 (publiziert 1992 in [Dob92]) erste allgemeine Ansätze für eine Integration der Persistenz in ProSet formuliert, die dann konkret 1992 als erste Persistenzkonzepte innerhalb der Sprachdefinition von ProSet [DFG+92a] eingeführt wurden.

Anfang 1993 wurden dann die Persistenzkonzepte von ProSet auf Basis einer PCTE-Implementierung partiell implementiert [DFKS93]. Die PCTE-Implementierung basierte allerdings nicht auf dem heutigen standardisierten ECMA-PCTE, sondern auf eine der nicht weitergeführten Weiterentwicklungen des ECMA-PCTE-Vorgängers. Genauere Informationen bzgl. der Historie von PCTE kann man [WJ93] und [Kel94] entnehmen.

2.3 Das Nicht-Standard-Datenbanksystem H-PCTE

Zur Datenintegration der Komponenten der angestrebten SEU und zur Realisierung der Persistenzkonzepte von ProSet hat man sich für das über H-PCTE zur Verfügung gestellte Nicht-Standard-Datenbanksystem entschieden. Im folgenden wird deswegen der Standard PCTE und das darauf basierende H-PCTE vorgestellt und die wichtigsten Gründe für die Entscheidung für H-PCTE erläutert. Außerdem werden die von H-PCTE angebotenen Dienste beschrieben, sofern sie im Zusammenhang mit der Diplomarbeit von Relevanz sind.

2.3.1 Der Standard PCTE

Seit einigen Jahren besteht ein wesentlicher Forschungsschwerpunkt der Software-Technologie darin, innerhalb von *integrierten Software-Entwicklungsumgebungen* leistungsfähige Werkzeuge anzubieten, die eine computer-gestützte Software-Entwicklung (Computer Assisted Software Engineering, CASE) unterstützen. Um die Konstruktion solcher integrierten SEUen zu erleichtern, wurde das Konzept einer dedizierten Betriebssystemschnittstelle für SEUen entwickelt.

PCTE, eine Abkürzung für ***P**ortable **C**ommon **T**ool **E**nvironment*, ist eine solche dedizierte Betriebssystemschnittstelle für SEUen. Genauer gesagt ist PCTE eine von der *ECMA*, der ***E**uropean **C**omputer **M**anufacturers **A**ssociation*, im Dezember 1990 international standardisierte Spezifikation eines ***P**ublic **T**ool **I**nterfaces (PTI)* für ein offenes Repository. Außerdem ist PCTE mittlerweile ein ISO-Standard und somit der zur Zeit einzige ISO-Standard für ein PTI. Ein PTI bzw. eine dedizierte Betriebssystemschnittstelle ist eine spezifizierte Schnittstelle, die über ihre Funktionen eine Konstruktion von portablen Werkzeugen einer SEU unterstützt. In einem *Repository* werden alle in einer SEU benötigten Informationen gespeichert. Innerhalb der Spezifikation werden eine Reihe von Funktionen definiert, die als Basis für eine Konstruktion von portablen und integrierten CASE-Werkzeugen dienen sollen.

Die Integration der Werkzeuge soll hierbei auf zwei Arten (ursprünglich auf drei Arten, aber bei der Präsentationsintegration hat man sich auf den X-Window-Standard geeinigt) unterstützt werden:

1. Die *Datenintegration* soll die gemeinsame Nutzung von Daten der Werkzeuge in der SEU unterstützen.
2. Die *Kontrollintegration* soll die Kooperation und Kommunikation zwischen unabhängigen Werkzeugen der SEU ermöglichen.

Das von PCTE definierte Repository besteht aus einem transparent verteilten, strukturell-objektorientierten [Dit86] Nicht-Standard-Datenbanksystem, in deren Objektbank alle Informationen einer SEU persistent gehalten werden können und das grundsätzliche Datenverwaltungsfunktionen, wie z.B. Transaktionen und Schutzmechanismen, zur Verfügung stellt.

Ein Kritikpunkt von PCTE ist, daß es zwar die Integration von grobgranularen Daten unterstützt, aber die Integration von feingranularen Daten auf Basis dieser Spezifikation zu ineffizient implementierbar ist. Dies ist zum einen in dem starken Overhead an „Verwaltungsattributen", die jedes Objekt standardmäßig unterhält, begründet und zum anderen in der sehr umfangreichen Funktionalität von PCTE, die, wenn man sie vollständig implementiert, zu Effizienzverlusten bzgl. der Laufzeit und des Speicherplatzbedarfs führen kann (Stichwort: Trade-Off zwischen Funktionalität und Effizienz von Repositorys, siehe [ES89]).

Um u.a. auch feingranulare Daten effizient in dem PCTE-Repository speichern zu können entwickelt die Forschungsgruppe unter Leitung von Herrn Prof. Dr. Udo Kelter an der Universität Siegen zur Zeit eine hochperformante PCTE-Implementierung, genannt H-PCTE, die im nachfolgenden Abschnitt vorgestellt wird.

2.3.2 H-PCTE

H-PCTE ist eine hochperformante Implementierung eines Teils des PCTE-Standards, d.h. ein Forschungsschwerpunkt liegt bei diesem Projekt bei der möglichst effizienten Implementierung der Datenmanipulationsfunktionen von PCTE. Man hat im Wesentlichen das in PCTE definierte verteilte, strukturell-objektorientierte *Objektmanagementsystem (OMS)* realisiert. Aus diesem Grunde kann man H-PCTE auch als Nicht-Standard-Datenbanksystem bezeichnen. H-PCTE ist eine *Hauptspeicher-Datenbank*, d.h. die Objektbank wird in den virtuellen Hauptspeicher einer Applikation geladen. Weitere Ausführungen zu diesem Thema werden in Abschnitt 2.3.2.3 ab Seite 15 gegeben.

Performanzmessungen der ersten H-PCTE-Versionen haben ergeben, daß der oben erwähnte Effizienz-Nachteil bei der Speicherung von feingranularen Daten behoben werden konnte und darüber hinaus sehr gute Leistungswerte bzgl. des Datenzugriffs ermittelt wurden.

Die Entscheidung für H-PCTE ist unter anderem im Fernziel der ProSet-Forschungsgruppe begründet: Das Fernziel der Forschungsgruppe unter Leitung von Herrn Prof. Dr. Ernst-Erich Doberkat am Lehrstuhl für Software-Technologie der Universität Dortmund besteht darin, Prototyping als Teil eines umfassenden Software-Entwicklungsprozesses durch eine integrierte SEU auf Basis der Prototyping-Sprache ProSet zu unterstützten. Die Datenintegration soll hierbei auf Basis der PCTE-Implementierung H-PCTE realisiert werden, da PCTE, wie erwähnt, der zur Zeit einzige ISO-Standard für Public Tool Interfaces ist. Außerdem können mit Hilfe von H-PCTE nicht nur die grobgranularen Daten der SEU, wie Quell-Code, Dokumentationen und ähnlichem, gespeichert werden, sondern auch die feingranularen ProSet-Werte effizient persistent abgelegt werden. Ein weiterer Grund ist der geringe Impedance Mismatch zwischen dem Datenmodell von H-PCTE und dem Datenmodell von ProSet, wodurch der Transformationsprozeß beim Datentransport minimiert wird. Darüber hinaus unterstützt H-PCTE über eine C-*API (**A**pplication **P**rogramming **I**nterface)* eine geeignete Anbindung an die C-Laufzeitbibliothek von ProSet.

Da bei der Realisierung der Persistenzkonzepte von ProSet also auf H-PCTE zurückgegriffen wird, werden nun die wichtigsten Merkmale von H-PCTE vorgestellt. An den Stellen, wo H-PCTE von der PCTE-Spezifikation abweicht, werden die Unterschiede kurz erwähnt. Desweiteren sei hier angemerkt, daß spezielle Begriffe zu den Themen diskretionärer Zugriffsschutz und Transaktionen in Kapitel 3 ab Seite 24 im Zusammenhang der Anforderungsanalyse erläutert werden.

2.3.2.1 Das Datenmodell von H-PCTE

Im folgenden wird das Datenmodell detailliert erläutert, da die Informationen zum Verständnis des Entwurfs und der Implementierung der Persistenzkonzepte notwendig sind. Das Datenmodell von H-PCTE ist strukturell-objektorientiert und kann als Erweiterung des (binären) *Entity-Relationship-Modells* [Che76] angesehen werden. Es werden zur Modellierung der Daten *Objekttypen, Attributtypen* und *Linktypen* zur Verfügung gestellt. Instanzen solcher Typen werden im weiteren als *Objekte, Attribute* und *Links* benannt. Das Attribut nimmt eine Sonderstellung ein, da es sowohl im Kontext eines Typs als auch einer Instanz gebraucht wird. Im ersten Fall wird ein Attributname mit dem zugehörigen Attributtyp assoziiert, wogegen im zweiten Fall der Attributname und der Attributwert assoziiert werden. Als Analogie zum Entity-Relationship-Modell, kann man die Objekte den Entitäten, die Attribute den Attributen einer Entität und die Links den Relationen zwischen zwei Entitäten zuordnen.

In PCTE unterscheidet man drei Arten von Objekttyp-Begriffen ([ECM93a], Abschnitte 8.3.1, 8.4.1 und 8.5.1), wobei der im folgenden verwendete Begriff keinem dieser drei Definitionen exakt zugeordnet werden kann. Ein *Objekttyp* definiert

- eine Menge von direkten Attributen,
- eine Menge von direkten Linktypen, deren Instanzen von diesem Objekt ausgehen dürfen und
- ein oder mehrere direkte Elterntypen.

Es wird von direkten Attributen, Eltern-Objekttypen und Linktypen gesprochen, da das Datenmodell eine Mehrfachvererbung vorsieht. Hierbei erbt ein Objekttyp alle Attribute und zulässigen Linktypen eines Vater-Objekttyps oder mehrerer Vater-Objekttypen. Die Objekttyphierarchie hat genau eine Wurzel, nämlich den Objekttyp `object`. Somit besitzen alle Objekttypen die von `object` definierten Attribute und Linktypen. Hierbei sollte beachtet werden, daß diese Vererbung abhängig von der Sichtbarkeit der verschiedenen Typen ist (vergleiche [ECM93a], Abschnitt 8.5.1). Alle Attribute, die von in H-PCTE vordefinierten Objekttypen stammen, werden auch *Standardattribute* genannt. Da der Begriff Objekt innerhalb der Thematik der Diplomarbeit überladen ist, wird im folgenden von *H-PCTE-Objekt* gesprochen, wenn aus dem Zusammenhang nicht eindeutig hervorgeht, welche Semantik dem Objektbegriff zugeordnet ist. Da H-PCTE, wie später noch erläutert wird, auch die Modellierung von komplexen Objekten unterstützt, wird eine Instanz von einem Objekttyp *atomares H-PCTE-Objekt* genannt.

Beziehungen zwischen Objekttypen werden in H-PCTE über einen *Linktyp* beschrieben, der folgendes definiert:

- eine Folge von n Schlüsselattributen ($n \geq 0$),
- eine Menge von Nicht-Schlüsselattributen,
- eine Menge von Objekttypen, auf deren Instanzen ein Link diesen Typs zeigen darf,
- eine Kategorie (Linkart), über die bestimmte semantische Eigenschaften des Links ausgedrückt werden,
- den Umkehrlinktyp,
- seiner Duplikationseigenschaft (legt fest, ob der Link beim Kopieren seines Ausgangsobjekts mitkopiert wird oder nicht) und
- diverse weitere in diesem Zusammenhang nicht so wichtige Eigenschaften (siehe [ECM93a], Kapitel 8).

Genaugenommen wird also eine Beziehung zwischen zwei Objekten über ein Paar gegenläufig gerichteter Links, Link und Umkehrlink genannt, beschrieben. Folglich können nur binäre Beziehungen ausgedrückt werden. Mit Hilfe der im Link- und/oder Umkehrlink-Typ definierten Schlüsselattribute können 1:1-, 1:N-, und M:N-Beziehungen modelliert werden. Jeder Link hat einen Linknamen, der ihn bzgl. seines Ausgangsobjekts eindeutig identifiziert.. Der Linkname kann über folgende EBNF beschrieben werden (siehe [DDLC]):

```
<Linkname> ::= (((<Schlüsselwert_i>".")*<Linktypname>) |
               "."<Linktypname>)
```

Über *Link-Kategorien* können die semantischen Eigenschaften der Links beeinflußt werden. Folgende Link-Eigenschaften sind beeinflußbar:

1. Die *Komponenten-Eigenschaft*: Das Zielobjekt ist Komponente des Ausgangsobjekts.
2. Die *Existenz-Eigenschaft*: Die Existenz mindestens eines Links mit dieser Eigenschaft ist eine notwendige Bedingung für die Existenz seines Zielobjekts. Die Existenz genau eines Links mit dieser Eigenschaft ist Voraussetzung zur Löschung des Zielobjekts.
3. Die *referentielle Integritäts-Eigenschaft*: Die Existenz eines Links mit dieser Eigenschaft ist ein hinreichendes Kriterium für die Existenz des Zielobjekts.
4. Die *Relevanz-Eigenschaft*: Wird ein Link mit dieser Eigenschaft gelöscht oder erzeugt, so gilt sein Zielobjekt als modifiziert.

Link-Kategorie	**Komponenten-Eigenschaft**	**Existenz-Eigenschaft**	**referentielle Integritäts-Eigenschaft**	**Relevanz-Eigenschaft**
`composition`	ja	ja	ja	ja
`existence`	-	ja	ja	ja
`reference`	-	-	ja	ja
`implicit`	-	-	ja	-
`designation`	-	-	-	ja

Tabelle 2-1: Link-Kategorien und ihre Eigenschaften

Wie in Tabelle 2-1 erläutert, werden durch Kombination dieser 4 Eigenschaften in H-PCTE 5 Link-Kategorien angeboten.

Aufgrund der über diese Kategorien beschriebenen Eigenschaften eines Links oder Umkehrlinks, schließen sich gewisse Kombinationen von Kategorien für Link und Umkehrlink aus (siehe [WJ93] oder [ECM93a]). Im weiteren wird aber aus Gründen einer geeigneten Abstraktion ein Link-Paar, das eine Beziehung zwischen zwei Objekten beschreibt, als ein Link betrachtet.

H-PCTE ermöglicht über `composition`-Links, komplexe Objekte zu modellieren. Dabei wird eine Gruppe von atomaren Objekten, die miteinander über `composition`-Links verbunden sind, als komplexes Objekt behandelt. Ein solches Objekt wird im folgenden, analog zum atomaren H-PCTE-Objekt, als *komplexes H-PCTE-Objekt* bezeichnet. Desweiteren wird ein atomares H-PCTE-Objekt eines solchen komplexen Objekts als *Komponentenobjekt* bezeichnet. Hier sei darauf hingewiesen, daß später aus Gründen der Abstraktion eine Teilmenge der atomaren Objekte eines komplexen Objekts ebenfalls als komplexes Objekt bezeichnet wird. Darüber hinaus soll das über einen vordefinierten Navigationspfad zuerst erreichte atomare Objekt eines komplexen Objekts *Wurzelkomponentenobjekt* genannt werden. Diese Begriffsdefinitionen werden in Abbildung 2-4 veranschaulicht

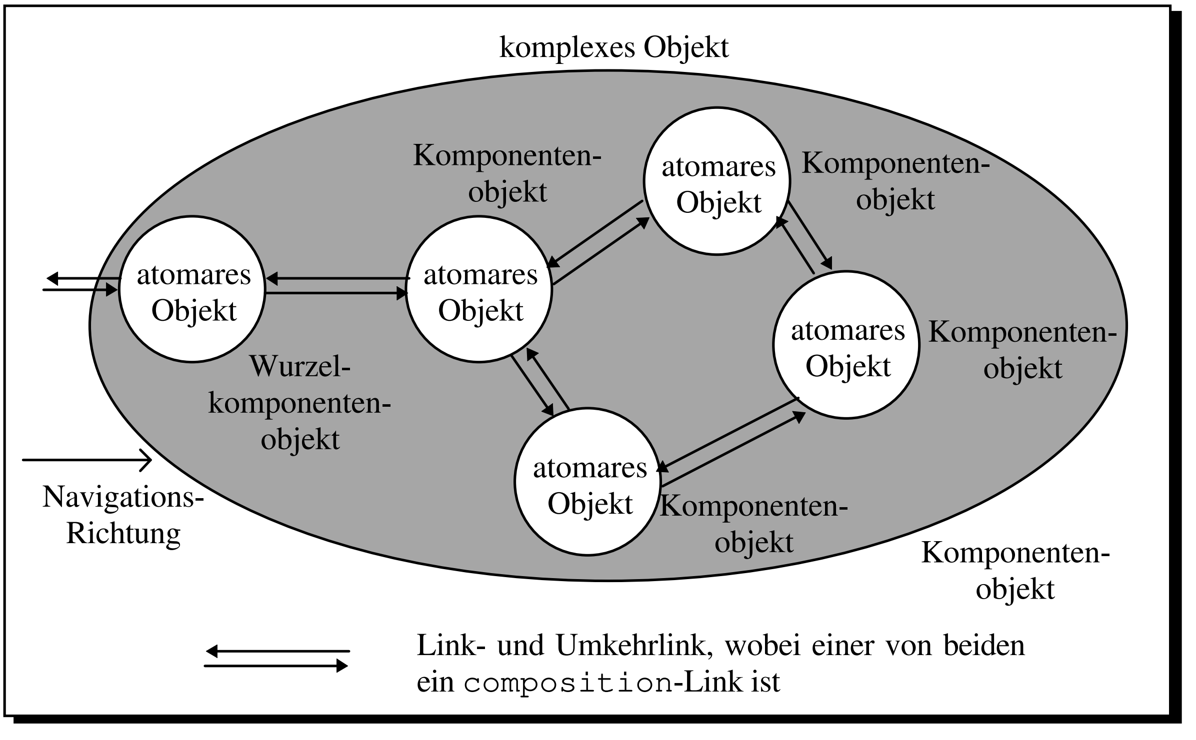


Abbildung 2-4: atomare und komplexe H-PCTE-Objekte

Über *Attribute* können die Eigenschaften sowohl von Objekten als auch von Links beschrieben werden. Hier ist nun ein Attribut als Tupel aus Attributname und Attributtyp gemeint. Ein Attribut eines Objekttyps ist definiert durch:

- seinen Attributnamen,
- seinen Attributtyp (und somit einen Wertebereich),
- einen Initialwert, auf den er bei seiner Instanziierung gesetzt wird und

- seine Duplikationseigenschaft, über die festgelegt wird, ob die Attributwerte beim Kopieren eines Objekts oder Links mitkopiert werden oder nicht.

H-PCTE bietet als Attributtypen `boolean`, `integer`, `natural`, `float`, `string`, `time` und `enumerate` (Aufzählungen) an. Im Gegensatz zu PCTE fungiert der `string`-Attributtyp auch als *langes Feld (long field,* siehe [See94a]), wobei das Attribut wie ein `content`-Attribut in PCTE behandelt werden kann. Derartige Attribute erlauben beispielsweise die Modellierung von Textdateien.

Somit besteht die Objektbank aus einem Netzwerk von Objekten und Links. Der Zugriff auf Objekte geschieht navigierend über Link-Sequenzen. Einen Direktzugriff auf Objekte über beispielsweise die Werte der Schlüsselattribute der Links ist nicht möglich. Ein Objekt kann über einen *Pfadnamen* referenziert werden. Ein Pfadname kann über folgende EBNF beschrieben werden: `<Pfadname> ::= "_"["/"<Linkname>]*`, wobei `"_"` ein spezielles Wurzelobjekt innerhalb der Objektbank einer H-PCTE-Installation bezeichnet

2.3.2.2 Das Schema-Definition-Set und das Working-Schema

Das Datenbankschema von H-PCTE wird über ***S**chema-**D**efinition-**S**ets (SDS)* definiert. Alle in einer H-PCTE-Installation vorhandenen Typen werden innerhalb der Objektbank über Objekte und Links als Metadaten verwaltet, da PCTE *selbstreferentiell* ist. Zur Formulierung eines SDS stellt H-PCTE eine Datendefinitionssprache zur Verfügung. Mit Hilfe eines speziellen Compilers *DDLC (**D**ata **D**efinition **L**anguage **C**ompiler)* werden dann für die innerhalb des SDS definierten Typen die entsprechenden Metadaten in der Objektbank erzeugt.

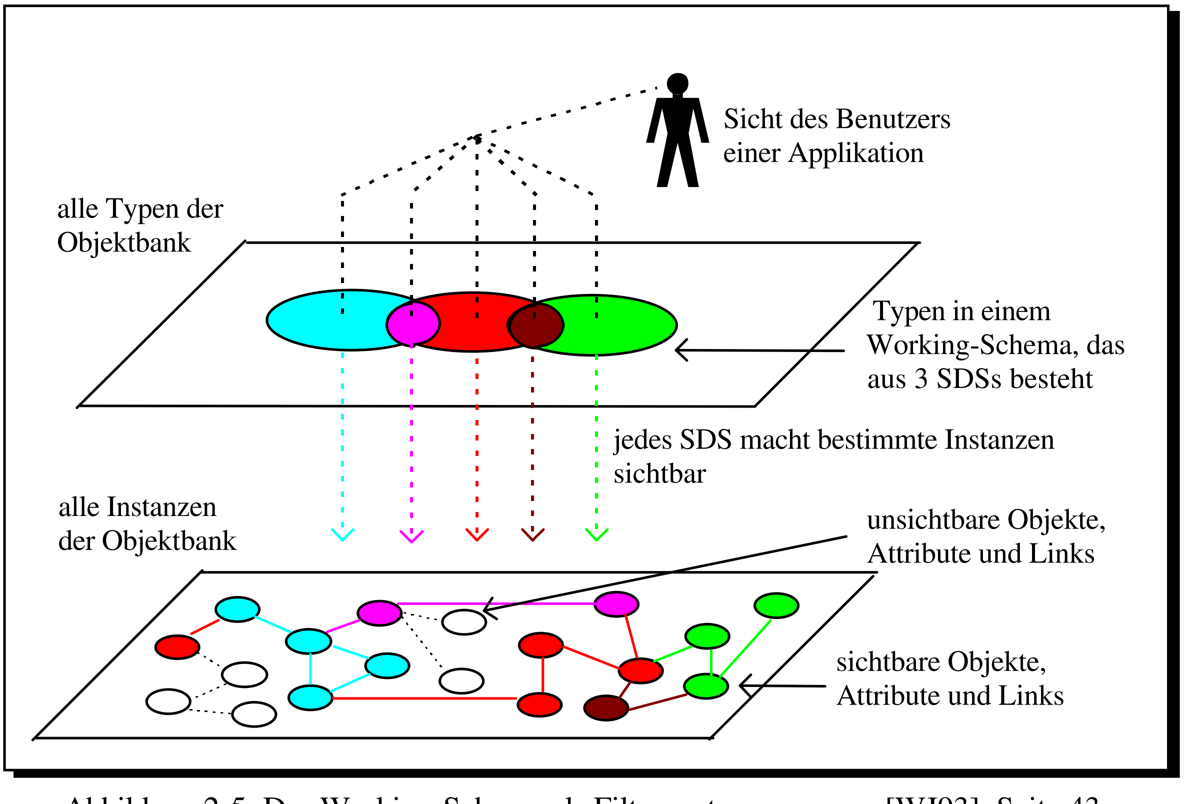


Abbildung 2-5: Das Working-Schema als Filter, entnommen aus [WJ93], Seite 43.

Ein Applikationsprozeß macht dann durch Laden einer entsprechenden SDS-Kollektion bestimmte Komponenten der Objektbank, wie z.B. Objekte und Links, für diese Applikation sichtbar. Denn wie Abbildung 2-5 illustriert sind nur die Objekte, Attribute und Links für den Benutzer einer Applikation sichtbar, deren Typen in einem SDS der geladenen Kollektion definiert sind. Die in einem Applikationsprozeß geladene SDS-Kollektion bildet das *Working-Schema* einer Applikation. Somit bildet die Gesamtheit aller definierten SDS das konzeptionelle Datenbankschema der Objektbank.

2.3.2.3 Verteilte Speicherung und Prozeßarchitektur von H-PCTE

H-PCTE unterstützt eine verteilte Speicherung der Objektbank, d.h. Teile der Objektbank können auf unterschiedlichen Rechnern gespeichert werden. Hierzu werden *Segmente* zur Verfügung gestellt, auf die die Objektbank verteilt werden kann, wobei H-PCTE jedoch von PCTE abweicht: PCTE bietet Volumes an, die jeweils bijektiv auf die vorhandenen physikalischen Datenträger der unterschiedlichen Rechner abgebildet werden. Somit befinden sich Objekte, die auf unterschiedlichen Volumes gespeichert sind, automatisch auch auf unterschiedlichen Datenträgern. In H-PCTE hingegen können mehrere Segmente auf ein Volume abgebildet werden. So können mehrere Objekte, die sich jeweils auf einem anderen Segment befinden, auf dem gleichen Datenträger gespeichert sein.

Ein Segment enthält eine Menge von Objekten und ihre ausgehenden Links, wobei ein atomares Objekt sich immer auf genau einem Segment befindet. Die Segmente werden in H-PCTE auf Betriebssystemebene jeweils einer Datei zugeordnet, wobei die Dateien über ein verteiltes Dateisystem auf verschiedenen Rechnern gespeichert werden können.

Die Segmentierung und somit auch die verteilte Speicherung ist in H-PCTE transparent, d.h. der Benutzer einer Applikation bemerkt sie bzgl. der Funktionalität nicht, da insbesondere der Zugriff auf Objekte über die Links in H-PCTE ortstransparent ist. Falls beim Navigieren ein Objekt erreicht wird, das sich auf einem anderen Segment befindet, wird das Segment automatisch geladen, sofern es noch nicht geladen wurde und es geladen werden kann (siehe Abbildung 2-6). Die Segmente werden selbstreferentiell verwaltet, wobei ein Segment jeweils über ein Objekt repräsentiert wird. Dieses Objekt wird im folgenden als *Segmentobjekt* bezeichnet.

Ein Segment von H-PCTE wird immer vollständig in den Hauptspeicher geladen, wobei das Segment entweder lokal im virtuellen Speicher des Applikationsprozesses (*Client*) geladen werden kann oder global im *Server*-Prozeß. Ein Client kann jeweils nur auf seine lokal geladenen Segmente zugreifen und auf die global im Server geladenen Segmente. Somit ist H-PCTE zur Zeit nur halb-verteilt. Die Prozeßverteilung von H-PCTE basiert auf dem *Workstation-Server-Modell*, einer Modifikation des *Client-Server-Modells* (siehe [DMFV90]). Der H-PCTE-Prozeß, der sich im virtuellen Adreßraum der Applikation befindet wird im folgenden *Client* genannt und der H-PCTE-Prozeß, der als eigenständiger Prozeß die Clients „bedient", wird im folgenden *Server* genannt.

Ein Zugriff auf die im Client geladenen Segmente ist erheblich effizienter, als ein Zugriff auf globale Segmente im Server, da sich die Segmente des Applikationsprozesses in seinem virtuellen Speicher befinden und somit der Interprozeßkommunikationsoverhead entfällt. Abbildung 2-6 veranschaulicht die oben erwähnten Verteilungsaspekte.

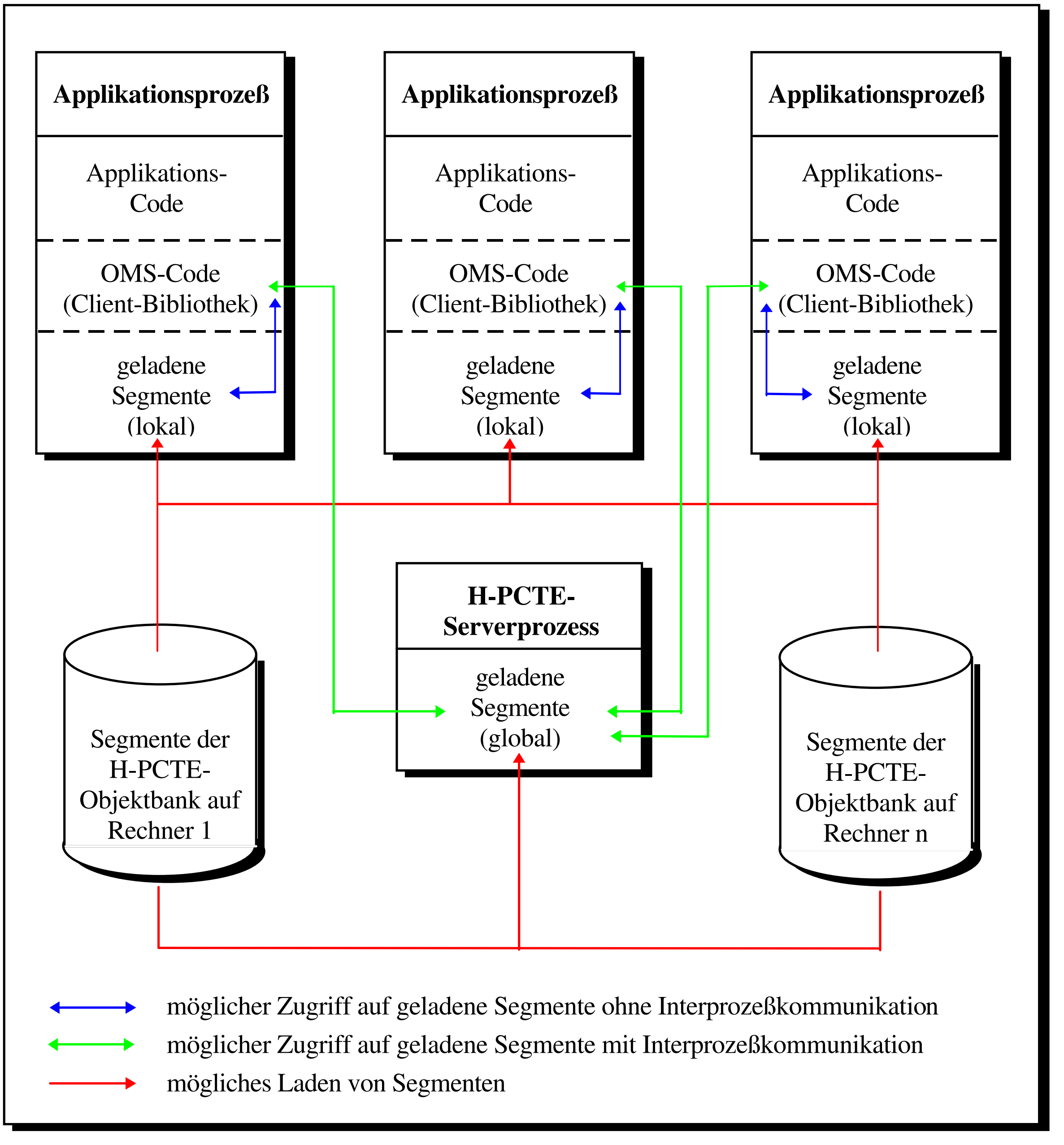


Abbildung 2-6: Prozeßarchitektur und Segmentverteilung [See94b]

Es gibt einige wichtige Regeln beim Umgang mit Segmenten und der Prozeßarchitektur in H-PCTE [Kel94], auf die insbesondere beim Entwurf und der Implementierung der Persistenzkonzepte noch Bezug genommen wird.

- **(H1):** Ein Segment sollte nicht zu groß werden. Tritt dieser Fall ein, sollte das Segment gesplittet werden. (Der Begriff „zu groß" sollte durch Effizienzkriterien bzgl. der Antwortzeiten beim Zugriff auf Segmentinhalte und bzgl. der Speicherauslastung anwendungsbezogen definiert werden).
- **(H2):** Es sollten nur die Daten in den Server geladen werden, auf die mehrere Applikationsprozesse zugreifen müssen. Private Daten sollten nur lokal geladen werden.
- **(H3):** Jeder Benutzer sollte sich ein oder mehrere Segmente einrichten, in denen seine privaten Daten gespeichert werden sollten. Dies kann über einem von H-PCTE speziell

zur Verfügung gestellten „Homedirectory“-Objekt vom Objekttyp `home` geschehen, wie später noch genauer erläutert wird.

- **(H4):** normale Applikationen sollten möglichst nicht dynamisch bei jedem Aufruf ein neues Segment erzeugen, sondern die Verwaltung der Segmente bzgl. einer performanzsteigernden Verteilung speziellen Administrationswerkzeugen überlassen.
- **(H5):** Eine H-PCTE-Installation sollte der Autonomie-Anforderung genügen, d.h., daß ein lokales Arbeiten auf den erreichbaren Segmenten möglich sein sollte. Aus diesem Grunde sollte das *Master-Administrationssegment*, das immer das Segment 0 ist, repliziert werden, wobei jedoch kein Replikationsprotokoll zur Verfügung gestellt wird. Somit dürfen sich nur Objekte auf diesem Segment befinden, die sich nicht verändern. Deshalb sollten alle Verwaltungsobjekte, die zur Realisierung der ProSet-Persistenz benötigt werden, auf einem anderen speziellen Segment gehalten werden.

2.3.2.4 Transaktionen in H-PCTE

Da es zur Zeit in H-PCTE nicht möglich ist Subprozesse aus einem H-PCTE-Prozeß zu starten, sieht H-PCTE im Gegensatz zum PCTE-Standard ein flaches Transaktionsmodell vor, bietet jedoch zusätzlich Sicherungspunkte an, mit deren Hilfe man grundsätzlich auch geschachtelte Transaktionen implementieren kann [Lin91].

Die Transaktionen gehören in H-PCTE zur Klasse der Aktivitäten. Hierbei werden drei Arten von Aktivitäten angeboten: die ungeschützte Aktivität, die geschützte Aktivität und die Transaktion. Nähere Informationen zu den Aktivitätsarten liefert [WJ93] und [ECM93a], Kapitel 16. Nur die Aktivität in Form der Transaktion bietet mit Hilfe der *Concurrency Control-Komponente* Schutz gegen Parallelitätsanomalien und über die *Recovery-Komponente* ein Transaktionsrollback. Diese Aktivität wird benutzt, wenn man die Daten vor nebenläufigen bzw. parallelen Aktivitäten schützen und die Integrität der Objektbank bewahren muß.

Implizit wird einem H-PCTE-Client eine (spezielle) ungeschützte Aktivität zugeordnet. Diese wird automatisch beim Start des zur Applikation gehörenden H-PCTE-Prozesses initialisiert. Beim Beenden des H-PCTE-Prozesses wird diese ebenfalls beendet. Ein höherer Aktivitätsschutz muß explizit über eine API-Funktion von H-PCTE aktiviert werden. Hierzu werden Funktionen zum Starten, Beenden und Abbrechen von Aktivitäten angeboten ([ECM93c], Seite 89, bzw. [H94]). Hierbei werden abhängig von der Aktivitätsart unterschiedliche Maßnahmen ergriffen.

Die Concurrency Control wird über Sperren realisiert, wobei in H-PCTE im wesentlichen sowohl Objekte als auch Links sperrbare *Ressourcen* darstellen. Wird nun über eine API-Funktion von H-PCTE auf eine Ressource zugegriffen, wird versucht vor Ausführung der Funktion auf diese Ressource implizit eine oder mehrere Sperren zusetzen, wobei die Art des Zugriffs und die Klasse der Aktivität entscheidet, welchen Modus die Sperre hat, die gesetzt werden soll. In H-PCTE werden die angebotenen Zugriffsfunktionen in drei Zugriffsarten unterteilt: in Lese-, Schreib- und Löschoperationen [Lin91].

Explizit kann man Sperren über API-Funktionen von H-PCTE setzen, wobei man auf diesem Weg nur Objekte und keine Links sperren kann. Die einzelnen Modi einer Sperre bestimmen, in welcher Art die Ressource gesperrt wird und mit welchen anderen Sperren diese Sperre verträglich ist. H-PCTE stellt verschiedene Sperrmodi auf Ressourcen zur Verfügung (siehe Kapitel 16.1.3 in [ECM93a]).

Prinzipiell erbt eine explizit gestartete Aktivität alle Sperren, die innerhalb der initialen ungeschützten Aktivität gesetzt wurden. Wird innerhalb einer Aktivität (implizit oder explizit) eine Sperre gesetzt, so muß diese mit allen bereits auf der Ressource gesetzten Sperren verträglich sein. Erst bei Beendigung oder Abbruch einer Aktivität werden automatisch implizit alle von einer Aktivität gehaltenen Sperren freigegeben. Darüber hinaus besteht auch die Möglichkeit explizit die Sperren über spezielle API-Funktionen freizugeben.

Weiterhin erkennt H-PCTE als Erweiterung zu PCTE zwar einen Deadlock, jedoch bricht H-PCTE daraufhin die Deadlock-verursachende API-Funktion von H-PCTE mit einem Fehlercode ab.

H-PCTE stellt zusätzlich zu PCTE Sicherungspunkte zur Verfügung, womit eine Aktivität durch expliziten Aufruf entsprechender API-Funktionen ([H94]) auf einen bestimmten Sicherungspunkt partiell oder vollständig zurückgesetzt werden kann (siehe [Pla91]). Außerdem wird bei einem expliziten Abbruch einer Transaktion über den Aufruf einer Funktion `Pcte_activity_abort` die gesamte Transaktion zurückgesetzt. Die beim partiellen oder vollständigen Zurücksetzen notwendigen Maßnahmen zur Wiederherstellung der Konsistenz der Objektbank werden über die Recovery-Komponente von H-PCTE ausgeführt. Die dazu notwendigen Maßnahmen werden im einzelnen im nachfolgenden Abschnitt vorgestellt.

2.3.2.5 Recovery in H-PCTE

Im folgenden werden die Recovery-Mechanismen von H-PCTE übersichtsartig erläutert. Die Begriffserklärungen und tiefergehende Informationen zum Recovery im Datenbankbereich findet man in [Wei88]. Details zum Recovery in H-PCTE kann man in [Pla91] nachlesen.

H-PCTE leistet ein logging-basiertes Rückwärts- und Vorwärts-Recovery.Das Vorwärts-Recovery wird von H-PCTE implizit geleistet, d.h. man kann es nicht über spezielle API-Funktionen steuern. Bei Wiederanlauf eines Systems, das nach einem schwerwiegenden Fehler abgestürzt ist, werden dabei auf Basis der bis zum Fehlerzeitpunkt gesammelten Logging-Daten die innerhalb einer Applikation ausgeführten API-Funktionen über *Redo-Funktionen* wiederholt. Ein Vorwärts-Recovery wird in folgenden Fällen geleistet:

1. für Funktionen einer Applikation, die nicht innerhalb einer Transaktion ausgeführt wurden und
2. für die innerhalb einer Transaktion ausgeführten Funktionen einer Applikation, wobei die Transaktion zwar erfolgreich beendet wurde, ihre vorgenommenen Segmentänderungen jedoch zum Fehlerzeitpunkt noch nicht auf einem stabilen Speicher gesichert worden sind.

Das Rückwärts-Recovery wird implizit nach einem Systemfehler für alle noch nicht erfolgreich beendeten Transaktionen geleistet. Darüber hinaus kann man, wie im vorigen Abschnitt beschrieben, über spezielle API-Funktionen explizit ein Rückwärts-Recovery ausführen, indem man eine Transaktion abbricht oder die Transaktion partiell auf einen bestimmten Sicherungspunkt zurücksetzt. Bei Ausführung einer dieser Rückwärts-Recovery-Mechanismen werden automatisch auch die Sperren dementsprechend modifiziert. Wird in einen dieser Fällen ein Rückwärts-Recovery ausgeführt, erkennt H-PCTE auf Basis der Logging-Daten die Funktionen, die innerhalb einer Transaktion ausgeführt wurden, und macht die dabei vorgenommenen Segmentänderungen über *Undo-Funktionen* rückgängig.

Die Logging-Daten werden innerhalb einer Datei, einer Log-Datei, gespeichert. H-PCTE sorgt implizit an einigen Stellen dafür, daß die entsprechenden Daten nicht über das Betriebssystem gepuffert, sondern auf einem nicht-flüchtigen Speicher gesichert werden [Pla91]. Erst nach erfolgreicher Ausführung einer API-Funktion werden jedoch entsprechende Logging-Daten gespeichert. Da das Speichern von Logging-Daten einen Leistungsverlust für H-PCTE bedeutet, kann man über eine API-Funktion `activity_set_logging_mode` beeinflussen, ob auch Logging-Informationen über die Attributwerte unterhalten werden sollen oder nicht. Für jedes Segment wird eine eigene Log-Datei unterhalten, d.h. die Logging-Daten für alle API-Funktionen, die auf einem bestimmten Segment arbeiten, werden auch in einer separaten Log-Datei abgespeichert. Die Log-Datei eines Segments wird immer beim Laden bzw. Kreieren des Segments angelegt und nach dem erfolgreichen Speichern bzw. Löschen des Segments wieder gelöscht.

2.3.2.6 Zugriffskontrollen in H-PCTE

PCTE stellt zwei Arten von Zugriffskontrollen zur Verfügung:

1. *Diskretionäre Zugriffskontrollen (Discretionary Access Controls, DAC):* Die Rechtevergabe für Zugriffe auf ein Schutzobjekt geschieht auf Basis der Identität der Subjekte. Die Zugriffskontrollen sind diskretionär (=im Ermessen liegend) insofern, als nur bestimmte Subjekte, die Besitzer eines Schutzobjekts, darüber entscheiden können, ob und wie andere Subjekte auf dieses Objekt zugreifen dürfen. Beispiel: Benutzer A darf auf Objekt x nicht schreibend zugreifen.
2. *Label-basierte Zugriffskontrollen (Mandatory Access Control, MAC):* Die Informationen können nach Sicherheitsstufen klassifiziert werden, und der potentielle Informationsfluß wird durch Prozesse überwacht. Hierbei haben stets alle Subjekte und Objekte Labels. Die Rechte werden aus Regeln auf Basis dieser Labels ermittelt. Beispiel: Ein Benutzer darf kein Objekt mit Label „geheim“ lesen.

Darüber hinaus sieht PCTE *Auditing*-Funktionen vor, über die sicherheitsrelevante Ereignisse protokolliert werden können. In H-PCTE wird nur diskretionäre Zugriffskontrollen angeboten. Desweiteren werden zwar die Objekt-Standardattribute von PCTE für ein Auditing zur Verfügung gestellt, nicht jedoch die PCTE-Auditingfunktionen.

Bei den diskretionären Zugriffskontrollen in H-PCTE werden die Rechte zur Ausführung der von H-PCTE angebotenen Operationen auf ein *Schutzobjekt*, die ein *Subjekt* besitzt, über eine *Zugriffskontrollmatrix* (Schutzobjekte $\times$ Subjekte) verwaltet. Diese Zugriffskontrollmatrix wird in H-PCTE spaltenweise beim Objekt gespeichert, d.h. jedes Objekt verwaltet seine eigene *Zugriffskontrolliste (ACL, Access Control List).* Solche diskretionäre Zugriffskontrollen nennt man deshalb auch *objekt-orientiert* im Gegensatz zu *subjekt-orientiert*.

Ein *Subjekt* in H-PCTE kann ein Benutzer oder eine Benutzergruppe sein, die intern in der Objektbank immer über ein entsprechendes Objekt vom Objekttyp `user` bzw. `user_group` repräsentiert werden. Da diese beiden Objekttypen vom Objekttyp `security_group` abgeleitet werden, wird sowohl ein Benutzer, als auch eine Benutzergruppe in der Terminologie von PCTE als Sicherheitsgruppe angesehen, wobei eine solche eben auch einen Benutzer in H-PCTE repräsentiert. Ein Objekt vom Typ `user` wird im folgenden *Benutzerobjekt* und ein Objekt vom Typ `user_group` *Benutzergruppenobjekt* genannt. Diese Objekte werden ausgehend von einem speziellen Objekt verwaltet, zu dem man über den Pfad `"_/.security_groups"` navigieren kann. Dieses Objekt wird im folgenden *Gruppenverzeichnisobjekt* genannt. Zur Verwaltung des Benutzernamens stellt H-

PCTE zusätzlich zu PCTE einen Link der Form <name>.named_user zur Verfügung, der immer ausgehend vom Gruppenverzeichnisobjekt auf das entsprechende Benutzerobjekt zeigt. Analog wird ein Link der Form <user_group_name>.named_user_group für Benutzergruppenobjekte angeboten.

H-PCTE bietet gruppenorientierte Zugriffskontrollen an, d.h. es unterstützt die Modellierung von hierarchischen Gruppenbeziehungen. Dies bedeutet im Einzelnen:

- Ein Benutzer kann Mitglied einer Benutzergruppe sein. Dies wird in H-PCTE durch einen Link vom Typ `has_users` angezeigt, der vom entsprechenden Gruppenobjekt auf das Benutzerobjekt zeigt.
- Eine Benutzergruppe A kann *direkte Obergruppe* einer anderen Benutzergruppe B sein. Diese Beziehung wird in H-PCTE über einen Link vom Typ `has_user_subgroups` ausgehend vom Obergruppenobjekt auf das Untergruppenobjekt ausgedrückt. Da die Benutzergruppe A wiederum eine Obergruppe C haben kann, die somit ebenfalls Obergruppe von B ist, nennt man eine solche Benutzergruppe C *indirekte Obergruppe* von B.
- Eine Benutzergruppe C ist *indirekte Obergruppe* einer anderen Benutzergruppe B, wenn C eine direkte Obergruppe einer direkten oder indirekten Obergruppe der Benutzergruppe B ist.
- Alle Benutzer und Benutzergruppen haben mindestens eine (direkte oder indirekte) Obergruppe in Form der Gruppe PCTE_ALL_USERS. Eine Menge von in H-PCTE definierten Benutzergruppen mit ihren Obergruppen/Untergruppen-Beziehungen bilden einen azyklischen Graphen mit der vordefinierten Standardgruppe PCTE_ALL_USERS als Wurzel. Diese Gruppe wird über einen vom Gruppenverzeichnisobjekt ausgehenden Link `"PCTE_ALL_USERS.known_user_group"` zum Standardgruppenobjekt verwaltet.

Zum Beispiel kann man so folgendes Szenario geeignet modellieren: Ein Benutzer `Kappert` arbeitet innerhalb eines Projekts `ProjektProSet` an einem Unterprojekt `ProjektPersistenz`. Das `ProjektPersistenz` ist wiederum ein Unterprojekt im Rahmen des Projekts `Diplomarbeiten`. Ein Projekt kann als Benutzergruppe modelliert werden. Zur geeigneten Modellierung dieses Szenarios kann man dann innerhalb der Objektbank von H-PCTE die in der nachfolgenden Abbildung 2-7 gezeigte Struktur mit Hilfe von API-Funktionen anlegen. Die Abbildung 2-7 enthält aus Gründen der Übersichtlichkeit nicht zusätzlich benötigte diverse Verwaltungslinks von H-PCTE.

Ein solches Subjekt greift dann innerhalb eines Prozesses durch Ausführung einer API-Funktion auf ein Schutzobjekt zu. Ein *Schutzobjekt* kann eine Objekt-Ressource oder eine Link-Ressource von H-PCTE sein. Mit Ressource ist hier eine Instanz innerhalb der Objektbank gemeint, die als zu einem Objekt oder Link gehörend angesehen wird.

Ein Benutzer kann nur über einen H-PCTE-Prozeß auf die Objektbank von H-PCTE zugreifen, wobei der Prozeß immer für mehrere *aktive Subjekte* arbeitet. Ein *aktives Subjekt* eines Prozesses in H-PCTE ist immer ein Benutzer, genauer der über die UNIX-User-ID des Prozesses spezifizierte Benutzer, des Prozesses. Somit setzt H-PCTE auf den UNIX-Sicherheitsmechanismen zur Identifikation und Authentifikation eines Benutzers auf. Zusätzlich zu PCTE wird beim Start eines H-PCTE-Prozesses immer eine Defaultgruppe automatisch aktiviert. Zur Spezifikation der Defaultgruppe eines Benutzers stellt H-PCTE einen Link vom Typ `adopted_user_group` vom Benutzerobjekt auf das entsprechende

Benutzergruppenobjekt zur Verfügung. Darüber hinaus kann man über einen speziellen Funktionsaufruf explizit weitere Benutzergruppen als Subjekte aktivieren. Mit der Aktivierung einer Benutzergruppe werden automatisch auch alle (direkten und indirekten) Obergruppen dieser Gruppe zu aktiven Subjekten.

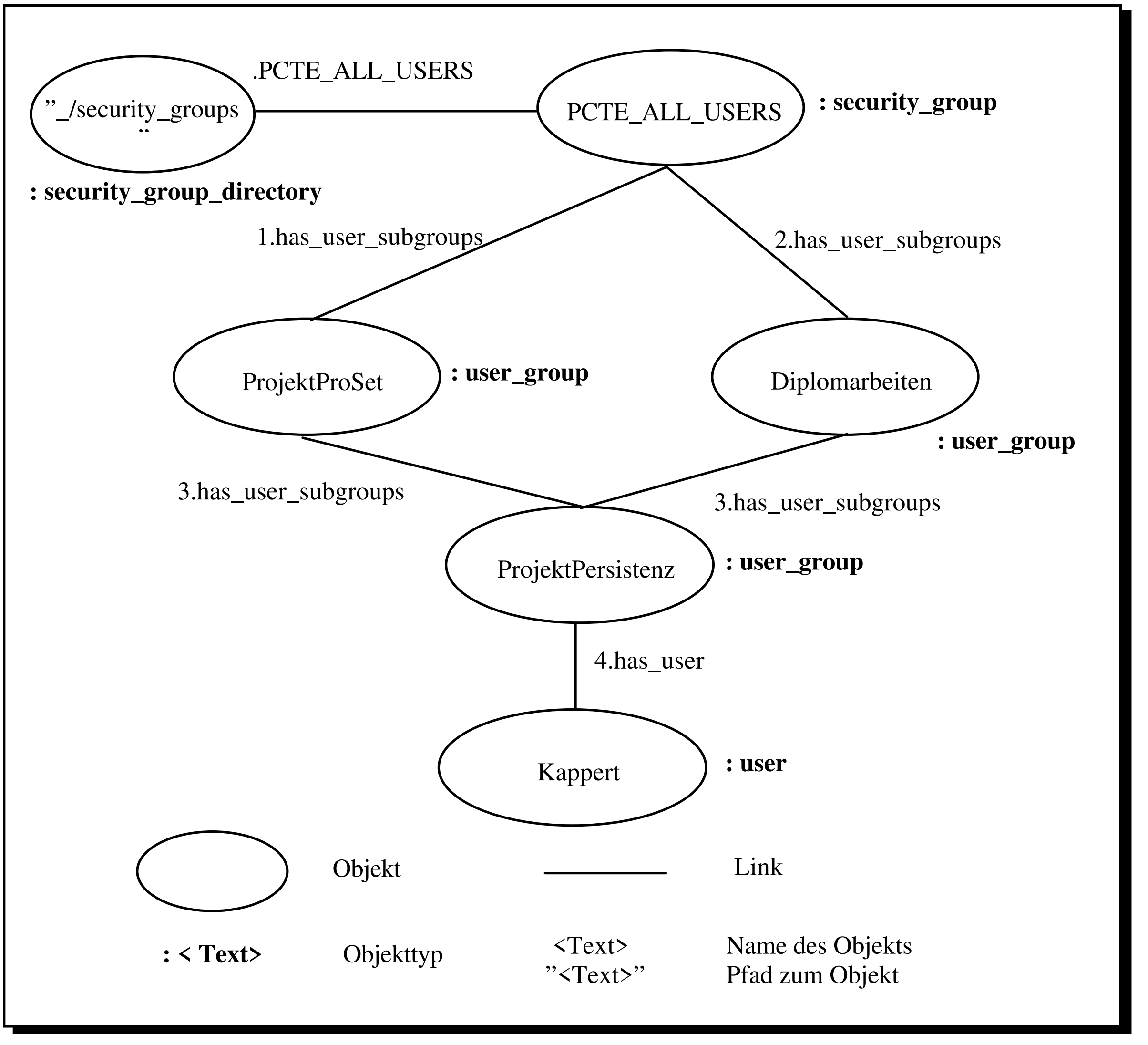


Abbildung 2-7: Instanziierung der Objektbank zur Realisierung der Subjekt-Hierarchien des Beispiel-Szenarios

Da die Anzahl der Operationen, die auf ein Schutzobjekt ausführbar sind, in der Regel so hoch ist, daß es ineffizient wäre, jede einzelne Operation zu kontrollieren, hat man die Operationen in H-PCTE zu sinnvollen Gruppen zusammengefaßt, die *Zugriffsmodi* genannt werden. Ein *Zugriffsmodus* bezeichnet also eine Menge von Operationen, die auf Schutzobjekten ausführbar sind.

Zu jedem Zeitpunkt kann dann auf Basis der Informationen in der Zugriffsmatrix ermittelt werden, ob der Zugriff eines Subjekts auf ein Schutzobjekt mittels einer Operation aus einem Zugriffsmodus erlaubt ist oder nicht. PCTE unterscheidet zwei Modi bei den im Rahmen der Zugriffskontrollen angebotenen Funktionen: Einen Modus für Funktionen auf atomare Objekte und einen für Funktionen auf komplexe Objekte. Da H-PCTE zur Zeit nur im

Rahmen der DAC Funktionen auf atomare Objekte anbietet, wird im Folgenden auch nur dieser Modus erläutert. Informationen zu dieser Thematik liefert [ECM93b], Kapitel 19.

In H-PCTE können mehrere Zugriffsmodi unterschieden werden (siehe Kapitel 19.1.3 in [ECMA2]), wobei jeweils für einen Zugriffsmodus drei Rechtewerte vergeben werden können:

- *erlaubt (+)*, d.h. dieser Zugriff ist für das Subjekt erlaubt
- *explizit verboten (–)*, d.h. dieser Zugriff ist für dieses Subjekt und für alle anderen mit ihm aktiven Subjekte verboten. Dies gilt selbst dann, wenn in einem anderen ACL-Eintrag der Zugriff für ein aktives Subjekt erlaubt wird. Hat ein aktives Subjekt ein Zugriffsrecht auf ein Schutzobjekt also explizit verboten, so kann auch kein anderes mit ihm aktives Subjekt dieses Recht wahrnehmen.
- *undefiniert (?)*, d.h. das Recht ist für dieses aktive Subjekt verwehrt, kann aber durchaus für ein anderes aktives Subjekt gewährt sein.

Sei S die Menge der für einen bestimmten Applikationsprozeß aktiven Subjekte, O die aktuell vorhandene Menge der Schutzobjekte in H-PCTE, M die (statische) Menge der Zugriffsmodi, W die Menge der Rechtewerte, also $W = \{+,-,?\}$ und *access:* $S \times O \times M \rightarrow W$ eine Abbildung die für ein aktives Subjekt $s \in S$, ein Schutzobjekt $o \in O$ und einen Zugriffsmodus $m \in M$ den über eine ACL festgelegten Rechtewert ermittelt.

Der Applikationsprozeß darf nun über eine Operation eines Zugriffsmodus m auf ein Schutzobjekt $o \in O$ zugreifen, wenn gilt

1. es existiert mindestens ein aktives Subjekt $s \in S$, so daß gilt $access(s,o,m) = +$ und
2. es existiert kein aktives Subjekt $s \in S$, so daß gilt $access(s,o,m) = -$.

Zur Veranschaulichung der diskretionären Zugriffskontrollen von H-PCTE sei obiges Beispiel weitergeführt. Der Benutzer `Kappert` startet über einen Aufruf der entsprechenden H-PCTE-API-Funktion einen H-PCTE-Prozeß. Somit ist Benutzer `Kappert` aus Sicht der DAC von H-PCTE das aktive Subjekt. Die Benutzergruppe `ProjektPersistenz` sei als Defaultgruppe dieses Benutzers spezifiziert worden. Dann wird beim Start des Prozesses zusätzlich die Benutzergruppe `ProjektPersistenz` ein aktives Subjekt und somit werden auch die Obergruppen `ProjektProSet`, `Diplomarbeiten` und `PCTE_ALL_USERS` zu aktiven Subjekten.

Subjekt	**Attribut lesen**	**Attribut schreiben**	**Objekt löschen**	**...**
`Kappert`	?	+	+	
`ProjektPersistenz`	+	?	?	
`ProjektProSet`	+	+	?	
`Diplomarbeiten`	?	–	?	

Tabelle 2-2: Beispiel-ACL des Objekts O

Greift nun Benutzer `Kappert` auf ein H-PCTE-Objekt O zu, wobei die ACL die in Tabelle 2-2 beschriebenen Einträge aufweist, so

- darf er nicht die Attribute von O lesen, da sein ACL-Eintrag ihm dieses Recht verwehrt;

- darf er nicht die Attribute von O verändern, da zwar sein ACL-Eintrag ihm dieses Recht gewährt, jedoch ein anderes für ihn aktives Subjekt D diesen Zugriff explizit verboten hat;
- darf er das Objekt löschen, da sein ACL-Eintrag ihm dieses Recht gewährt und kein anderes Subjekt dieses Recht explizit verboten hat.

Hier ist noch anzumerken, daß bei einem Zugriff auf eine zu einem Objekt gehörende Ressource anhand der für die aktiven Subjekte vorhandenen ACL-Einträge des Objekts geprüft wird, ob der Zugriff für den Benutzer gewährt ist.

2.3.2.7 Entwicklungsstand von H-PCTE

Diese Diplomarbeit basiert auf der Version 2.6 von H-PCTE, die im Februar 1995 erschienen ist. In dieser Version werden noch nicht alle Operationen auf komplexe Objekte unterstützt, insbesondere die Kopierfunktionen sind noch nicht für komplexe Objekte erweitert worden. Desweiteren werden keine Funktionen für eine Versionierung von Objekten angeboten. H-PCTE bietet diskretionäre Zugriffskontrollen an, aber weder MAC noch Auditingfunktionen. Es werden weiterhin nur Operationen auf ACLs von atomaren Objekten unterstützt. Das Verteilungskonzept von H-PCTE ist immer noch „halb-verteilt", eine vollverteilte Version wird in Ausblick gestellt. Es werden allerdings keine Replikationsfunktionen angeboten.

H-PCTE wird in zwei Varianten ausgeliefert: Eine läuft unter dem UNIX-Betriebssystem SunOS 4.1 (Solaris 1), die zweite unter der neuen SunOS-Version Solaris 2. Die API von H-PCTE ist in ANSI-C geschrieben und wird entsprechend der beiden UNIX-Versionen über Bibliotheken angeboten.

3 Anforderungsanalyse

Um nun PROSET-Werte mit Hilfe von H-PCTE persistent machen zu können, bedarf es der Integration der Persistenz in die Programmiersprache PROSET. Hierzu existiert eine vielfältige Auswahl an Konzepten, die im folgenden auf ihre Eignung für eine Integration in eine Prototyping-Sprache geprüft werden. Darüber hinaus werden diverse allgemeine Anforderungen an die persistente Datenhaltung in einem Repository, die aus den Bereich der Datenintegration in SEUs stammen, untersucht und innerhalb der Persistenzkonzepte berücksichtigt.

3.1 Anforderungen aufgrund der Konzeption von PROSET

Zunächst werden die bereits in Kapitel 2.2.2 genannten grundsätzlichen Anforderungen an die Persistenzkonzepte einer Prototyping-Sprache, wie PROSET, überblicksartig zusammenfaßt, da sie einen nicht unerheblichen Einfluß auf die Persistenzkonzepte besitzen:

- **(P1):** Die Persistenzkonzepte sollten dem Programmierer, gemäß dem Selbstverständnis von PROSET als Prototyping-Sprache, auf einem möglichst hohem Abstraktionsniveau zur Verfügung gestellt werden. Dies bedeutet insbesondere, daß der Programmierer sich nicht selbst um eine Überwindung des Impedance Mismatch kümmern muß.
- **(P2):** Gleichzeitig sollte jedoch dort, wo es sinnvoll erscheint, der algorithmische Transformationsansatz von PROSET unterstützt werden. Dies kann z.B. über spezielle Befehlsbibliotheken geschehen, die in Form eines PROSET-Moduls zur Verfügung gestellt werden können und Operationen zur feineren und flexibleren Bearbeitung von persistenten PROSET-Werten anbieten.
- **(P3):** Die Persistenzkonzepte sollten orthogonal zu anderen Sprachkonzepten verwendbar sein, um den Gebrauch und die Erlernbarkeit der Sprache zu erleichtern.. Aus genau demselben Grund sollten die integrierten Persistenzkonzepte möglichst leicht verständlich und anwendbar sein.
- **(P4):** Ein Nichtbenutzen der Persistenz von PROSET soll den Programmierer nichts bzw. möglichst wenig kosten. D.h., insbesondere bzgl. der benötigten PROSET-Laufzeitbibliotheken (siehe Kapitel 2.1.2) sollte kein Overhead entstehen, um sowohl den Ressourcenbedarf, als auch die Übersetzungszeiten eines PROSET-Programms zu minimieren.
- **(P5):** Beim (explorativen) Prototyping ist es in einem besonderen Maße wichtig, daß die Prototyping-Sprache in möglichst breiten Anwendungsbereichen einsetzbar ist. Aus diesem Grunde sollte man dort, wo es nicht ein Konflikt mit anderen Anforderungen gibt, in den oben erwähnten Befehlsbibliotheken auch Operationen aufnehmen, die allein den Anwendungsbereich von PROSET erweitern. Dies heißt jedoch nicht, daß hohe Effizienzansprüche an die Operationen gestellt werden, da zur Konstruktion des Produktionsprogramms, in welchem dann die Effizienz von größerer Bedeutung ist, meistens eine spezielle auf den Anwendungsbereich abgestimmte Programmiersprache benutzt wird.

Wie schon in Kapitel 2.2.3 beschrieben, stellt PROSET bereits Sprachkonstrukte zur Unterstützung der Persistenz zur Verfügung, die dem Programmierer auf einem hohen Abstraktionsniveau ein Arbeiten mit persistenten PROSET-Werten ermöglicht.

Im folgenden werden weitere benötigte Persistenzkonzepte in Programmiersprachen analysiert. Einige dieser Konzepte sollten auch auf Sprachebene über zusätzliche ProSet-Konstrukte oder über ProSet-Module angeboten werden. Wogegen andere lediglich intern, d.h. innerhalb der Laufzeitbibliothek von ProSet, dazu benötigt werden, um die Persistenz geeignet realisieren zu können.

3.2 Persistenzkonzepte in Programmiersprachen

Es wird nun basierend auf [KLLM94] ein konzeptionelles Grundgerüst für Persistenzkonzepte in Programmiersprachen präsentiert, das als Grundlage für eine detaillierte Anforderungsanalyse bzgl. der Integration von Persistenzkonzepten in ProSet dienen soll. Parallel dazu werden die Konzepte, die bereits in ProSet Eingang fanden, vorgestellt und erweitert.

Die ursprünglich in [KLLM94] vorgestellten Konzepte beziehen sich auf objektorientierte Programmiersprachen. Diese Konzepte wurden hier auf imperative Prototyping-Sprachen übertragen und das Grundgerüst an sich um einige konzeptionelle Betrachtungen erweitert.

3.2.1 Spezifikation persistenter Datenobjekte

Bei der Einführung der Persistenz in eine Programmiersprache hat man prinzipiell bei der Wahl einer geeigneten Spezifikationsmethode für die persistenten Datenobjekte zwei Kriterien zu beachten [AFH94]:

1. Den *Zeitpunkt der Spezifikation:* Wann wird das Datenobjekt persistent gemacht?
2. Die *Selektion der Daten:* Welche Datenobjekte sollen persistent gemacht werden können?

3.2.1.1 Zeitpunkt der Spezifikation

Man kann hier generell folgende Möglichkeiten unterscheiden:

1. Die Überführung des Datenobjektzustands in einen persistenten Zustand bei der Erzeugung des (transienten) Datenobjekts, also z.B. bei seiner Deklaration.
2. Die Überführung in einen persistenten Zustand zu irgendeinem späteren Zeitpunkt, wie z.B. zum Ende einer Programmeinheit bzw. des Programms oder beim ersten Zugriff auf das Datenobjekt.

In ProSet wird der ProSet-Wert bei seiner Deklaration persistent gemacht. Zu Beginn der Ausführung einer Programmeinheit wird versucht, alle in der entsprechenden Programmeinheit deklarierten persistenten ProSet-Werte von H-PCTE in den Adreßraum des Programms zu laden. Existiert ein per `persistent` deklarierter persistenter ProSet-Wert in einem P-File noch nicht, so wird eine Ausnahme `p_missing_name` ausgelöst, die der Programmierer über einen selbstdefinierten Handler abfangen kann. Da diese Ausnahme zur Klasse der `notify`-Ausnahmen gehört, wird nach Beendigung des Handlers per `resume <proset_value>` mit dem Programm fortgefahren, indem der angesprochene ProSet-Wert mit dem Rückgabewert des Handlers initialisiert wird. Nach Beendigung seiner Programmeinheit wird dieser Wert dann im persistenten Speicher neu angelegt. Wurde der nicht-existierende persistente ProSet-Wert hingegen per `persistent constant` als Konstante deklariert, wird eine Ausnahme `p_missing_constant` generiert. Diese Ausnahme zählt zu der Klasse der `escape`-Ausnahmen, die nur mit `return` beendet werden können. Weitere Details bzgl. der Ausnahmebehandlung werden im Zusammenhang mit dem PROSET-Transaktionsmodell erläutert.

Existiert bereits das vom Programm angesprochene P-File nicht, wird eine `escape`-Ausnahme `p_missing_p_file` erzeugt, wobei hier aber nicht nach Beendigung des Handlers das P-File explizit angelegt, sondern das Programm abgebrochen wird. Das Anlegen und Löschen von P-Files bzw. Sub-P-Files sollte nur über spezielle Werkzeuge vollzogen werden. Darüber hinaus sollte zur Kontrolle des Systemressourcenverbrauchs es nur dem Administrator der Prototyping-Umgebung bzw. einem H-PCTE-Datenbank-Administrator erlaubt sein, P-Files für einen Benutzer anzulegen, in die der Benutzer dann seine persistenten PROSET-Werte ablegen kann.

3.2.1.2 Selektionskriterien für persistente Datenobjekte

Die Klassifikation eines Datenobjekts als persistent oder transient kann typunabhängig oder typabhängig sein.

- Falls nur eine Teilmenge der von der Sprache angebotenen Typen von Datenobjekten persistent gemacht werden können, nennt man die Unterstützung der Persistenz *typspezifisch*.
- Falls ein persistentes Datenobjekt jeglichen von der Sprache angebotenen Datentyp besitzen kann, nennt man die Persistenzunterstützung *typorthogonal*.

In PROSET ist die Persistenz bisher typorthogonal *für alle Datentypen mit Bürgerrechten erster Klasse*, d.h. alle Instanzen der von PROSET angebotenen Datentypen mit Bürgerrechten erster Klasse können persistent gemacht werden. Außerdem kann noch der undefinierte PROSET-Wert `Om`, der keinen Datentyp besitzt, persistent gemacht werden. Hingegen können Programmeinheiten, wie `procedure`, `lambda`, `handler` und `module`, nicht direkt persistent gemacht werden. Mit direkt ist hier gemeint, daß man sie nicht als persistent deklarieren kann. Indirekt können sie persistent gemacht werden, wie man später noch sehen wird.

3.2.2 Zugriff auf persistente Datenobjekte

Um auf ein persistentes Datenobjekt zugreifen zu können, wird im allgemeinen der an dieses Datenobjekt *gebundene Name* benutzt. Hierbei müssen aber Name und Identität des Datenobjekts unterschieden werden, da der gleiche Name an verschiedene Datenobjekte gebunden sein kann. Somit lautet nach [ABM88] eine Forderung, daß die Identität des Datenobjekts eine gekapselte Eigenschaft des Datenobjekts sein muß, anhand der man es von anderen Datenobjekten unterscheiden kann.

Ein Name kann entweder

- ein *Bezeichner* (identifier name) sein, der explizit Teil des Programmtextes ist, oder
- aus einem *Ausdruck* (expression name) bestehen, aus dem erst zur Laufzeit durch seine Auswertung ein Name abgeleitet werden kann.

Eine dritte Zugriffsart auf persistente Datenobjekte bildet der *assoziative Zugriff*: Hiermit sollen Zugriffe auf Datenobjekte über „Datenbankabfragen“ bezeichnet werden. Man könnte so z.B. mit Hilfe von Wildcards über Namensmuster Datenobjekte selektieren. Solche Zugriffsfunktionen werden jedoch innerhalb dieser Diplomarbeit nicht berücksichtigt. Es wäre allerdings wünschenswert diese im Rahmen einer Erweiterung der Diplomarbeit um eine Anfragesprache auf persistente PROSET-Werte und (Sub-)P-Files anzubieten.

In PROSET kann man bisher auf die persistenten PROSET-Werte mit Hilfe ihres Bezeichners zugreifen, wogegen ein P-File über einen PROSET-Wert vom Typ `string` angesprochen wird. Möchte der Programmierer ein persistenten PROSET-Wert benutzen, so muß er dies im Deklarationsteil der entsprechenden Programmeinheit mit Hilfe des Schlüsselwortes `persistent` angekündigen. Hierbei identifiziert ein Bezeichner den PROSET-Wert, auf den man zugreifen möchte und ein (Sub-)P-File-Name den (Sub-)P-File, aus dem der PROSET-Wert entnommen werden soll. Ein P-File bzw. Sub-P-File bildet somit einen Namensraum für die in ihm enthaltenen persistenten PROSET-Werte. Ein P-File kann jedoch einen Sub-P-File und einen persistenten PROSET-Wert enthalten, die den gleichen Namen besitzen. Der Zugriff auf einen persistenten PROSET-Wert in einem PROSET-Programm wird anhand von Beispiel 3-1 illustriert.

```
program demo;
          persistent  x :  ”MyDatabase.Project1”;
  begin
     x := 7;
  end demo;
```

Beispiel 3-1: Zugriff auf einen persistenten PROSET-Wert eines Sub-P-Files

Zur genaueren Bezeichnung und späteren Unterscheidung der P-File-Namen auf PROSET-Ebene und der Namen der H-PCTE-Objekte, die einen P-File repräsentieren, werden die in der EBNF-Grammatik von Abbildung 3-1 definierten *P-File-Pfadname*, *P-File-Name* und *Sub-P-File-Name* verwendet.

```
<persistente Deklaration> ::=
          ['visible' | 'hidden'] 'persistent' ['constant']
          <Bezeichner> ':' <P-File-Pfadname>';'.
<P-File-Pfadname>   ::= <P-File-Name>['.'<Sub-P-File-Name>]*.
<P-File-Name>       ::= <ProSet-Bezeichner>.
<Sub-P-File-Name>   ::= <ProSet-Bezeichner>.
<ProSet-Bezeichner> ::= <Buchstabe>[<Zahl>|_|<Buchstabe>]*.
<Buchstabe>         ::= A | .. | Z | a | .. | z.
<Zahl>              ::= 0 | .. | 9.
```

Abbildung 3-1: P-File-Pfadname, P-File-Name und Sub-P-File-Name

Die Forderung nach der Identität eines Datenobjekts wird im folgenden Abschnitt noch einmal gesondert diskutiert, da die Identität von besonderer Bedeutung ist und nicht bei jedem persistenten PROSET-Wert speziell gesichert wird.

3.2.3 Identität der persistenten Datenobjekte

In ProSet wird prinzipiell keine spezielle Identitätskennung an den ProSet-Werten benötigt, da die Zuweisungen in ProSet eine *Wertsemantik* besitzen.

Wertsemantik von Zuweisungen bedeutet, daß bei ProSet-Werten A, B und einer Zuweisung A:=B nur die Werte von A und B gleich sind, nicht aber ihre Identität (im Gegensatz zur Referenzsemantik). Um jedoch die Werte von ProSet-Werten höherer Ordnung vergleichen zu können, besitzen diese explizit eine Identität. Dabei wird jedem ProSet-Wert höherer Ordnung zur Laufzeit (bei seiner Erzeugung) ein Atom zugeordnet. Da ein Atom einen weltweit eindeutigen ProSet-Wert darstellt, ist die Identität eines ProSet-Werts höherer Ordnung zu jedem Zeitpunkt gesichert.

Die Identität bei einem persistenten primitiven oder zusammengesetzten ProSet-Wert ist nur über ihren Bezeichner gesichert, jedoch war eine zusätzliche Identitätskennung eben wegen der Wertsemantik auch nicht notwendig.

Eine zusätzliche Anforderung an die Identität der persistenten ProSet-Werte entsteht jedoch bei Einführung der Persistenz in die Sprache. Hier wäre aus Datenmodellierungsgründen wünschenswert, daß die Identität der geladenen persistenten ProSet-Werte zu jedem Zeitpunkt mit der Identität der sie repräsentierenden Objekte in der Objektbank übereinstimmt.

3.2.4 Operationen auf persistenten Datenobjekten

ProSet bietet, wie bereits erwähnt, Operationen auf einem so hohen Abstraktionsniveau an, daß der Programmierer sich nicht selbst um das Laden und Speichern von persistenten ProSet-Werten kümmern muß. ProSet übernimmt diesen Datentransport für jeden als persistent deklarierten ProSet-Wert implizit.

Zur Unterstützung der Anforderungen (P2), Erweiterung des Anwendungsbereichs der Sprache, und (P5), Unterstützung des transformationellen Ansatzes von ProSet, sind jedoch ebenfalls spezielle Operationen für die persistenten ProSet-Werte und P-Files bzw. Sub-P-Files wünschenswert. Um jedoch nicht mit den oben genannten Anforderungen (P1), (P3) und (P4) in Konflikt zu geraten, sollten die Operationen über ein ProSet-Modul innerhalb der Sprache angeboten werden. Dies bedeutet, daß in ANSI-C geschriebene Funktionen, die diese Operationen realisieren, in einem ProSet-Modul gekapselt werden und so über die Importschnittstelle eines anderen Moduls benutzt werden können.

Wünschenswerte Operationen sind hierbei:

- Der Programmierer kann mit Hilfe der für den entsprechenden Datentyp des ProSet-Werts zur Verfügung gestellten Operationen den Wert eines persistenten ProSet-Werts zwar ändern, jedoch werden die Änderungen erst bei Beendigung der Programmeinheit persistent gemacht. Um in einigen Anwendungsgebieten flexibler agieren zu können, wären Operationen für ein punktuelles Speichern und auch Laden von persistenten ProSet-Werten wünschenswert.
- Außerdem sollten spezielle Operationen zur Verfügung gestellt werden, die es dem Programmierer ermöglichen, auf die auf einem hohen Abstraktionsniveau angebotenen Persistenzkonzepte direkter (auf einem niedrigen Abstraktionsniveau) Einfluß zu nehmen. Dann wäre es sogar möglich, in ProSet geeignete Werkzeuge zur Unterstützung der Persistenzkonzepte von ProSet zu schreiben. (Bsp.: Verwaltungswerkzeuge für die

persistenten ProSet-Werte und P-Files, Werkzeuge zur Unterstützung eines Schutzkonzeptes, etc.).

Welche genauen Operationen über ProSet-Module zur Verfügung gestellt werden sollen, wird später immer im Zusammenhang mit den jeweiligen Persistenzkonzepten vorgestellt.

3.2.5 Mobilität der persistenten Datenobjekte

Zu jedem Zeitpunkt hat ein Datenobjekt einen festen physikalischen Ort, an dem es gespeichert ist. Ein persistentes Datenobjekt sollte sich zwischen verschiedenen Primärspeichern und insbesondere Sekundärspeichern bewegen können, um z.B. den Anforderungen an ein verteiltes System, wie z.B. H-PCTE, zu genügen. Hierbei soll zwischen einer horizontalen und vertikalen Mobilität unterschieden werden, wie in Abbildung 3-2 veranschaulicht wird.

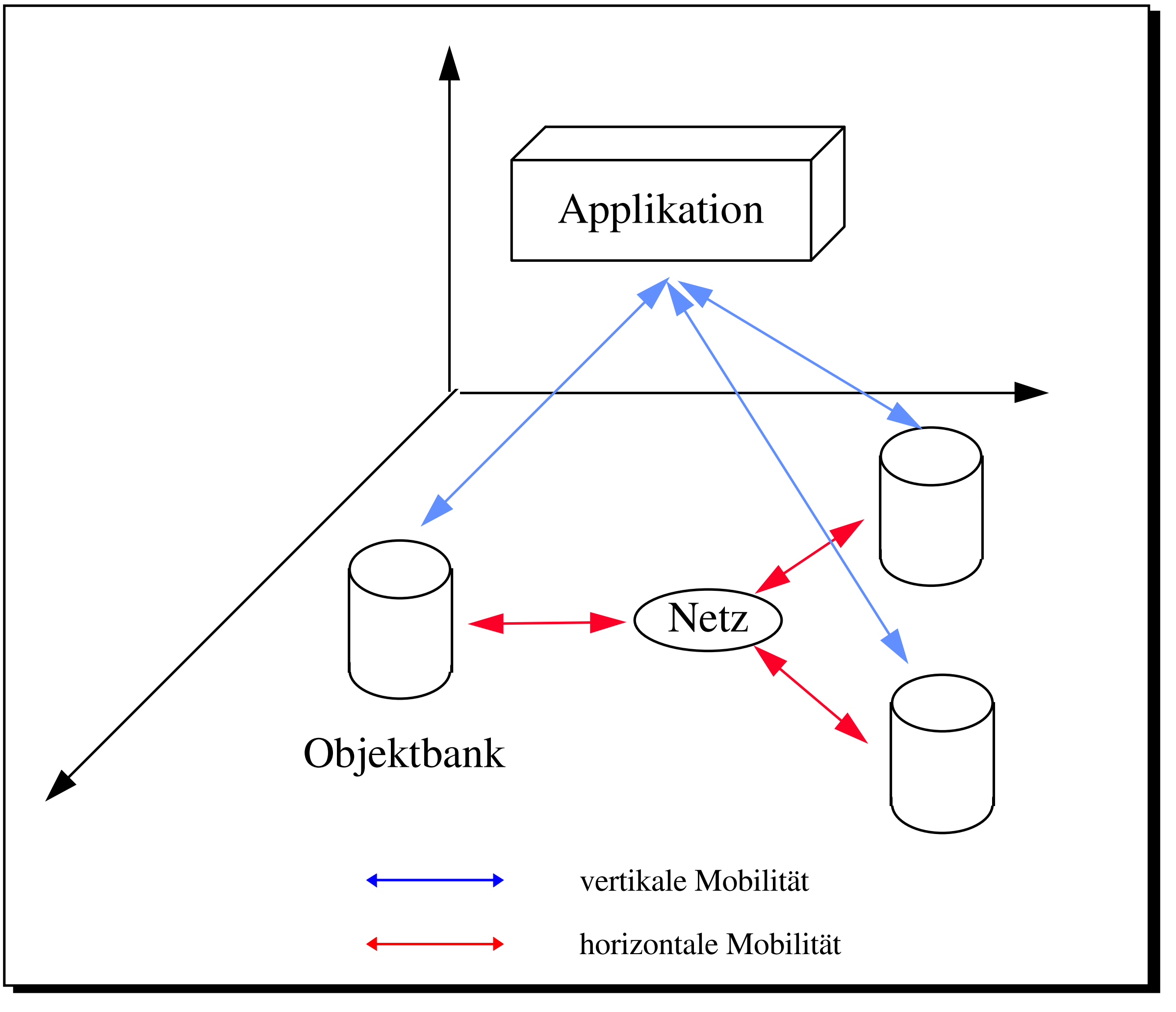


Abbildung 3-2: Horizontale und vertikale Mobilität der Objekte

- Die *horizontale Mobilität* soll die Mobilität der persistenten Datenobjekte zwischen den Speichern von verschiedenen Rechnerknoten in einem verteilten System bezeichnen, wogegen
- die *vertikale Mobilität* die Mobilität der persistenten Datenobjekte zwischen dem virtuellen Speicher des Applikationsprozesses und dem Speicher der Objektbank bezeichnen soll.

Innerhalb von Programmiersprachen kann man grundsätzlich über Operationen auf die Mobilität der persistenten Datenobjekte Einfluß nehmen. Hierbei unterscheidet man zwei Konzepte:

1. Falls eine Programmiersprache dem Programmierer keinen Mechanismus anbietet, mit dem er den Speicherungsort eines persistenten Datenobjekts feststellen kann, wird die Sprache *ortstransparent* oder *ortsunabhängig* genannt.
2. Falls der Mechanismus zum Zugriff auf ein persistentes Datenobjekt einer Sprache unabhängig von dem Ort des persistenten Objekts ist, unterstützt die Sprache einen *ortsunabhängigen Zugriff*.

ProSet unterstützt einen ortsunabhängigen Zugriff. Einen ortsabhängigen Zugriff sollte man auch nur über Werkzeuge gestatten, da solche Zugriffe in erster Linie mit der Intention getätigt werden, eine Performanzverbesserung beim Zugriff auf die persistenten Datenobjekte zu erzielen. Denn Optimierungen des Speichers, wie z.B. durch *Clustering* von bestimmten Datenobjekten oder eine verteilte Speicherung, gehören in die Aufgabengebiete eines Systemadministrators.

Somit unterstützt die horizontale Mobilität eines persistenten Datenobjekts die Performanz bei Zugriffen auf die Objektbank und insbesondere die Verfügbarkeit der Datenobjekte (bei einer verteilten Speicherung), wogegen eine vertikale Mobilität, zu der man speziell ein *Caching* zählen kann, allein die Zugriffsperformanz unterstützt – diese allerdings in einem besonderen Maße.

Im nächsten Abschnitt werden die im engen Zusammenhang zur vertikalen und horizontalen Mobilität stehenden Strategien zum Caching und Clustering von Datenobjekten vorgestellt und bewertet.

3.2.6 Caching und Clustering von persistenten Datenobjekten

Unter Caching und Clustering von persistenten Datenobjekten versteht man folgendes:

- *Caching* bezeichnet in diesem Zusammenhang den Vorgang, ein persistentes Datenobjekt zusammen mit anderen persistenten Datenobjekten vom langsamen stabilen Hintergrundspeicher in einen schnellen Primärspeicher zu laden (i.d.R. Hauptspeicher).
- *Clustering* bezeichnet hier den Vorgang, ein persistentes Datenobjekt zusammen mit anderen persistenten Datenobjekten in einem zusammenhängenden Speicherbereich auf einen (im Vergleich zum Primärspeicher langsamen) stabilen Hintergrundspeicher abzulegen.

Grundvoraussetzung für ein effektives Caching ist jedoch, daß auf einen relativ (zur Größe des Cache-Speichers) kleinen Teil der Datenobjekte relativ oft zugegriffen wird. Diese Forderung wird *Lokalität der Zugriffe* genannt.

3.2.6.1 Caching

Da ProSet die Persistenz mit Hilfe der Haupspeicherdatenbank H-PCTE realisiert, kann man die vertikale Mobilität zweistufig definieren. Dies sei am Beispiel des Ladens eines persistenten ProSet-Werts demonstriert:

1. In einer ersten Stufe wird unter der Verantwortung von H-PCTE das Segment eines vom Applikationsprozeß benötigten Datenobjekts global in den Adreßraum des H-PCTE-Servers bzw. lokal in den Adreßraum des Applikationsprozesses geladen.

2. Daraufhin wird in einer zweiten Stufe das Datenobjekt aus dem Adreßraum des H-PCTE-Servers in den Adreßraum des PROSET-Applikationsprozesses transferiert bzw. das Datenobjekt innerhalb des virtuellen Adreßraums des PROSET-Applikationsprozesses umkopiert. Hier sei zwischen Transferieren und Umkopieren unterschieden. Transferieren benötigt im Gegensatz zum Umkopieren die Interprozeßkommunikation.

Da H-PCTE, wie in Kapitel 2.3.2.3 auf Seite 15 dargelegt, eine Workstation-Server-Prozeßarchitektur aufweist, kann man innerhalb der ersten Stufe der vertikalen Mobilität Einfluß auf die Strategie zum lokalen und globalen Laden der Segmente in H-PCTE nehmen. Hier sollte dem Benutzer eines auf persistente PROSET-Werte arbeitenden PROSET-Programms eine Möglichkeit eingeräumt werden, die Segmente lokal in seinem Applikationsprozeß zu laden, um so eine für H-PCTE optimale Zugriffsperformanz zu erzielen. Ein Konzept für eine solche Strategie wird in Kapitel 4.1 ab Seite 50 vorgestellt.

Unter dem Aspekt der vertikalen Mobilität der zweiten Stufe wird nun ein Caching von persistenten PROSET-Werten im Adreßraum des PROSET-Applikationsprozesses in der Form untersucht, daß bei einem Zugriff auf diesen Datenobjekten ein Umkopieren oder Transferieren entfällt. Mit Hilfe dieser Art von Caching soll also verhindert werden, daß ein geladener persistenter PROSET-Wert aus dem Programm enfernt wird, obwohl er zu einem späteren Zeitpunkt der Programmausführung nocheinmal benötigt wird. Eine solche Optimierung bewirkt allerdings nur dann einen spürbare Steigerung der Zugriffsperformanz, wenn das Segment des persistenten PROSET-Werts global und nicht lokal geladen wurde und somit ein Transferieren mit einem Interprozeßkommunikationsaufwand entfällt.

Eine solche Problemsituation tritt bei Iterationen und Rekursionen von Programmeinheiten auf, die mit persistenten PROSET-Werten arbeiten, wie Beispiel 3-2 illustriert.

```
program demo;
begin
  S:={1..100};
  for x in S do iterate(); end for;
  procedure iterate;
     persistent x : ”MyDatabase”;
  begin
     x:= x + 1;
  end iterate;
end demo;
```

Beispiel 3-2: Iteration von Programmeinheiten, die auf persistenten PROSET-Werten arbeiten

Denn bei strenger Verfolgung der bisherigen, in Kapitel 3.2.1.1 auf Seite 25 vorgestellten Load-Store-Strategie für persistente PROSET-Werte, würde man bei jeder Iteration bzw. Rekursion den benötigten persistenten PROSET-Wert neu in den Applikationsprozeß laden. Jedoch sollten die persistenten PROSET-Werte einer Programmeinheit immer zum Ende eines Iterationsdurchlaufs zwischengespeichert werden, um mit einem Transaktionskonzept nicht in Konflikt zu geraten. Diese Vorgehensweise kann man mit der *Write-Through* Caching-Strategie vergleichen. So kann man jederzeit, wenn ein Fehler in einem i-ten Durchlauf auftritt, auf den Datenbankzustand nach dem (i-1)-ten Durchlauf zurücksetzen.

Diese Optimierung müßte jedoch auf Compilerebene durchgeführt werden. Die Realisierung einer solchen Caching-Strategie ist allerdings sehr schwierig: Aus dem ProSet-Programm wird mit Hilfe einer in ANSI-C geschriebenen Laufzeitbibliothek ein C-Programm erzeugt und somit werden auch die (geschachtelten) ProSet-Programmeinheiten auf flache C-Funktionen abgebildet. Da die lokalen Werte einer Programmeinheit auch nur lokal in einer C-Funktion erzeugt werden, müßte also sowohl die Abbildungsvorschrift von ProSet nach C als auch die Laufzeitbibliothek selbst erheblich erweitert werden (insbesondere lokale persistente Werte müßten global gehalten werden). Eine solche Optimierung wäre zwar wünschenswert, würde jedoch den Rahmen der Diplomarbeit sprengen und wird deshalb auch nicht realisiert.

3.2.6.2 Clustering

H-PCTE bietet über die Speicherung von Objekten auf Segmenten eine Möglichkeit für ein Clustering von Datenobjekten an. Mit Hilfe der Segmente kann man auf der einen Seite eine verteilte Speicherung von persistenten ProSet-Werten und auf der anderen Seite ein Caching bzgl. der Mobilität der ersten Stufe, d.h. das lokale Laden von Segmenten im Applikationsprozeß, unterstützen. Ein solches Clustering hat insbesondere Einfluß auf das lokale Laden von Segmenten. Wie in Abschnitt 2.3.2.3 auf Seite 15 bereits erwähnt, kann jeweils nur der Applikationsprozeß auf die persistenten ProSet-Werte eines Segments zugreifen, der dieses Segment lokal geladen hat. Benutzt dieser Prozeß allerdings nur eine kleine Teilmenge der auf diesem Segment gespeicherten ProSet-Werte, so erhöht dies die Wahrscheinlichkeit, daß ein anderer Prozeß ebenfalls auf diese Daten zugreifen möchte. Daher sollten möglichst nur die von einem Programm benötigten persistenten ProSet-Werte auf ein Segment abgelegt werden. Da in ProSet ein P-File ebenfalls (wie das Segment in H-PCTE) als „Container" für persistente ProSet-Werte dient, wird zur Unterstützung der Anforderung (P3) jedem P-File und Sub-P-File ein Segment zugeordnet, auf dem alle (Sub-) P-File-Einträge gespeichert werden.

Zur Unterstützung der Anforderung (P1) wird das Clustering für den Programmierer transparent vorgenommen, d.h. nach Deklaration eines noch nicht existierenden persistenten ProSet-Werts wird dieser automatisch auf das Segment seines P-Files abgelegt. Wie bereits erwähnt, kann man die P-Files und Sub-P-Files nur über spezielle Werkzeuge anlegen, wobei als zusätzliche Anforderung die P-Files und Sub-P-Files implizit auf jeweils ein eigenes Segment abgelegt werden.

3.2.7 Mehrbenutzerbetrieb

Wenn eine Programmiersprache dem Programmierer anbietet, Datenobjekte persistent zu halten, so sollte die Sprache prinzipiell auch die gemeinsame Nutzung (Data Sharing) dieser Datenobjekte durch mehrere Benutzer ermöglichen [AFH94]. Darüber hinaus werden größere Software-Systeme arbeitsteilig in Teams entwickelt. Zur Unterstützung einer parallelen Arbeit der Entwickler und der Kooperation innerhalb des Teams sollte ein gleichzeitiger Zugriff auf die persistenten Datenobjekte von mehreren Benutzern bzw. Prozessen aus ermöglicht werden ([ES89], Kap. 3.2.2). Denn nur so kann die Wiederverwendbarkeit von persistenten Datenobjekten in einem Software-Entwicklungsprozeß mit ProSet geeignet unterstützt werden.

Die Forderung nach *Parallelität* entspringt insbesondere den Zielsetzungen bei Datenbankanwendungen (auf einem höheren Abstraktionsniveau sind dies Programme, die die Persistenz einer Sprache nutzen) einen möglichst hohen Durchsatz zu gewährleisten. So sollen auf der einen Seite die Antwortzeiten für den bzw. die Benutzer möglichst kurz sein und auf der anderen Seite möglichst viele Datenbankoperationen pro Zeiteinheit abgearbeitet werden.

Ein solcher Mehrbenutzerbetrieb erfordert allerdings eine Unterstützung durch eine Reihe von Mechanismen, die aus dem Bereich von Datenbanksystemen stammen. Diese Mechanismen werden sowohl innerhalb einer Programmiersprache, die Persistenz anbietet, als auch bei einem Mehrbenutzerbetrieb auf ein Repository der Prototyping-Umgebung benötigt. Da die Analyse von Anforderungen an eine Prototyping-Umgebung noch ein nicht abgeschlossener Forschungsschwerpunkt der ProSet-Forschungsgruppe ist und innerhalb dieser Diplomarbeit nur eine Grundlage geschaffen werden soll, um die Arbeit auf persistenten ProSet-Werten von ProSet-Programmen und von Werkzeugen aus zu unterstützen, können die Anforderungen an die persistente Datenhaltung einer Prototyping-Umgebung nur ansatzweise berücksichtigt werden.

Ein Mehrbenutzerbetrieb erfordert (nach [Vos94], Kap. 18, übertragen auf ein Repository für persistente ProSet-Werte) insbesondere einen Schutz vor Parallelitätsanomalien ([Vos94], Kap. 18.2). Hiermit sind Inkonsistenzen gemeint, die durch einen zeitgleichen Zugriff verschiedener Benutzer bzw. Applikationsprozesse auf denselben persistenten Datenobjekten entstehen können. Jeder Benutzer sollte unabhängig von einem anderen Benutzer mit den persistenten Datenobjekten arbeiten können, als ob ihm ein exklusiver Zugriff gewährt wird. Dies bedeutet, daß die Benutzer sich weder gegenseitig bemerken noch gar stören dürfen. Einen Schutz vor Parallelitätsanomalien kann eine *Concurrency Control (Synchronisationskontrolle)* bieten, wie sie über die Aktivitäten und Sperren in H-PCTE angeboten werden (siehe Abschnitt 2.3.2.4, Seite 17).

ProSet führt die Operationen auf persistente ProSet-Werte innerhalb von Transaktionen aus. Innerhalb dieser ProSet-Transaktionen bietet ProSet implizit geeignete Concurrency Control-Mechanismen an, die sich auf die von H-PCTE angebotene Concurrency Control abstützt. Die daraus entstehenden Anforderungen an eine Concurrency Control von ProSet-Transaktionen werden in Abschnitt 3.3 auf Seite 36 erörtert.

Ein gemeinsamer Zugriff auf persistente Datenobjekte erfordert außerdem einen gewissen Grad an *Fehlertoleranz*. Im folgenden Abschnitt werden die Anforderungen an die Fehlertoleranz von persistenten Datenobjekten in ProSet genauer analysiert.

3.2.8 Fehlertolerante persistente Datenobjekte

Eine Programmiersprache, die Persistenz anbietet, sichert zwar das Überleben eines Datenobjekts nach Beendigung seines Programms, jedoch sollte ein persistentes Datenobjekt auch schwerwiegende Fehlersituationen überleben, wie z.B. Prozeßabstürze, Systemabstürze oder Hardware-Fehler des stabilen Speichers (Platten-Crash). Aus der Sicht einer Prototyping-Umgebung sollte außerdem der Betrieb auf das Repository durch die im laufenden Betrieb möglichen Fehlersituationen nicht so gestört werden, daß Inkonsistenzen im Repository auftreten. Aus diesen Gründen werden gewisse Anforderungen an die Fehlertoleranz von persistenten Datenobjekten erhoben.

So sollten Mechanismen angeboten werden, die nach solchen Fehlersituationen automatisch die Konsistenz der Datenobjekte im Repository wiederherstellen. Weiterhin sollte dem Applikationsprozeß eine möglichst hohe Verfügbarkeit der Datenobjekte gewährleistet werden, d.h. die Wiederanlaufzeiten nach schwerwiegenden Fehlern sollten möglichst kurz sein bzw. die Wahrscheinlichkeit eines Fehlerauftritts möglichst klein. Diese Fehlertoleranz kann in einem Repository über ein *Recovery (Zuverlässigkeitskontrolle)* gewährleistet werden. H-PCTE bietet solche Mechanismen an (siehe Abschnitt 2.3.2.5, Seite 18).

In verteilten Systemen kann man außerdem durch die Replikation von Datenobjekten die Fehlertoleranz erhöhen. Eine solche Replikation wird aus zwei Gründen innerhalb dieser Diplomarbeit nur am Rande berücksichtigt: Erstens stellt H-PCTE noch keinen vollständigen Satz an Operationen auf komplexen Objekten zur Verfügung, die aber z.B. für eine effiziente Replikation von Datenobjekten benötigt werden. Zweitens sichert bereits H-PCTE implizit über die Recovery-Mechanismen seines Objektmanagementsystems eine für das Prototyping ausreichende Fehlertoleranz für die in seiner Objektbank abgelegten persistenten Datenobjekte. Desweiteren wäre die Implementierung von Replikationsmechanismen sehr aufwendig und würde sich negativ auf die Zugriffsperformanz auswirken, da man ein Replikationsprotokoll benötigt, um die Konsistenz der replizierten Datenobjekte (*Replikas*) aufrechtzuerhalten.

ProSet führt innerhalb einer Programmeinheit die Operationen auf die in ihr deklarierten persistenten ProSet-Werte immer innerhalb einer Transaktion aus. Aus diesem Grunde muß ProSet implizit ein transaktionsorientiertes Recovery anbieten. Die bei Abarbeitung einer Transaktion auftretenden Fehler, sowie die benötigten Recovery-Arten werden im Zusammenhang in Abschnitt 3.3 auf Seite 36 erläutert.

3.2.9 Versionierung

Beim Prototyping unterliegen der Prototyp sowie die von ihm benutzten persistenten Datenobjekte ständigen Veränderungen. Aus diesem Grunde sollte man in einer Programmiersprache, die Persistenz anbietet, die Entwicklung eines persistenten Datenobjekts über die Zeit durch Erzeugung von Datenobjekt-Versionen protokollieren können. Dabei müssen aber auch die Beziehungen zwischen Versionen verwaltet werden, um z.B. die letzte Version identifizieren, oder eine bestimmte frühere Version wiederherstellen zu können. Diese Funktionalität kann über ein *Versionsmanagement* angeboten werden. In diesem Zusammenhang sei auf den Unterschied zwischen einem Versionsmanagement und einem *Konfigurationsmanagement* hingewiesen: Im Rahmen eines Versionsmanagements wird die zeitliche Entwicklung eines individuellen Datenobjekts verwaltet, wogegen ein Konfigurationsmanagement eine Menge von individuellen Versionen der Datenobjekte verwaltet ([WJ93], Kap. 3.2.6).

ProSet bietet bisher kein Versions- oder gar Konfigurationsmanagement an. Jedoch sollte ProSet gerade als eine Prototyping-Sprache innerhalb einer Prototyping-Umgebung diese Funktionalität anbieten. Ein solches Konzept muß allerdings mit den entsprechenden Mechanismen, die innerhalb einer Prototyping-Umgebung angeboten werden, abgestimmt werden. Da im Rahmen dieser Diplomarbeit nicht schwerpunktmäßig Anforderungen an eine Prototyping-Umgebung untersucht werden und H-PCTE bisher keine effizienten Funktionen zur Versionierung von Objekten anbietet, wird im folgenden kein Konzept für eine Versionierung aufgestellt.

3.2.10 Verteilte Speicherung

3.2.10.1 Motivation

In einem Software-Entwicklungsprozeß sind in der Regel mehrere Benutzer in Gruppen organisiert, die von verteilten Arbeitsplätzen aus an einem Projekt arbeiten ([ES89], Kapitel

3.2.2). Aus diesem Grunde sollte ProSet über eine verteilte Speicherung der persistenten ProSet-Werte eine räumlich verteilte Software-Entwicklung unterstützen.

Nicht zuletzt aufgrund der steigenden Rechenleistung von vernetzten Arbeitsplatzrechnern und der Vielfalt an leistungsfähigen Kommunikationsmöglichkeiten hat die Attraktivität von verteilten Systemen (und der verteilten Datenspeicherung) gerade in den letzten Jahren stark zugenommen. Der Einsatz von verteilten Systemen ist allgemein motiviert durch [SPG91]:

- die *gemeinsame Nutzung von Ressourcen*: Potentiell jeder Knoten eines Rechnernetzes kann die Ressourcen eines anderen Knotens im Netz benutzen, d.h. verschiedene Benutzer können von verschiedenen Rechnern aus gemeinsam auf denselben Daten arbeiten.
- die *bessere Ausnutzung der im Netz vorhandenen Rechenleistung*: Insbesondere rechenintensive Prozesse können auf verschiedene Netzknoten aufgeteilt und somit gleichzeitig bearbeitet werden.
- die *Verbesserung der Zuverlässigkeit*: Nach Ausfall eines oder mehrerer Knoten eines verteilten Systems können die Rechner des übriggebliebenen Teilsystems in der Regel (wenn nicht die übriggebliebenen Netzknoten die Daten der ausgefallenen Knoten zum Weiterarbeiten benötigen) trotzdem weiterarbeiten.

3.2.10.2 Anforderungen an eine verteilte Speicherung

Um u.a. die oben genannten Eigenschaften zu unterstützen, sollten verteilte Systeme folgende allgemeine Anforderungen erfüllen [CD88]:

- **(V1)** *Ortstransparenz:* Eine verteilte Speicherung sollte für den Benutzer transparent sein, d.h. es sollte für ihn nicht sichtbar sein, ob er auf Daten seines Rechners oder die eines entfernten Rechners zugreift. Eine solche Transparenz bietet H-PCTE und sollte auch in ProSet aufrechterhalten bleiben.
- **(V2)** *Lokalität von Daten:* Damit durch die im Rahmen des verteilten Systems gespeicherten Daten eine größere *Zugriffslokalität* erreicht werden kann, sollten Daten dort gespeichert werden, wo sie primär benötigt werden. So wird in der Regel die Zugriffsperformanz erhöht. Eine solche Optimierung fällt in den Aufgabenbereich eines Systemadministrators.
- **(V3)** *Lokale Autonomie:* Jeder Knoten sollte möglichst autonom arbeiten können, d.h. insbesondere, daß der Knoten zur Verrichtung seiner Arbeit nur die lokal auf dem Rechner vorhandenen Daten benötigen sollte.
- **(V4)** *Replikation:* Benötigen mehrere Rechnerknoten bestimmte Daten, um ihre Applikation starten zu können, so sollten diese zur Steigerung der Fehlertoleranz auf den entsprechenden Rechnern repliziert werden. Bei einer solchen Replikation werden dann allerdings Replikationsprotokolle benötigt, die dafür sorgen, daß die replizierten Daten bei Änderungen konsistent bleiben, d.h. wenn man Änderungen an den Daten eines Rechnerknotens vornimmt, so müssen diese Änderungen auch an ihren Replikas vorgenommen werden. Solche Replikationsprotokolle mindern allerdings unter Umständen erheblich die Effizienz bei Datenzugriffen. H-PCTE bietet deshalb keine Möglichkeit einer solchen Replikation. Dies ist auch in diesem Themenkontext nicht notwendig, da wie in Abschnitt 3.2.8 ab Seite 33 erläutert, die Ansprüche an eine Fehlertoleranz beim Prototyping nicht sonderlich hoch sind.

Diese Anforderungen werden zum größten Teil von dem verteilten System H-PCTE berücksichtigt (siehe Abschnitt 2.3.2.3, ab Seite 15). Anforderungen, die nicht von H-PCTE abgedeckt werden können, fallen darüber hinaus bei ProSet in den Aufgabenbereich eines Systemadministrators, können aber durch geeignete Persistenzkonzepte in ProSet unterstützt werden. Diese Persistenzkonzepte werden in Abschnitt 4.1 ab Seite 50 vorgestellt.

3.3 Anforderungsanalyse für ein Transaktionskonzept für ProSet

Im folgenden wird nun übersichtsartig eine Einführung in die Thematik der Transaktionen gegeben, um dann speziell für ein Transaktionskonzept für ProSet eine Anforderungsanalyse durchzuführen. Eine ausführlichere Einführung in Transaktionen im Datenbankbereich bieten [VG93] und [Wei88].

3.3.1 Grundlagen

Unter einer Transaktion versteht man generell eine Folge von Operationen, die eine gegebene Datenbank von einem konsistenten Zustand in einen (nicht notwendig verschiedenen) konsistenten Zustand überführt. Der Grundgedanke ist hierbei, eine so definierte Folge von Operationen als logische, konsistenzerhaltende Einheit aufzufassen, die atomar ausgeführt werden soll, als ob ihr die Datenbank exklusiv zu Verfügung stünde ([Vos94], S.444). Im allgemeinen werden dabei drei Operationen zum Starten (Start), Beenden (Commit) und Abbrechen (Abort) einer Transaktion angeboten.

Bei einem gemeinsamen Zugriff zweier oder mehrerer parallel ablaufender Transaktionen können die drei typische Parallelitätsanomalien *Lost Update*, *Dirty Read* und *Unrepeatable Read* auftreten (nach [Vos94], Kap. 18.2). Um diese Konflikte zu vermeiden, werden die Zugriffe von mehreren Transaktionen auf eine gemeinsam benutzte Menge von Datenobjekten mittels einer Concurrency Control-Komponente synchronisiert.

3.3.1.1 ACID-Prinzip von Transaktionen

Damit parallel abgearbeitete Transaktionen korrekt synchronisiert werden können und eine gewisse Fehlertoleranz gewährleistet ist, wird im Datenbankbereich die Anforderung erhoben, bei der Verarbeitung von Transaktionen folgende vier Eigenschaften zu gewährleisten, die unter dem Begriff *ACID-Prinzip* bekannt sind:

1. *Atomarität (**A**tomicity):*

   Eine Transaktion wird entweder vollständig oder gar nicht ausgeführt. Dies bedingt, daß die durch eine Transaktion vorgenommenen Änderungen auf eine Datenbank bei einem Fehlerauftritt rückgängig gemacht (Undo) bzw. die Datenbank in den konsistenten Zustand gebracht wird, in dem sie sich vor dem Start dieser Transaktion befand (Rollback).

2. *Konsistenz (**C**onsistency):*

   Die Integritätsbedingungen der Datenbank werden bewahrt, d.h. eine Transaktion hinterläßt stets einen konsistenten Zustand.

3. *Isolation (**I**solation):*

   Die Transaktion läuft isoliert von anderen parallel ausgeführten Transaktionen ab. Dies bedeutet, daß insbesondere der Transaktion nur sichere Daten aus der Datenbank zur Verarbeitung zur Verfügung gestellt werden. Sichere Daten sind Daten, die Teil eines konsistenten Datenbankzustands sind.

4. *Dauerhaftigkeit (**D**urability):*

   Wenn die Transaktion erfolgreich beendet wurde (per Commit), überleben die durch sie getätigten Änderungen in der Datenbank jeden Hardware- oder Software-Fehler.

Wenn eine Transaktion die ACID-Eigenschaften besitzt, ist folgendes gewährleistet: Kann eine Transaktion aufgrund eines Fehlers nicht erfolgreich beendet werden, so hinterläßt diese keine „Spuren“ in der Datenbank, da sie *atomar* verarbeitet wird. Der nach einem Fehler wiederherzustellende Datenbankzustand ist wegen der *Dauerhaftigkeits*-Eigenschaft (zusammen mit der Atomaritäts-Eigenschaft) der Transaktion wohldefiniert. Dies ist nämlich jeweils der jüngste Zustand, der alle Änderungen von Transaktionen beinhaltet, die vor dem Fehler erfolgreich abgeschlossen wurden, und somit keine Änderungen von Transaktionen beinhaltet, die zum Fehlerzeitpunkt noch aktiv waren. Nur mit Hilfe der *Isolations*-Eigenschaft von Transaktionen können die Parallelitätsanomalien bei einem Mehrbenutzerbetrieb verhindert werden. Ohne diese Eigenschaft können die Parallelitätsanomalien selbst dann auftreten, wenn jede Transaktion fehlerfrei abläuft.

3.3.1.2 Transaktionsmodelle

Da anwendungsabhängig Transaktionen mit unterschiedlicher Semantik benötigt werden, haben sich verschiedene Transaktionsmodelle entwickelt. Gerade in Entwurfsanwendungen, wie z.B. im CAD oder CASE, hat man jedoch festgestellt, daß es in diesen Anwendungsbereichen wünschenswert sein kann, das ACID-Prinzip auf bestimmten Stufen zu umgehen ([Heu92], Kapitel 9.4). ProSet sieht ein geschachteltes Transaktionskonzept vor [DFKS93]. Nachfolgend werden deshalb als Grundlageinformationen das klassische *flache Transaktionsmodell* und das flexiblere *geschachtelte Transaktionsmodell* kurz vorgestellt.

Flache Transaktionen, mit ihrer Ursprungsform der Debit/Credit-Transaktion ([Vos94], S.538), bestehen aus einer Folge von Operationen und genügen dem ACID-Prinzip. Allerdings weisen flache Transaktionen bei bestimmten Anwendungen Nachteile auf ([GSD93]):

1. Es gibt keine adäquate Möglichkeit einer Modularisierung von Arbeitseinheiten.
2. Es gibt gewöhnlich keine Möglichkeit, zu bestimmten Sicherungspunkten innerhalb einer Transaktion zurückzukehren.
3. Ein transaktionsinterner Parallelismus ist nicht in einer kontrollierten Weise möglich.

Um diese Nachteile zu vermeiden und den Anforderungen eines breiteren Anwendungsbereichs zu genügen [Wei89], ist man dazu übergegangen, ein geschachteltes Transaktionsmodell aufzustellen [Mos85].

Geschachtelte Transaktionen bestehen ebenso wie flache Transaktionen aus einer Folge von Aktionen. Jedoch können nun diese Aktionen wiederum Transaktionen sein, deren Aktionen wiederum Transaktionen sein können, usw. Auf diese Weise kann ein ganzer Transaktionsbaum entstehen, mit einer Wurzeltransaktion und möglicherweise mit mehreren Subtransaktionen auf jeder Ebene dieses Baumes. Nur die Wurzeltransaktion besitzt die ACID-Eigenschaften, wogegen die Subtransaktionen nur die Atomaritäts- und Isolations-Eigenschaft aufweisen.

3.3.1.3 Synchronisationskontrolle

Die Operationsfolgen innerhalb von Transaktionen kann man über das *Read-Write-Modell* beschreiben. Das Read-Write-Modell abstrahiert die Operationen innerhalb einer Transaktion als eine Sequenz aus Lese- und Schreiboperationen. Das Problem ist, die Operationen aller parallel ablaufenden Transaktionen so zu synchronisieren, daß keine der oben genannten Parallelitätsanomalien auftreten können. Zu diesem Thema findet man in der Literatur, wie z.B. in [Wei88] und [VG93], mehrere Synchronisationsprotokolle. Eine Klasse von diesen Protokollen bilden die sperrenden Synchronisationsprotokolle. Hier werden die Parallelitätsanomalien auf Basis des Read-Write-Modells über ein geeignetes Setzen von Schreib- und Lesesperren auf den von mehreren Transaktionen gemeinsam genutzten Datenobjekten vermieden.

- Eine *Lesesperre* verhindert, daß eine andere Transaktion einen Schreibzugriff auf das entsprechende Datenobjekt ausführt, solange diese Transaktion nicht beendet wurde.
- Eine *Schreibsperre* verhindert, daß eine andere Transaktion einen Lese- oder Schreibzugriff auf das entsprechende Datenobjekt ausführt, solange diese Transaktion nicht beendet wurde.

Jede Transaktion versucht, vor Ausführung eines Lesezugriffs auf ein Datenobjekt eine Lesesperre und vor Ausführung eines Schreibzugriffs eine Schreibsperre zu setzen. Das Setzen einer Sperre auf ein Datenobjekt wird erlaubt, wenn diese Sperre nicht in Konflikt mit einer anderen Sperre auf diesem Datenobjekt gerät. Ein Konflikt zwischen zwei Sperren auf ein Datenobjekt tritt auf, wenn eine der beiden Sperren eine Schreibsperre und die jeweils andere eine Lesesperre ist oder, wenn beide Sperren Schreibsperren sind. Anders ausgedrückt darf ein Datenobjekt nur mehrere Lesesperren besitzen, wenn Parallelitätsanomalien ausgeschlossen werden sollen. Gerät der Sperrversuch einer Transaktion in Konflikt mit den Sperren einer anderen Transaktion, so wird sie in einen Wartezustand versetzt. Sobald durch die Freigabe einer Sperre ein Konflikt aufgelöst wird, aktiviert die Concurrency Control-Komponente die entsprechende Transaktion wieder. Ob und zu welchem Zeitpunkt die Sperren innerhalb einer Transaktion gesetzt und wieder freigegeben werden, regelt das Synchronisationsprotokoll.

Bei (geschlossen) geschachtelten Transaktionen ([Wei89]) vererben sich die zur Vermeidung von Parallelitätsanomalien unterhaltenen Sperren auf die Subtransaktionen einer Transaktion, und die innerhalb einer Subtransaktion gesetzten Sperren vererben sich nach einer erfolgreichen Abarbeitung einer Subtransaktion wiederum auf ihre Vatertransaktion. Bei einem Abbruch einer Subtransaktion werden die innerhalb der Subtransaktion gesetzten Sperren freigegeben. Erst nach erfolgreicher Beendigung der Wurzeltransaktion werden alle innerhalb dieser Transaktion und ihrer Subtransaktionen gesetzten Sperren freigegeben. Mit Hilfe der Atomaritäts- und Isolations-Eigenschaft der Subtransaktionen lassen sich diese bei einem Fehler einzeln zurücksetzen, ohne daß die gesamte Vatertransaktion abgebrochen werden muß. So kann man dann in der Regel von dem konsistenten Datenbankzustand aus, auf den nach Abbruch einer Subtransaktion zurückgesetzt wurde, erneut versuchen, die Transaktion erfolgreich abzuarbeiten.

Die Parallelitätsanomalien können also über ein sperrendes Synchronisationsprotokoll verhindert werden. Um jedoch bei Abarbeitung einer Transaktion eine gewisse Fehlertoleranz sicherzustellen, wird ein transaktionsorientiertes Recovery benötigt.

3.3.1.4 Transaktionsorientiertes Recovery

Reuter [Reu87] unterscheidet vier Arten von Recovery:

1. *R1-Recovery:* Erlaubt ein *partielles Zurücksetzen (partial roll-back)* von Transaktionen, d.h. es wird ein isoliertes Zurücksetzen einer oder mehrerer unvollständiger Transaktionen ermöglicht.

2. *R2-Recovery:* Erlaubt bei einem Systemabsturz diejenigen bereits zum Zeitpunkt des Fehlerauftritts erfolgreich beendeten Transaktionen zu wiederholen, die benötigt werden, um die Datenbank wieder in einen konsistenten Zustand zu überführen *(partielles Wiederholen, partial redo).*
3. *R3-Recovery:* Um bei einem Systemabsturz einen konsistenten Zustand zu erreichen, müssen zum Zeitpunkt des Fehlerauftritts noch nicht beendete Transaktionen rückgängig gemacht werden *(complete roll-back, vollständiges Zurücksetzen).*
4. *R4-Recovery:* (*complete redo, vollständiges Wiederholen*): Diese Form des Recoverys wird bei einem Defekt auf einem der nichtflüchtigen externen Speichermedien benötigt. Um nach einem solchen Fehler den jüngsten konsistenten Datenbankzustand wiederherstellen zu können,

   a) muß ein älterer Datenbankzustand mittels einer Archivkopie wiederhergestellt werden. Dieser muß nicht notwendigerweise konsistent sein!

   b) müssen alle zwischen dem Zeitpunkt der Archiverstellung und dem Systemausfall abgeschlossenen Transaktionen vollständig wiederholt werden.

3.3.2 Anforderungen an ein Transaktionskonzept

Im folgenden werden nun die Anforderungen an ein Transaktionskonzept von ProSet untersucht.

Die Operationen innerhalb einer Programmeinheit von ProSet, in der persistente ProSet-Werte deklariert sind, werden als Transaktion ausgeführt. Da ProSet geschachtelte Programmeinheiten anbietet und viele Anwendungen die im vorigen Abschnitt 3.3.1.2 genannten Vorteile eines geschachtelten Transaktionsmodells nutzen wollen, stellt ProSet ein geschachteltes Transaktionsmodell zur Verfügung [DFKS93]. Als Spezialfall enthält dieses Modell auch flache Transaktionen. Um nicht den Umfang der Diplomarbeit zu sprengen und aufgrund der implementierungstechnischen Rahmenbedingungen in Form der Funktionalität von H-PCTE, werden hier keine weiteren Transaktionsmodelle untersucht. Es wäre allerdings als Erweiterung der Diplomarbeit eine detaillierte Analyse wünschenswert, in der durch den Praxiseinsatz von ProSet in verschiedenen Anwendungsbereichen untersucht wird, welche weiteren Transaktionsmodelle benötigt werden.

3.3.2.1 Synchronisationskontrolle in ProSet

Zur korrekten Synchronisation und somit Vermeidung der oben genannten Parallelitätsanomalien muß vor dem Laden eines persistenten ProSet-Werts dieser Wert mit einer Lesesperre und vor dem Speichern eines persistenten ProSet-Werts der entsprechende Wert mit einer Schreibsperre versehen werden.

Desweiteren können Probleme bei einer Concurrency Control aufgrund von Deadlocks und Livelocks auftreten:

1. Ein *Deadlock* tritt auf, wenn n (n>1) Transaktionen Sperren auf ihre bisher zugegriffenen Datenobjekte unterhalten, zusätzlich aber für ein weiteres Datenobjekt eine Sperre setzen wollen, wobei jedoch diese Sperre nicht gesetzt werden kann, da bereits eine andere Transaktion eine zu dieser in Konflikt stehende Sperre auf dieses Datenobjekt unterhält und bei diesen Sperranforderungen der n Transaktionen ein Zyklus aufgetreten ist.
2. Je nach Scheduling-Strategie kann eine wartende Transaktion immer wieder von neuen Transaktionen überholt werden und kommt somit niemals zum Zuge, obwohl sie prinzi-

piell durchaus die Sperre zugeteilt bekommen könnte. Eine Transaktion, die durch eine nicht erfüllte Schreibsperranforderung blockiert ist, könnte durch die Lesesperren von immer neu eintreffenden Transaktionen verdrängt werden. Eine solche Situation nennt man *Livelock*.

Diese Konflikte müssen über geeignete Strategien behandelt werden. Diese werden im Zusammenhang in Abschnitt 4.3 auf Seite 62 präsentiert.

3.3.2.2 Recovery

Hier werden nun die Anforderungen von Abschnitt 3.2.8 auf Seite 33 im Kontext von Transaktionen präzisiert. Wie erwähnt, muß im Fehlerfall ein transaktionsorientiertes Recovery zur Verfügung gestellt werden, damit immer ein konsistenter Objektbankzustand wiederhergestellt werden kann. Bei Abarbeitung einer Transaktion in ProSet können folgende Fehler auftreten, die im Rahmen des transaktionsorientierten Recovery wie folgt behandelt werden sollten:

- **(R1):** Bei einem Konflikt zwischen mehreren parallel ablaufenden Transaktionen oder auch bei einer fehlerhaften Ausführung einer ProSet-Operation sollte ein R1-Recovery geleistet werden.
- **(R2):** Bei einem schwerwiegenden Fehler im ProSet-Programm oder in den diesem Programm unterliegenden Software-Schichten (z.B. Betriebssystem), der zu einem Systemabsturz führt, sollte ein R2- und R3-Recovery geleistet werden.
- **(R3):** Bei einem Hardware-Fehler, der zum Verlust des Objektbankinhalts führt, wie z.B. bei einem Platten-Crash, sollte ein R4-Recovery angeboten werden.
- **(R4):** Desweiteren kann noch ein inkonsistenter Objektbankzustand durch Stromausfall oder andere Umgebungseinflüsse auftreten. In einem solchen Fall sollte man über entsprechende Archivkopien, die z.B. im Rahmen eines Backups erstellt wurden, einen konsistenten Objektbankzustand wiederherstellen. Natürlich muß bei dieser Verfahrensweise dafür gesorgt werden, daß regelmäßig eine Kopie eines *konsistenten* Objektbankzustands angefertigt wird.

Zum Abschluß der Analyse werden nun die wichtigsten Anforderungen an das Transaktionskonzept von ProSet zusammengefaßt.

3.3.2.3 Zusammenfassung

- **(T1):** ProSet sollte geschachtelte Transaktionen anbieten, um dem Programmierer ein flexible zu gebrauchendes Transaktionsmodell anbieten zu können. Dieses Transaktionsmodell genügt den Anforderungen von ProSet an eine persistente Datenhaltung.
- **(T2):** Es wird ein sperrendes Synchronisationsprotokoll benötigt, das keine Parallelitätsanomalien zuläßt. Deshalb muß vor einem Schreibzugriff eine Schreibsperre und vor einem Lesezugriff eine Lesesperre gesetzt werden, wobei man einen Sperrenkonflikt vermeiden muß. Erst bei Beendigung oder Abbruch der jeweiligen Wurzeltransaktion bzw. bei Abbruch einer Subtransaktion dürfen die innerhalb dieser Transaktion gesetzten Sperren aufgehoben werden. Bei Beendigung einer Subtransaktion werden die Sperren an die Vatertransaktion vererbt.
- **(T3):** Es müssen geeignete Strategien, wenn implementierungstechnisch möglich, zur Vermeidung oder zur Behandlung von Deadlocks und Livelocks gefunden werden.

- **(T4):** Die im vorigen Abschnitt aufgestellten Anforderungen an ein transaktionsorientiertes Recovery müssen berücksichtigt werden.

3.4 Anforderungsanalyse für ein Schutzkonzept für ProSet

3.4.1 Motivation und Einführung

Damit nur autorisierte Benutzer auf ein persistent abgelegtes Datenobjekt zugreifen können, benötigt man ein Schutzkonzept, mit dem man die Zugriffe in einer geeigneten Form überwachen und kontrollieren kann. Ein solches Schutzkonzept kann unter dem Gesichtspunkt der Sicherheit allerdings beliebig weit gefaßt werden, da es nicht nur von der Art der Benutzung der persistenten Datenobjekte innerhalb eines ProSet-Programms oder Werkzeugs abhängt, sondern auch von dem organisatorischen Umfeld, der unterliegenden Software, wie dem Betriebsystem, und der physischen und logischen Architektur der unterliegenden Hardware. Eine detaillierte Einführung in die verschiedenen Aspekte der Sicherheitsthematik bieten [Kel91] und [LS87].

Zur Eingrenzung des Sicherheitsbegriffs soll im folgenden Zugriffsschutz jedoch als Mittel verstanden werden, um eine vom Betreiber oder Benutzer eines *Systems* durch eine ungewollte oder unerlaubte Benutzung von persistenten Datenobjekten entstehenden *Schaden* zu verhindern. Schutz ist hier schon als erste Abschwächung von Sicherheit zu verstehen, da primär die persistenten Datenobjekte nur vor Zugriffen über Funktionen der API von H-PCTE geschützt werden können, im Gegensatz zum Schutz vor jeglichen Zugriffen, wie z.B. ein Zugriff über selbstkonstruierte Programme unter Ausnutzung von Kenntnissen der internen Speicherungsstruktur von H-PCTE. Allerdings werden so weit wie möglich Maßnahmen berücksichtigt, welche die Sicherheit des Systems erhöhen.

Um eine vernünftige Grundlage für die folgenden Sicherheitsbetrachtungen zu schaffen, wird nachfolgend vorausgesetzt, daß die dem jeweiligen System unterliegende Hardware und Software sicher ist, d.h. insbesondere fehlerfrei bzgl. der vom System benutzten Funktionalität. Mit Software ist in diesem Zusammenhang das Betriebssystem UNIX, das Objektmanagementsystem von H-PCTE, sowie die Laufzeitbibliothek von ProSet gemeint.

Auf Basis des über H-PCTE zur Verfügung gestellten Repositorys von ProSet kann man zwei Arten von *Systemen* betrachten:

1. Das Repository innerhalb einer SEU auf Basis der Prototyping-Sprache ProSet: Innerhalb ihrer SEU soll ProSet ein kooperatives Arbeiten auf den von ProSet-Programmen oder Werkzeugen erzeugten persistenten Datenobjekten unterstützen. Aus diesem Grunde wurde bereits die Forderung aufgestellt, ProSet mehrbenutzerfähig zu machen. Damit jedoch Zugriffe von Benutzern auf die persistenten Datenobjekte oder auch der Ressourcenverbrauch in geeigneter Form kontrolliert werden kann, werden darüber hinaus *Schutzmechanismen* benötigt. Dieser Begriff wird im nächsten Abschnitt genauer definiert. So sollte z.B. ein Benutzer seine persistenten ProSet-Werte vor bestimmten Zugriffen anderer Benutzer schützen können.
2. Jegliche mit Hilfe von ProSet konstruierte Applikationen (Prototyp bzw. Produktionsprogramm), die auf die im Repository abgelegten persistenten ProSet-Werte und P-Files zugreifen: Bei einem Gebrauch von ProSet als eine Programmiersprache, die Persistenz

anbietet, stellt man in vielen Anwendungsbereichen fest, daß man zusätzliche Sprachkonstrukte innerhalb der Sprache benötigt, um z.B. einen anwendungsspezifischen Zugriffsschutz realisieren zu können. Auf diese Weise kann dann der Anwendungsbereich der Sprache erweitert werden.

In beiden Fällen ist eine wesentliche Anforderung an ein Repository, die persistenten Datenobjekte einer Applikation bzw. eines Werkzeugs in einheitlicher Weise unter zentraler Kontrolle zu verwalten und sie den einzelnen Benutzern und Benutzergruppen entsprechend ihren Erfordernissen zugänglich zu machen. Gemäß den beiden oben erwähnten Systemarten, sollte ein *Schutzkonzept* sowohl Anforderungen einer SEU als auch Anforderungen aus dem Anwendungsbereich von ProSet berücksichtigen.

Im folgenden werden nun auf Basis von [Kel91] als Grundlage zur Anforderungsanalyse eines Schutzkonzepts für ProSet verschiedene Begriffsdefinitionen aufgestellt und die innerhalb eines Schutzkonzepts benötigten Schutzmechanismen vorgestellt, um dann die Anforderungen an ein Schutzkonzept für ProSet zu analysieren. Die Informationen zu diesem Thema sind ausführlicher, da ProSet bisher kein Schutzkonzept anbietet. .Zu beachten ist, daß die in [Kel91] definierten Begriffe bzgl. des hier verwendeten Zugriffsschutz-Begriffs angepaßt werden.

3.4.2 Grundlagen

Über eine *Sicherheitsmaßnahme* wird hier das Ziel verfolgt, einen im obigen Sinne definierten Schaden zu verhindern, das Ausmaß des Schadens zu begrenzen oder einen eingetretenen Schaden zu entdecken.

Eine konkrete Auswahl an Sicherheitsmaßnahmen und ihre organisatorische Einbettung wird *Sicherheitsstrategie* genannt.

Zur Realisierung einer Sicherheitsstrategie für ein System muß sowohl eine *organisatorische* als auch eine *automatisierte Sicherheitsstrategie* festgelegt werden.

Im Rahmen einer *organisatorischen Sicherheitsstrategie* wird eine Menge von Gesetzen, Vorschriften, Regeln und Praktiken spezifiziert, die reguliert, wie eine Organisation die persistenten Datenobjekte, für die gewisse Sicherheitsziele zu erreichen sind, verwaltet, schützt und verteilt.

Eine *automatisierte Sicherheitsstrategie* beinhaltet eine Menge von Sicherheitsmaßnahmen, Restriktionen und sonstigen Eigenschaften, durch die ein System die Benutzung von persistenten Datenobjekten verhindert, wenn hierbei organisatorische Sicherheitsstrategien verletzt werden würden. Eine solche automatisierte Sicherheitsstrategie soll im folgenden kurz *Schutzkonzept* genannt werden.

Eine *Sicherheitsfunktion* ist eine Menge von zusammengehörigen Schnittstellen und/oder Eigenschaften, die eine Ebene der darüberliegenden Ebene zur Realisierung von Sicherheitsmaßnahmen anbietet.

Nach obigen formulierten Voraussetzungen, kann hier vorausgesetzt werden, daß die dem System unterliegenden Schichten bereits die in [Kel91] genannten Sicherheitsfunktionen, wie Initialisierung von Speicherbereichen, Gewährleistung der Funktionsfähigkeit und eine Datenübertragungssicherung implizit gewährt.

Innerhalb eines Schutzkonzeptes müssen bzw. sollten nach [Kel91] und [Wec84] folgende Sicherheitsfunktionen berücksichtigt werden:

1. Innerhalb einer *Zugangskontrolle* wird über die *Identifikation* und *Authentifikation* von Benutzern der Zugang zum System kontrolliert, d.h. über diese Funktionen, die z.B. über eine Login-Prozedur (Beispiel UNIX) implementiert werden, wird erzwungen, daß sich jeder Benutzer beim System anmeldet und über weitere „Beweise", wie einem Paßwort, seine Anmeldung verifiziert.

   - *Identifikation* bezeichnet den Vorgang vor Beginn einer Sitzung, in dem sich der Benutzer beim System identifiziert.
   - *Authentifikation* bezeichnet den Prüfvorgang der Identifikation eines Benutzers durch das System.

2. Die *Zugriffskontrollen* umfassen insbesondere Funktionen zur *Rechteverwaltung* und zur *Autorisierung von Zugriffen* (Rechteprüfung). Über Zugriffskontrollen wird ermöglicht, einzelnen *Subjekten* eine bestimmte Operation (oder eine Menge von Operationen) auf ein *Objekt* zu erlauben oder zu verbieten.

   - Ein *Schutzobjekt* soll im folgenden eine identifizierbare zu schützende *Einheit (Granul)* bezeichnen. Eine *Einheit* oder *Granul* kann in PROSET ein persistenter PROSET-Wert oder ein (Sub-)P-File bzw. in H-PCTE eine Objekt- oder Link-Ressource sein.
   - Ein *Subjekt* soll desweiteren eine aktive Einheit in dem System bezeichnen, die vom System angebotene Operationen auf Schutzobjekte ausführt. Ein Subjekt kann somit ein Benutzer sein, aber auch eine Benutzergruppe oder ein Benutzer in einer adoptierten Rolle. Es wird nachfolgend davon ausgegangen, daß ein Benutzer innerhalb eines Prozesses Operationen auf Schutzobjekte ausführt. Da ein Benutzer z.B. Mitglied in mehreren Benutzergruppen sein kann, jedoch oft nur von einer Teilmenge dieser Gruppen die Rechte wahrgenommen werden sollen, bietet ein System oft an, innerhalb des Benutzerprozesses verschiedene Gruppen (oder Rollen) zu aktivieren, deren Rechte der Prozeß ausnutzen darf. Diese Subjekte werden im folgenden *aktive Subjekte* genannt.
   - *Autorisierung* bezeichnet die Vergabe von Nutzungsrechten an ein Subjekt, d.h. bei einem Zugriff eines Subjekts auf ein Schutzobjekt wird anhand der Rechte, die das Subjekt für das Schutzobjekt besitzt, geprüft, ob dem Subjekt ein Nutzungsrecht für das Schutzobjekt erteilt werden kann oder nicht.
   - Ein *Recht* besagt, daß ein Subjekt eine Operation auf ein Schutzobjekt ausführen darf.
   - Über die *Rechteverwaltung* werden Funktionen zur Verfügung gestellt, durch die dem System mitgeteilt werden kann, welche *Subjekte* wie auf welche *Objekte* zugreifen dürfen.

   Nach [Kel91] können diese Funktionen den Informationsfluß zwischen Subjekten, den Zugang zu Informationen (Lesen), das Erzeugen (Löschen) oder Ändern von Informationen, die dabei benutzten Speicher und allgemein die Benutzung von Ressourcen kontrollieren bzw. einschränken.

3. *Auditing*: Über Auditingfunktionen werden sicherheitsrelevante Ereignisse protokolliert. Eine solche Protokollierung soll dazu dienen, später feststellen zu können, wie ein vorhandener Systemzustand entstanden ist. So kann man evtl. einen Mißbrauch von Rechten anhand dieser Protokolle nachträglich entdecken.

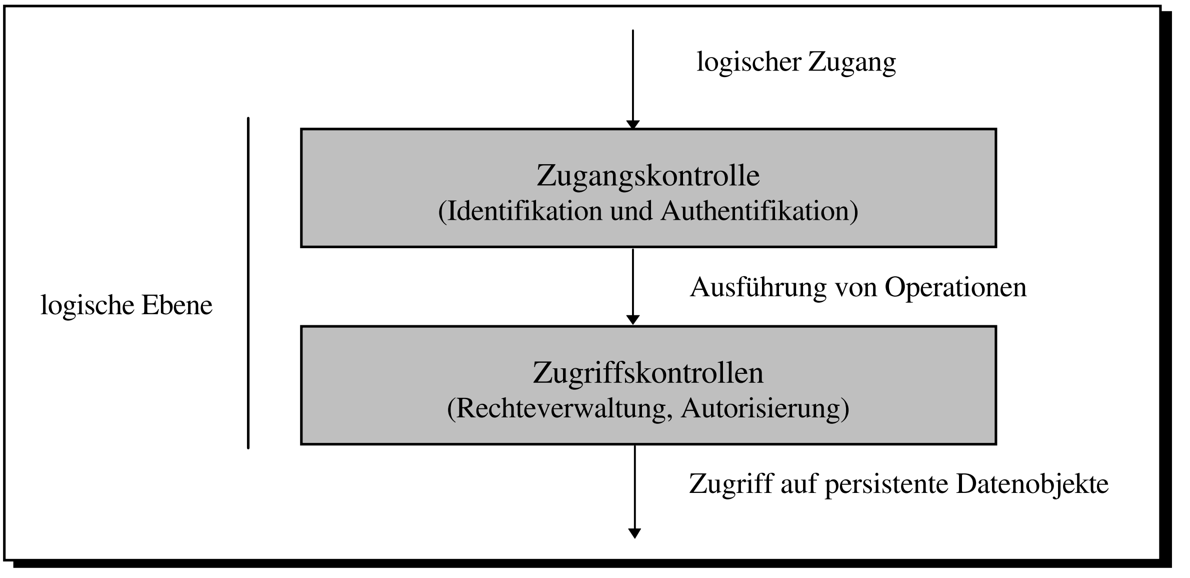


Abbildung 3-3: Sicherheitsfunktionen der logischen Schutzebene

Schutzmechanismen sollen im folgenden durch entsprechende Datenstrukturen und Algorithmen implementierte Sicherheitsfunktionen bezeichnen, also im wesentlichen Funktionen für eine Identifikation und Authentifikation von Subjekten, für eine Benutzer- und Gruppenverwaltung, sowie Funktionen zur Rechteverwaltung, für die Autorisierung von Zugriffen und für ein Auditing.

3.4.3 Anforderungen an ein Schutzkonzept

Es wird nun untersucht, welche Schutzmechanismen in ProSet benötigt werden.

Wie oben beschrieben, können die Sicherheitsanforderungen an ein System stark variieren. In Bezug auf das Prototyping mit ProSet kann man jedoch feststellen, daß es primär von Interesse ist, den Anforderungen einer SEU zu genügen und den Anwendungsbereich der Sprache durch Funktionen zur Realisierung von Schutzmechanismen zu erweitern. Da die Schutzmechanismen von H-PCTE gerade dazu ausgelegt sind, eine SEU geeignet zu unterstützen, hat man also den ersten Anforderungsbereich zum größten Teil abgedeckt. Der zweite Anforderungsbereich (Erweiterung des Anwendungsbereichs) kann nur dadurch unterstützt werden, daß Administrationsfunktionen zur Nutzung und Anpassung der ProSet-Schutzmechanismen über ein ProSet-Modul innerhalb der Sprache zur Verfügung gestellt werden und die Funktionen möglichst so allgemein (generisch) angeboten werden, daß auf diesen basierend auch anwendungsspezifische Schutzmechanismen implementiert werden können.

Nun werden die wichtigsten Anforderungen nach [Kel88], [Kel91] und [Reu87] an ein Schutzkonzept analysiert und dabei jeweils die Konzepte von H-PCTE genannt, die gewissen Anforderungen bereits genügen.

- **(S1) Eindeutige Schnittstellen:** Der Zugang zum Repository, hier der Objektbank von H-PCTE, sollte nur und ausschließlich über die vom System vorgegebenen Schnittstellen zum Anwendungsprogramm möglich sein. H-PCTE stellt diese Schnittstelle über eine API zur Verfügung, jedoch muß dieser Schnittstelle noch eine weitere auf den Bedarf von ProSet abgestimmte Schnittstelle übergeordnet werden.

- **(S2) Geringe Beeinflussung der Systemleistung:** Der Aufwand zur Realisierung der Sicherheitsfunktionen als ProSet-Schutzmechanismen muß auf den tatsächlichen Schutzbedarf abgestimmt werden. D.h. man muß die Kosten in Form eines Laufzeit- und Administrationsaufwands mit dem Nutzen in Form eines ausreichenden Schutzes abwägen. Die Laufzeitkosten sind zwar aufgrund der steigenden Rechenleistung und sinkenden Hardwarepreise eher zweitrangig, jedoch ist der Administrationsaufwand, d.h. insbesondere der Aufwand um sich die notwendigen Kenntnisse für eine adäquate Handhabung der Administrationsfunktionen anzueignen, unter Umständen erheblich. Dieser Administrationsaufwand ist insbesondere wegen der Anforderung (P3) möglichst gering zu halten.

- **(S3) Modellierung und Verwaltung von Subjekten:** Man muß Subjekte modellieren können. Insbesondere innerhalb einer SEU muß man neben Benutzern auch Hierarchien von Gruppen und Rollen modellieren können.

  Da H-PCTE gerade zur Unterstützung von den organisatorischen Gegebenheiten einer SEU konstruiert und PCTE gerade wegen der ausreichenden Berücksichtigung dieser Anforderungen standardisiert wurde, sollte man somit die Möglichkeiten zur Modellierung von hierarchischen Arbeitsgruppen und Rollen innerhalb eines Software-Entwicklungsprozesses auch in ProSet anbieten. H-PCTE stellt jedoch nur rudimentäre Funktionen zur Modellierung von Subjekten zur Verfügung. Aus diesem Grund wird ein Administrationswerkzeug zur Benutzer- und Benutzergruppenverwaltung benötigt, um sowohl geeignet weitere Benutzer beim System *anmelden*, d.h. ihre Identifizierungen beim System bekannt machen, als auch Gruppenhierarchien und Rollen modellieren zu können.

  Diese entsprechenden Arbeitsgänge sollte nur ein Systemadministrator ausführen dürfen, da die Benutzer- und Benutzergruppenverwaltung ein sensibler Teil einer Schutzkomponente des Systems ist.

- **(S4) Schutzmechanismen zur Identifikation und Authentifikation:** Prinzipiell können Benutzer natürlich nicht direkt auf die persistenten ProSet-Werte zugreifen, sondern nur über eine Schnittstelle, hier der API von H-PCTE, einen Prozeß starten, der in ihrem Auftrag bestimmte Zugriffe auf ein Schutzobjekt ausführt. Die Benutzer müssen also im System in einer geeigneten Form als Subjekte repräsentiert werden, und jedem Prozeß muß genau ein Benutzer zugeordnet werden, in dessen Auftrag der Prozeß arbeitet. Somit wird ein geeigneter Schutzmechanismus für eine Identifikation und Authentifikation der Benutzer benötigt. Wie in Abschnitt 2.3.2.5 ab Seite 18 erläutert, stützt sich H-PCTE auf den über die login-Prozedur von UNIX zur Verfügung gestellten Schutzmechanismus ab. Nur ein in der Benutzerverwaltung von UNIX eingetragener Benutzer kann auch in H-PCTE als Benutzer angemeldet werden. Somit kann sich diesbezüglich auch ProSet auf die von UNIX angebotenen Mechanismen zur Identifikation und Authentifikation abstützen.

- **(S5) Zugriffskontrollen:**

  Ziel von Zugriffskontrollen ist es hier,

  1. die Vertraulichkeit bzw. die Integrität der in einem Schutzobjekt unterhaltenen Informationen zu schützen und
  2. eine unzulässige Benutzung von Ressourcen zu verhindern. Dabei kann allerdings der Zugriff auf Ressourcen primär qualitativ als quantitativ eingeschränkt werden, indem

die Nutzung von Ressourcen entweder verhindert oder auf bestimmte Nutzungsarten bzw. Operationen beschränkt wird.

Man muß also erstens festlegen, welche Schutzobjekte modelliert werden müssen und zweitens muß man jedem Subjekt bestimmte Rechte auf ein Schutzobjekt zuteilen können. Die Rechte sollten dabei so feingranular wie möglich verwaltet werden, damit jedem Benutzer exakt die Rechte gewährt werden können, die dieser zur Erfüllung seiner Aufgabe benötigt (*Prinzip des kleinstmöglichen Privilegs, least privilege principle*).

H-PCTE bietet, wie in Abschnitt 2.3.2.5 auf Seite 18 beschrieben, diskretionäre Zugriffskontrollen an.

- **(S6) Wahl der Schutzobjekte:** In ProSet müssen sowohl P-Files und Sub-P-Files als auch die persistenten ProSet-Werte geschützt werden.
- **(S7) Wahl der Zugriffsrechte:** Gemäß den in ProSet angebotenen Operationen und den unter (S6) festgelegten Schutzobjekten, muß man Zugriffsrechte sowohl für (Sub-)P-Files als auch für die persistenten ProSet-Werte definieren. Bei der Definition der Rechte muß darüber hinaus ein Kompromiß zwischen der Anforderung (S5) und den Anforderungen (S2) und (P3) gefunden werden.

  Da eine Kenntnis der UNIX-Rechte aufgrund der Verbreitung von UNIX bei jedem Benutzer fast vorausgesetzt werden kann, und die UNIX-Rechte hier auch im wesentlichen übertragbar sind, sollte man aufgrund der Anforderung (P3) auch read(r), write(w) und execute(x) Rechte innerhalb von ProSet vergeben.

  Ein persistenter ProSet-Wert kann somit von einem Subjekt

  - geladen werden, wenn das Subjekt ein read-Recht auf dieses Schutzobjekt besitzt,
  - gespeichert und gelöscht werden, wenn das Subjekt ein write-Recht auf dieses Schutzobjekt besitzt.

  Das execute-Recht wird hier nicht benötigt und kann somit als defaultmäßig immer gesetzt interpretiert werden. Ein persistenter ProSet-Wert kann somit mit einer Datei in UNIX verglichen werden.

  Darüber hinaus kann ein Subjekt

  - einen persistenten ProSet-Wert erzeugen oder löschen, wenn das Subjekt beim zugehörigen (Sub-)P-File ein write-Recht besitzt.
  - den Inhalt eines (Sub-)P-Files auflisten, wenn das Subjekt ein read-Recht auf das (Sub-) P-File besitzt.
  - über das H-PCTE-Objekt, das einen (Sub-)P-File repräsentiert, navigieren und auf einen persistenten ProSet-Wert zugreifen, wenn das Subjekt ein execute-Recht auf das (Sub-) P-File besitzt.

  Somit kann ein P-File bzw. Sub-P-File mit einem Directory in UNIX verglichen werden.

  Da H-PCTE drei Rechtewerte anbietet, über denen man, wie in Abschnitt 2.3.2.5 ab Seite 18 erläutert, auch explizite Verbote aussprechen kann, sollte auch ProSet den dadurch entstehenden Flexibilitätsvorteil bei der Modellierung der Zugriffsrechte drei Rechtewerte zur Verfügung stellen. Da es in den Aufgabenbereich der Persistenzschnittstelle bzw. von Werkzeugen fällt, dem Benutzer zu ermöglichen, Zugriffsrechte zu setzen, steht es frei aus Gründen der Kompatibilität mit den UNIX-Rechten, explizit nur die in UNIX bekannten zwei Rechtewerte anzubieten. Es sei noch einmal darauf hingewiesen, daß sowohl bei der dreiwertigen Rechtelogik als auch bei der zweiwertigen Rechtelogik die Rechte der für einen Benutzer aktiven Subjekte über spezielle Regeln miteinander verknüpft werden und erst aus dieser Verknüpfung ein Zugriffsrecht für den Benutzer abgeleitet wird.

Eine weitere Forderung in diesem Zusammenhang ist, daß ein *positives Rechtekonzept* verfolgt werden sollte, d.h. ein Subjekt sollte nur die Rechte besitzen, die ihm explizit erteilt worden sind, sonst keine. H-PCTE verfolgt dieses Konzept, indem ein für ein Subjekt nicht vorhandener ACL-Eintrag bei einem H-PCTE-Objekt interpretiert wird, als ob der entsprechende Zugriffsmodus für ihn nicht gewährt ist. In ProSet muß dieses Konzept jedoch auch umgesetzt werden.

- **(S8) Berücksichtigung der organisatorischen Sicherheitsstrategie:** Bei diskretionären Zugriffskontrollen ist nicht ohne weiteres der Schutz der Vertraulichkeit bzw. Integrität von Informationen gesichert [Kel91]. Man kann z.B. durch einfaches Kopieren von Schutzobjekten das Recht als Besitzer des Objekts erwerben, die Zugriffsrechte auf das Objekt zu verändern. Die Wirksamkeit von diskretionären Zugriffskontrollen hängt entscheidend davon ab, daß die Besitzer von Objekten Rechte entsprechend der zu realisierenden *organisatorischen Sicherheitsstrategie* vergeben und die Rechte auch nur gemäß der *organisatorischen Sicherheitsstrategie* ausgenutzt werden.

- **(S9) Festlegen von Besitzern:** Bei diskretionären Zugriffskontrollen dürfen nur die Besitzer eines Schutzobjekts die Rechte von Subjekten auf dieses Objekt ändern. In ProSet ist ein spezieller Benutzer als Systemadministrator Besitzer von jedem Schutzobjekt. Darüber hinaus soll der Besitzer eines persistenten ProSet-Werts der Benutzer sein, der diesen erzeugt hat. Da ein P-File für einen Benutzer nur durch den Systemadministrator angelegt werden kann, ist der Besitzer eines P-Files der Benutzer, für den diese angelegt wurden und der als Besitzer eingetragen ist. Da ProSet gerade auch eine kooperative verteilte Software-Entwicklung unterstützen soll, ist es wünschenswert, daß nicht nur ein Systemadministrator existiert. Zur Unterstützung der Autonomie von räumlich verteilt arbeitenden Gruppen sollte die Möglichkeit gegeben sein, mehrere *Gruppen-Administratoren* angeben zu können, die jeweils in ihrer Gruppe die Rolle eines Systemadministrators übernehmen.

- **(S10) Integrale Zugriffskontrollen:** Die Vorkehrungen zur Zugriffskontrolle beeinflussen andere Komponenten wie die Sperrkontrolle und Transaktionen. Aus diesem Grunde muß das Schutzkonzept auch auf andere Persistenzkonzepte bzw. H-PCTE-Mechanismen abgestimmt werden.

- **(S11) Trennung der Adreßräume:** Die zu schützenden Datenobjekte und der Code einer Applikation sollten sich nicht in dem gleichen virtuellen Adreßraum befinden, denn sonst könnte durch einen Fehler des Programms oder auch vorsätzlich (mit Hilfe entsprechender Speicheradressierung) direkt auf die Daten zugegriffen werden. Diese Anforderung kann in H-PCTE verletzt werden, da es möglich ist, Segmente lokal in den Adreßraum eines Applikationsprozesses zu laden. Eine Lösung dieses Problems wird im Zusammenhang mit dem Verteilungskonzept im Abschnitt 4.1 auf Seite 50 vorgestellt.

- **(S12) Auditing:** Es sollten die sicherheitsrelevanten Ereignisse über Auditingfunktionen protokolliert werden. Sicherheitsrelevante Ereignisse sind hier das Laden und Speichern von persistenten ProSet-Werten, sowie das Erzeugen und Löschen von Einträgen eines (Sub-) P-Files. Da ProSet im Rahmen des Prototypings nicht sonderlich hohe Sicherheitsansprüche stellt und ein Trade-Off zwischen einem Auditing und der Anforderung

(S2) besteht, sollte nur das Speichern eines persistenten ProSet-Werts protokolliert werden. Denn ein Speichern bedeutet fast immer eine Änderung eines persistenten ProSet-Werts. Darüber hinaus wird nur als Analogie zu einem Dateisystem protokolliert, wann und von wem bzw. für wen ein persistenter ProSet-Wert oder ein (Sub-)P-File erzeugt wurde.

Ein Schutzkonzept, daß diese Anforderungen erfüllt, wird in Abschnitt 4.2 präsentiert.

3.5 Zusammenfassung der Anforderungen

Im folgenden wird als Zusammenfassung eine grobe Übersicht über die Anforderungen gegeben, die im Rahmen der Persistenzkonzepte von ProSet berücksichtigt werden:

- Alle ProSet-Werte mit Bürgerrechten erster Klasse müssen persistent gemacht werden können. Hierzu werden Operationen zum Laden und Speichern der persistenten ProSet-Werte, sowie eine Strategie benötigt, um über den ProSet-Compiler die entsprechenden C-Funktionsaufrufe generieren zu können.
- P-Files und Sub-P-Files darf nur ein Systemadministrator anlegen und sollten jeweils auf einem eigenen Segment abgelegt werden. Es wird ein entsprechendes Werkzeug benötigt. Darüber hinaus sollen die Segmente der P-Files und Sub-P-Files lokal oder global geladen werden können. Es wird eine Operation benötigt, um erstens die lexikalische Korrektheit des P-File-Pfads und zweitens die Existenz eines solchen (Sub-)P-Files zu prüfen.
- Die Anforderungen an eine verteilte Speicherung von persistenten ProSet-Werten werden innerhalb eines Verteilungskonzepts für ProSet berücksichtigt.
- Es wird ein Mehrbenutzerbetrieb auf persistente ProSet-Werte realisiert.
- Das geschachtelte Transaktionskonzept von ProSet wird realisiert.
- Gemäß den Sicherheitsanforderungen von ProSet wird ein Schutzkonzept aufgestellt und realisiert.
- Die Operationen werden über eine Persistenzschnittstelle dem ProSet-Compiler zur Verfügung gestellt. Darüber hinaus wird eine Schnittstelle für Werkzeuge angeboten. Außerdem wird eine Schnittstelle zur Verfügung gestellt, die auf ProSet abgestimmte C-Funktionen zur Unterstützung diverser Persistenzkonzepte anbietet, um später einmal diese über ProSet-Module in einem ProSet-Programm benutzen zu können.
- Da sowohl ProSet als auch H-PCTE stetig weiterentwickelt werden und darüber hinaus die Persistenzschnittstelle von mehreren unterschiedlichen Applikationen (ProSet-Compiler, Werkzeuge, reine H-PCTE-Applikationen) benutzt werden sollen, wird soweit wie möglich eine Schichten-Architektur der Module innerhalb der Persistenzschnittstelle unterstützt.
- Bei Fehlersituationen werden innerhalb der Persistenzschnittstelle zur Unterstützung der Orthogonalität der Persistenzkonzepte (P3) gemäß der Ausnahmebehandlung von ProSet Ausnahmen generiert. Um jedoch z.B. Werkzeuge in ihrer Arbeit nicht negativ zu beeinflussen, werden diese Ausnahmen erst in der obersten Schicht der Persistenzschnittstelle generiert.

Als Abgrenzung dieser Diplomarbeit werden nun Anforderungen an ProSet genannt, die nicht berücksichtigt werden:

- Die Parallelprogrammierung kann nicht unterstützt werden, da H-PCTE zur Zeit weder ein geschachteltes Prozeßmodell, geschweige denn leichtgewichtige Prozesse unterstützt. Eine Erweiterung von H-PCTE diesbezüglich würde den Rahmen dieser Diplomarbeit sprengen.
- Die Versionierung von persistenten ProSet-Werten wird nicht über Versionsmanagement-Funktionen unterstützt, da H-PCTE keine Kopierfunktionen für komplexe Objekte anbietet. Allerdings wird insbesondere im Verteilungskonzept berücksichtigt, daß man manuell Kopien von persistenten ProSet-Werten und P-Files anlegen kann, damit der Benutzer so ansatzweise auf „Versionen“ arbeiten kann.
- Das Formulieren von expliziten Integritätsbedingungen wird nicht unterstützt, obwohl der Autor sich hierzu Gedanken gemacht hat. Eine den wissenschaftlichen Anforderungen genügende Präsentation würde den Umfang der Diplomarbeit zu sehr ausweiten. Eine entsprechende Methodik beim Programmieren mit ProSet läßt allerdings zu, zumindest Integritätsbedingungen zu formulieren und zu prüfen. Wenn man jeden benutzten persistenten ProSet-Wert und die auf ihn definierten Operationen z.B. über das Modul-Konzept von ProSet als Abstrakten Datentyp (ADT) konstruiert, dessen Operationen persistent abgelegt werden, so kann man innerhalb dieser Operationen Integritätsbedingungen prüfen und eine Verletzung über entsprechende Ausnahmen signalisieren. Voraussetzung ist natürlich, daß jeder Benutzer auch nur über die persistent abgelegten Funktionen auf den persistenten ProSet-Wert zugreift.

3.6 Systemumgebung

Die Systemumgebung besteht aus Sun SPARC Workstations mit den UNIX-Betriebssystemen SunOS 4.1.3 (Solaris 1) und SunOS 5.x (Solaris 2.x). Als Compiler wird zur Zeit der C-Compiler Gnu-C in den Versionen 2.4.5 und 2.6.0 benutzt. Darüber hinaus wird der ProSet-Compiler in der Version 0.5 und die aktuelle H-PCTE-Version 2.6 verwendet (mit der Diplomarbeit begonnen wurde auf Version 2.4).

4 Entwurf

In diesem Kapitel werden nun die Anforderungen im Rahmen des Entwurfs der Persistenzkonzepte berücksichtigt. Hierzu wird auf Basis von H-PCTE ein Verteilungskonzept, ein Schutzkonzept und ein Transaktionskonzept entworfen. Darüber hinaus wird die Modularchitektur der Persistenzschnittstelle vorgestellt.

4.1 Verteilungskonzept von ProSet

Gemäß den in Abschnitt 3.2.6 ab Seite 30 aufgestellten Anforderungen bzgl. eines Cachings und Clusterings von persistenten ProSet-Werten sollte man die von H-PCTE angebotenen Zugriffsoptimierungsmöglichkeiten ausnutzen. Desweiteren wurde bereits in Abschnitt 3.2.10 ab Seite 34 motiviert, daß eine verteilte Speicherung von persistenten ProSet-Werten unterstützt werden sollte.

Aus diesen Gründen wurde die Anforderung erhoben, für jedes P-File und Sub-P-File jeweils ein eigenes Segment in H-PCTE anzulegen, auf dem dann alle seine persistenten ProSet-Werte abgelegt werden. Eine solche Zuordnung eines P-File bzw. Sub-P-Files auf ein Segment unterstützt sowohl eine verteilte Speicherung als auch die Möglichkeit eines effizienteren Zugriffs auf die im (Sub-)P-File abgelegten persistenten ProSet-Werte. Denn auf diese Weise stellt jedes P-File bzw. Sub-P-File ein Verteilungsgranul dar, das auf verschiedenen Datenträgern eventuell verschiedener Rechner gespeichert werden kann, und deren persistente ProSet-Werte wahlweise lokal im Applikationsprozeß oder global im Server-Prozeß geladen werden können, um eine hinsichtlich der Zugriffsperformanz in H-PCTE optimale Leistung zu erzielen.

Möchten mehrere Benutzer oder auch nur mehrere Applikationen eines Benutzers auf die persistenten ProSet-Werte eines P-Files zugreifen, muß das Segment des (Sub-)P-Files global im Server geladen werden, ansonsten kann es zur Performanzsteigerung lokal im Applikationsprozeß geladen werden.

Zur Unterstützung dieses Konzeptes besitzt jedes P-File-Objekt in H-PCTE ein zusätzliches Attribut `public` vom Attributtyp `boolean`. Mit Hilfe dieses Attributs kann der Benutzer über ein entsprechendes Administrationswerkzeug bestimmen, ob ein P-File bzw. Sub-P-File während der Ausführung eines ProSet-Programms lokal oder global geladen werden sollte. Ist der Attributwert `true`, so wird sein Segment global im Server geladen, so daß sowohl mehrere Applikationsprozesse verschiedener Benutzer als auch mehrere Applikationsprozesse eines Benutzers auf die persistenten ProSet-Werte des entsprechenden P-Files zugreifen können. Bei einem Attributwert `false` hingegen wird das Segment lokal im Client geladen, so daß zwar nur immer ein Applikationsprozeß auf die persistenten ProSet-Werte des P-Files zugreifen kann, aber der Prozeß so eine optimale Zugriffsperformanz erzielt. Im weiteren wird dieses Konzept mit *Private/Public-Deklaration* von (Sub-)P-Files umschrieben. Hierbei treten allerdings diverse Probleme auf: In H-PCTE wird entweder implizit ein Segment geladen [Pla91], sobald man zu einem Objekt auf einem noch nicht geladenen Segment navigiert, oder explizit, indem man eine API-Funktion `HPcte_segment_load` [H94] aufruft. Diese Funktion referenziert ein Segment nur über seine Segment-ID. Bei der impliziten Methode hat man keine Möglichkeit, wahlweise das Segment lokal oder global zu laden. Allerdings wird in einer zukünftigen H-PCTE-Version in Aussicht gestellt, daß man über ein entsprechendes Setzen eines bestimmten Attributs eines *Segmentobjekts* (siehe Abschnitt 2.3.2.3, Seite 15) diese Wahlmöglichkeit hat. Solange H-PCTE allerdings eine solche Funktionalität nicht zur Verfügung stellt, kann man das Konzept nur über einen expliziten Aufruf der API-Funktion realisieren. Dabei benötigt man jedoch zusätzlich zum `public`-Attribut ein Attribut, in dem die ID des Segments gespeichert ist, auf welches ein P-File bzw. Sub-P-File abgelegt ist. Da ein Administrator ein (Sub-)P-File auf mehrere Segmente aufteilen können soll, werden die Segment-IDs durch Kommata getrennt in einem `on_segments`-Attribut vom Typ `string` verwaltet. Das Objekt, das dieses Attribut verwaltet, darf sich allerdings nicht selbst auf einem dieser Segmente befinden, da man zum Auslesen des Attributs auf das Objekt navigieren muß und somit das Segment bereits implizit geladen würde. Die deshalb benötigte Objektstruktur wird in Abschnitt 4.1.2 erläutert.

Die Private/Public-Deklaration von P-Files bzw. Sub-P-Files kann allerdings mit dem Schutzkonzept in Konflikt geraten, da beim lokalen Laden eines Segments eines P-Files oder Sub-P-Files die Anforderung (S11) verletzt wird. In [See93] wurden allerdings einige Konzepte formuliert, die helfen sollen, dieses Problem zu mindern:

- Es sollten nur Objekte eines Segments in den Adreßraum des Applikationsprozesses geladen werden, auf die aktive Subjekte des Prozesses Zugriffsrechte besitzen und die für den Prozeß bzgl. seines Working Schemas sichtbar sind.
- Darüber hinaus kann der Adreßraum partitioniert werden, und die Partitionen können über spezielle Betriebssystemfunktionen geschützt werden.

Diese Konzepte sind in der aktuellen Version 2.6 von H-PCTE noch nicht implementiert. Besonders interessant ist in diesem Zusammenhang ein Schutz des Adreßraums, in dem die H-PCTE Segmente lokal geladen werden: Da man davon ausgehen kann, daß bei Ausführung der API-Funktionen keine ungewünschten Veränderungen der in den Objekten, Attributen und Links gespeicherten Informationen vorgenommen werden, muß man „nur“ dafür sorgen, daß der entsprechende Adreßraum vor und nach der Ausführung einer API-Funktion schreib- und/oder lesegeschützt wird. Solche Mechanismen werden aber innerhalb dieser Diplomarbeit nicht angeboten.

Der Systemadministrator hat dafür zu sorgen, daß ein Segment nicht zu groß wird. In einem solchen Fall wird entweder das nächste zu erzeugende ProSet-Objekt des P-Files auf einem neuen Segment angelegt oder ein zweites P-File erzeugt.

4.1.1 Das Home-Objekt

Damit jeder Benutzer prinzipiell auf seine privaten Datenobjekte arbeiten kann, wird bei einem Zugriff auf ein P-File bzw. Sub-P-File immer über ein speziell von H-PCTE zur Verfügung gestelltes *Home-Objekt*, ein Objekt vom Objekttyp `home`, navigiert. Der Objekttyp `home` wird von H-PCTE als Erweiterung zu PCTE angeboten, um dem Benutzer ein dem Home-Verzeichnis in UNIX ähnliches Konzept anzubieten. Ein Benutzer kann so ausgehend von seinem Home-Objekt seine privaten Datenobjekte speichern. Zu diesen Datenobjekten zählen nicht nur P-Files, Sub-P-Files und persistente ProSet-Werte, sondern auch jegliche andere Daten, die z.B. bei Benutzung eines Werkzeugs angefallen sind und gespeichert werden sollen. Ein solches Home-Objekt wird immer auf einem eigenen Segment angelegt, da dieses Segment dann später auch zur Ablage von Datenobjekten gebraucht werden kann, die nicht aufgrund der Nutzung der Persistenz von ProSet erzeugt wurden, sondern z.B. Datenobjekte einer SEU sind.. Bei der Navigation zu den Objekten, die persistente ProSet-

Werte speichern, wird immer vom Home-Objekt des Benutzers über den von H-PCTE eigens dazu angebotenen Pfadpräfix `"~/"` begonnen.

Damit alle Datenobjekte, die im Rahmen der Nutzung der Persistenz von ProSet anfallen, zentral gespeichert werden und man darüber hinaus auch im Rahmen eines Mehrbenutzerbetriebs auf P-Files und ihre Einträge von anderen Benutzern zugreifen kann, ist das Home-Objekt mit einem speziellen *ProSet-P-Store-Objekt* verbunden. Hierüber können die in der Objektbank von H-PCTE abgelegten P-Files, Sub-P-Files und persistenten ProSet-Werte aller Benutzer erreicht werden. Eine andere Vorgehensweise war hier nicht möglich, da sich alle Benutzer für ihre P-Files und Sub-P-Files den gleichen Namensraum teilen müssen (es sei denn, man legt eine Suchreihenfolge beim Zugriff auf die P-Files fest, deren Festlegung allerdings sehr problematisch ist).

4.1.2 Objektstruktur für P-Files und Sub-P-Files

Zur Unterstützung der *Private/Public-Deklaration von (Sub-)P-Files* wird ein P-File bzw. Sub-P-File (vorläufig) als ein komplexes H-PCTE-Objekt, im folgenden *P-File-* bzw. *Sub-P-File-Objekt* genannt, mit zwei Komponenten modelliert.

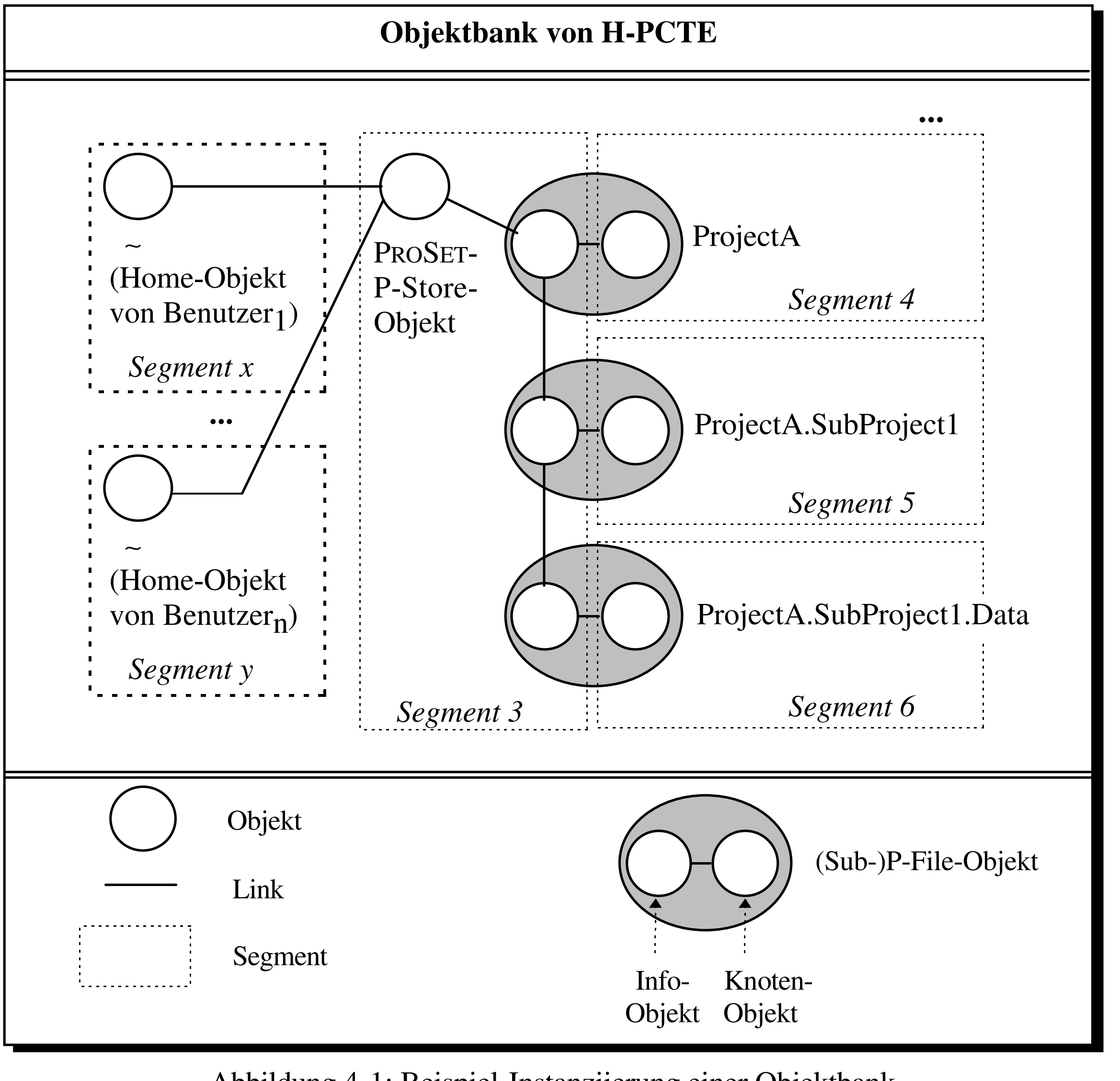


Abbildung 4-1: Beispiel-Instanziierung einer Objektbank

- Die erste Komponente besteht aus einem *Info-Objekt*, über das die hierarchische Verknüpfungsstruktur der P-Files und ihrer Sub-P-Files realisiert wird und das die Attribute `public` und `on_segments` sowie Attribute zur Realisierung eines Auditings verwaltet.
- Die zweite Komponente besteht aus einem *Knoten-Objekt*, welches in der Regel nur über das Info-Objekt erreichbar ist und von dem ausgehend Links auf alle im (Sub-)P-File enthaltenen PROSET-Objekte verweisen.

Die Info-Objekte der (Sub-)P-File-Objekte werden auf einem speziellen gemeinsamen Segment, im folgenden *PROSET-Administrationssegment* genannt, angelegt, wogegen über die Knoten-Objekte die Zuordnung eines (Sub-)P-Files zu einem eigenen Segment realisiert wird. Das PROSET-Administrationssegment nimmt eine Sonderstellung ein. Es darf nur global geladen werden, da es Informationen enthält, die jede Applikation benötigt, nämlich die Schachtelungsstruktur der (Sub-)P-Files und die `public`- und `on_segments`-Attribute der Info-Objekte. Mit Hilfe dieser Konvention kann nun jeder Applikationsprozeß unabhängig davon, ob ein anderer Applikationsprozeß das Segment eines P-Files lokal geladen hat, auf die Sub-P-Files des P-Files zugreifen.

4.1.3 Administrationsobjekte

Zur geeigneten Unterstützung einer verteilten Speicherung werden einige Administrationsobjekte benötigt, die entweder auf dem speziellen PROSET-Administrationssegment *(PAS)* oder auf dem *Master-Administrationssegment (MAS)* von H-PCTE gespeichert werden. Ein solches Administrationsobjekt ist das *Home-Objekt* eines Benutzers, das jeweils auf ein eigenes Segment angelegt wird und das PROSET P-Store-Objekt, das auf dem PAS gespeichert wird.

Das PAS enthält folgende H-PCTE-Objekte:

- Ein *Wurzel-Objekt* vom Objekttyp `persistent_root`, über das man alle P-Store-Objekte und somit auch alle P-File- bzw. Sub-P-File-Objekte und PROSET-Objekte von dem H-PCTE-Wurzel-Objekt (`"_"`) aus erreichen kann.
- Das *PROSET-P-Store-Objekt* vom Objekttyp `p_store` als Verzeichnis für die P-Files und Sub-P-Files eines jeden H-PCTE-Benutzers bzw. Benutzers der Persistenz von PROSET.
- die Info-Objekte aller in der Objektbank angelegten P-File- und Sub-P-File-Objekte,
- die Objekte, welche die von PROSET benötigten Standardbibliotheksfunktionen speichern, sowie alle anderen Daten, die prinzipiell von allen Benutzern benötigt werden.

Die Segment-ID des PAS wird in einem Attribut des P-Store-Objekts gespeichert, damit man unabhängig von der Installation von H-PCTE die Segment-ID des PAS bestimmen und das Info-Objekt eines neu anzulegenden (Sub-)P-File-Objekts dementsprechend auf das PAS anlegen kann.

Um der Autonomieanforderung von H-PCTE zu genügen, werden alle Administrationsobjekte von H-PCTE, die zur geeigneten Realisierung der Persistenz in PROSET benötigt werden und nicht über ein PROSET-Programm geändert werden können, auf dem MAS von H-PCTE gespeichert. Dies sind im wesentlichen alle Benutzer- und Benutzergruppenobjekte, sowie ein *temporäres Verzeichnis-Objekt*, das zur Unterstützung von Administrationswerkzeugen zur Benutzer- und Benutzergruppen-Verwaltung benötigt wird. Weitere Informationen hierzu werden später gegeben.

4.1.4 Zusammenfassung

- Ein P-File bzw. Sub-P-File wird durch ein komplexes H-PCTE-Objekt repräsentiert, das aus einem Info-Objekt und einem Knoten-Objekt besteht.
- Für jedes P-File und Sub-P-File wird ein eigenes Segment in H-PCTE angelegt, wobei immer das Info-Objekt auf dem PAS und das Knoten-Objekt auf dem (Sub-)P-File-Segment gespeichert wird. Die Segment-ID(s) werden innerhalb des `on_segments`-Attributs im Info-Objekt verwaltet. Die P-Files dürfen nur von einem Systemadministrator über ein Werkzeug angelegt werden dürfen. Diese Vorgehensweise geht konform mit H-PCTE-Regel (H4).
- Ein P-Store-Objekt dient als Container für alle P-Files und Sub-P-Files.
- Ein Home-Objekt dient als Verzeichnis für die privaten Daten eines Benutzers in H-PCTE, womit Regel (H3) unterstützt wird. Das Home-Objekt ist immer mit genau einem P-Store-Objekt verbunden. So kann man mit Hilfe von mehreren P-Store-Objekten ansatzweise eine Versionierung unterstützen. Dies wird in Abschnitt 4.4.2 ab Seite 67 noch genauer erläutert.
- Beim Zugriff auf einen persistenten PROSET-Wert über die Persistenzschnittstelle wird immer über das Home-Objekt des Benutzers navigiert.
- Gemäß dem Attributwert des Zusatzattributs `public` bei einem P-File- oder Sub-P-File-Objekt werden die Segmente eines (Sub-)P-Files global im Server bzw. lokal im Applikationsprozeß (Client) geladen. Diese Verfahrensweise unterstützt außerdem Regel (H2).
- Übersteigen die P-File-Einträge eine bestimmte Anzahl, sollte das Segment über ein Administrationswerkzeug gesplittet werden, so daß mehrere kleinere Segmente entstehen und somit Regel (H1) beachtet wird.
- Auf einem PAS werden alle Administrationsobjekte zur Realisierung der Persistenz von PROSET gespeichert. Die Segment-ID wird innerhalb eines Attributs im P-Store-Objekt gespeichert. Diese Verfahrensweise genügt Regel (H5). Das PAS darf außerdem nur global geladen werden.
- Die Benutzer- und Benutzergruppenobjekte werden auf dem Master-Administrationssegment gespeichert, um Regel (H5) zu unterstützen.
- Da ein Segment über einen UNIX-Pfad referenziert wird, muß letztendlich der Systemadministrator über entsprechende Verzeichnisse eines verteilten Dateisystems eine Verteilung auf verschiedenen Rechner realisieren.

4.2 Das Schutzkonzept von PROSET

Innerhalb der in Abschnitt 3.4 ab Seite 41 vorgenommenen Anforderungsanalyse bzgl. eines Schutzkonzepts für PROSET wurde bereits festgelegt, welche Subjekte man modellieren können muß (S3), welche Schutzobjekte (S6) mit welchen Rechten (S7) versehen werden müssen und welche zusätzlichen Schutzmechanismen wünschenswert sind, wie z.B. Funktionen zur Identifikation und Authentifikation (S4), sowie Auditingfunktionen (S12). Aus diesem Grunde wird im nächsten Abschnitt beschrieben, wie die PROSET-Zugriffsrechte mit Hilfe der von H-PCTE zur Verfügung gestellten Zugriffsmodi und Rechtewerte realisiert werden können. Außerdem werden die zur Realisierung des Schutzkonzepts notwendigen Objektstrukturen und der Entwurf der Auditingfunktionen vorgestellt.

4.2.1 Realisierung der PROSET-Zugriffsrechte

Im folgenden werden zuerst die PROSET-Zugriffsmodi und die Zugriffsmodi von H-PCTE vorgestellt, um dann eine geeignete Abbildung der PROSET Zugriffsrechte auf die von H-PCTE zur Verfügung gestellten Zugriffsmodi und ihrer Rechtewerte vorzunehmen.

4.2.1.1 Die PROSET-Zugriffsmodi

Gemäß der in Abschnitt 3.4.3 ab Seite 44 aufgestellten Anforderung (S7) werden die in Tabelle 4-1 präsentierten Zugriffsmodi in PROSET für einen Zugiff auf einen persistenten PROSET-Wert V (für Value) benötigt. Um auch als Randbedingung Zugriffe über ein Werkzeug zu beachten, werden diese Modi entsprechend erweitert. Analog zum bereits definierten (Sub-)P-File-Objekt sei im folgenden vorläufig ein *PROSET-Objekt* als ein H-PCTE-Objekt definiert, das einen persistenten PROSET-Wert speichert.

PROSET-Zugriffsmodus	Zugriffe
read[V]	Laden eines persistenten PROSET-Werts
write[V]	Speichern bzw. Verändern und Löschen eines persistenten PROSET-Werts

Tabelle 4-1: Zugriffsmodi auf einen persistenten PROSET-Wert

Darüber hinaus werden bei einem Zugriff auf einen P-File oder Sub-P-File P folgende Zugriffsmodi definiert:

PROSET-Zugriffsmodus	Zugriffe
read[P]	Auflisten der Einträge eines (Sub-)P-Files
write[P]	Erzeugen und Löschen eines (Sub-)P-File-Eintrags, Verändern und Löschen des (Sub-)P-File-Objekts
execute[P]	Zugriff auf (Sub-)P-File-Einträge per Navigation über das (Sub-)P-File-Objekt

Tabelle 4-2: Zugriffsmodi auf ein P-File- bzw. Sub-P-File

4.2.1.2 Die H-PCTE-Zugriffsmodi

H-PCTE faßt einige Zugriffsmodi von PCTE zu einem neuen Zugiffsmodus zusammen bzw. hat einige Zugriffsmodi von PCTE (noch) nicht berücksichtigt. Alle neu definierten Zugriffsmodi von H-PCTE sind in [See94a] erläutert, wogegen die PCTE-Zugriffsmodi in [ECM93b], Kapitel 19.1.3, ausführlich erklärt werden. Mit Hilfe dieser Zugriffsmodi und ihrer drei Rechtewerte „erlaubt“, „verboten“ und „undefiniert“, können nun die PROSET-Zugriffsmodi mit den gleichen Rechtewerten definiert werden.

4.2.1.3 Abbildung der PROSET-Zugriffsrechte auf die H-PCTE-Zugriffsrechte

Um eine geeignete Abbildung der PROSET-Zugriffsrechte auf die H-PCTE-Zugriffsrechte zu finden, werden zuerst die PROSET-Zugriffsmodi auf Basis der Zugriffsmodi von H-PCTE definiert (+ wird hier als Vereinigung der Mengen von Operationen interpretiert):

- Zugriffsmodi auf PROSET-Objekte:

  Da die in einem persistenten PROSET-Wert gespeicherten Informationen in Attribute von PROSET-Objekten oder ihren Links abgelegt werden, kann man über `READ_ATTRIBUTES` und `READ_LINK_ATTRIBUTES` einen Lesezugriff auf die Attribute kontrollieren.

  read[V] = `READ_ATTRIBUTES` + `READ_LINK_ATTRIBUTES`

  Ein Schreibzugriff auf die Attribute wird über die Zugriffsmodi `WRITE_ATTRIBUTES`, `WRITE_LINK_ATTRIBUTES`, `APPEND_LINK_ATTRIBUTES` und `APPEND_ATTRIBUTES` kontrolliert. `APPEND_LINKS` und `APPEND_IMPLICIT` wird benötigt, um die Komponenten eines komplexen H-PCTE-Objekts erzeugen sowie kopieren zu können. Über `DELETE`, `DELETE_IMPLICIT`, `DELETE_LINKS` und `CONTROL_OBJECT` wird das Löschen bzw. Verändern des PROSET-Objekts kontrolliert.

  write[V] = `WRITE_ATTRIBUTES` + `APPEND_ATTRIBUTES` + `WRITE_LINK_ATTRIBUTES` + `APPEND_LINK_ATTRIBUTES` + `DELETE_LINKS` + `DELETE_IMPLICIT` + `DELETE` + `APPEND_LINKS` + `APPEND_IMPLICIT` + `CONTROL_OBJECT`

  Man benötigt darüber hinaus sowohl beim Laden als auch beim Speichern eines persistenten PROSET-Werts die Zugriffsmodi `LIST_LINKS` und `NAVIGATE`. Denn nur über diese Modi wird garantiert, daß man jederzeit zu den Komponenten eines PROSET-Objekts navigieren bzw. Referenzen auf die Komponentenobjekte erzeugen kann. Aus Kompatibilität zu den UNIX-Rechten müssen diese Zugriffsmodi implizit immer auf erlaubt(+) gesetzt werden, sobald entweder ein read[V]-Recht oder ein write[V]-Recht auf einen persistenten PROSET-Wert gewährt wurde. Sind beide Rechte nicht gewährt, so werden die beiden Zugriffsmodi auf undefiniert(–) gesetzt. Aus Sicht von PROSET ist somit ein positives Rechtekonzept gemäß Anforderung (S7) jederzeit gesichert.

- Zugriffsmodi auf (Sub-)P-File-Objekte:

  Da ein P-File bzw. Sub-P-File und ein Verzeichnis in UNIX miteinander verglichen werden können, sollten soweit wie möglich auch die UNIX-Rechte für Verzeichnisse auf P-Files und Sub-P-Files übertragen werden (aus den in Abschnitt 3.4.3 auf Seite 44 erwähnten Gründen).

  Um den Inhalt eines P-Files auflisten zu können, muß man über `LIST_LINKS` alle vom P-File-Objekt ausgehenden Links auflisten können. Über `LIST_LINKS` kann man somit das Auflisten der Einträge eines (Sub-)P-Files kontrollieren.

  read[P] = `LIST_LINKS`

  Da bei einem Schreibzugriff auf ein (Sub-)P-File ein neuer Eintrag (PROSET-Objekt oder Sub-P-File-Objekt) angelegt oder ein bestehender gelöscht werden kann, wird über `APPEND_LINKS` und `APPEND_IMPLICIT` ein Erzeugen von P-File-Einträgen kontrolliert. Wogegen über `DELETE_LINKS`, `DELETE_IMPLICIT` das Löschen von

(Sub-)P-File-Einträgen kontrolliert werden kann. Außerdem wird über DELETE das Löschen des (Sub-)P-File-Objekts selbst kontrolliert. So kann ein Benutzer jederzeit den Speicherplatz von nicht mehr benötigten (Sub-)P-Files freigeben. Über den Zugriffsmodus CONTROL_OBJECT kann eine Veränderung des (Sub-)-P-File-Objekts durch Bewegen auf ein anderes Segment kontrolliert werden.

write[P] = APPEND_LINKS + APPEND_IMPLICIT + DELETE + DELETE_LINKS + DELETE_IMPLICIT + CONTROL_OBJECT

Über NAVIGATE kann in etwa das execute-Recht realisiert werden, da dieser Modus benötigt wird, um zu einem entsprechenden H-PCTE-Objekt überhaupt erst zu gelangen. Darüber hinaus muß man, um über ein ProSet-Programm auf den (Sub-)P-File-Einträgen Operationen ausführen zu können, das Segment eines (Sub-)P-Files laden können. Hierzu bietet H-PCTE keinen expliziten Zugriffsmodus an. Da jedoch innerhalb der Persistenzschnittstelle explizit eine Funktion zum Laden eines (Sub-)P-File-Segments aufgerufen wird und diese aus dem on_segments-Attribut die dazu benötigte Segment-ID ausliest, benötigt man einen Zugriffsmodus READ_ATTRIBUTES.

execute[P] = NAVIGATE + READ_ATTRIBUTES

Wie bereits erwähnt werden die Rechtewerte von H-PCTE in ProSet übernommen, wobei eventuell spezielle Werkzeuge nur die in UNIX bekannte zweiwertige Rechtelogik anbieten kann. In einem solchen Fall dürfen nur die Rechtewerte erlaubt und undefiniert benutzt werden.

ProSet-Rechtewert	**H-PCTE-Rechtewert**
erlaubt (+)	erlaubt (+)
undefiniert (?)	undefiniert (?)
verboten (–)	verboten (–)

Tabelle 4-3: Abbildung der ProSet-Rechtewerte auf die Rechtewerte von H-PCTE

Wird ein Rechtewert für einen ProSet-Zugriffsmodus gesetzt, so werden implizit alle die zur Realisierung dieses Zugriffsmodus benötigt HPCTE-Zugriffsmodi entsprechend gesetzt. Ein ProSet-Zugriffsrecht wird dann wie in Abschnitt 2.3.2.6 ab Seite 19 beschrieben aus diesen Angaben abgeleitet. Um der Anforderung (S7) über ein positives Rechtekonzept zu genügen, werden die Zugriffsmodi von H-PCTE, die für die Abbildung der ProSet-Zugriffsmodi hier nicht gebraucht wurden, implizit immer als für das Subjekt undefiniert gesetzt. Eine Ausnahme bildet hier lediglich der CONTROL_DISCRETIONARY-Zugriffsmodus, der benötigt wird, um die ACL eines Objekts verändern zu dürfen. Auf diesen Aspekt wird in einem späteren Abschnitt noch gesondert Bezug genommen.

4.2.2 Objektstruktur zur Realisierung des Schutzkonzepts

Gemäß der Anforderung (S1) von Abschnitt 3.4.3 ab Seite 44 darf ein Benutzer bzw. ein ProSet-Programm oder Werkzeug nur über Funktionen der Persistenzschnittstelle auf persistente ProSet-Werte zugreifen. Darüber hinaus wird das Konzept aber möglichst so entworfen, daß der Schutz der Schutzobjekte auch bei einem direkten Zugriff über die API von H-PCTE gewahrt bleibt. Deshalb sollten die Zugriffsrechte auf ein komplexes Objekt immer in der ACL eines jeden Komponentenobjekts gespeichert werden. Aus diesem Grund werden die Zugriffsrechte auf ein (Sub-)P-File in der ACL des Info-Objekts und des Knoten-Objekts gespeichert. Es sei an dieser Stelle darauf hingewiesen, daß ein verwehrtes Zugriffsrecht auf ein (Sub-)P-File nicht als sicher gelten kann, da bei einem Zugriff über Funktionen der API von H-PCTE und bei entsprechenden Kenntnissen der internen Objektbankstruktur ein Zugriff auf P-File-Einträge über Verwaltungsobjekte von H-PCTE möglich ist.

Die Verwaltung der Zugriffsrechte auf einen persistenten ProSet-Wert in der ACL von allen Komponentenobjekten bereitet jedoch ein Problem: In der schwachgetypten Sprache ProSet kann ein ProSet-Wert seinen Typ ändern. Dies verursacht in H-PCTE insofern Probleme, als daß ein Objekt nicht ohne weiteres seinen Objekttyp ändern kann, es sei denn, er steht in einer Ableitungsbeziehung zum neuen Objekttyp. Eine solche Konvention wäre allerdings keine geeignete Modellierung für die persistenten ProSet-Werte. Somit kann man nur bei einem Wechsel des Typs des ProSet-Werts das ihn repräsentierende H-PCTE-Objekt löschen und ein H-PCTE-Objekt vom entsprechenden neuen Typ neu erzeugen. Diese Vorgehensweise verursacht innerhalb eines Schutzkonzeptes aber Nachteile, da wie oben beschrieben jedes Objekt seine Zugriffsrechte selbst innerhalb seiner ACL verwaltet. Die Informationen innerhalb der ACL gehen somit bei einem Löschvorgang des Objekts verloren, und das Objekt mit dem neuen Typ enthält keine der ACL-Einträge seines Vorgängers mehr. Damit nun die ACL während eines solchen Löschvorgangs nicht verloren geht, wird ein persistenter ProSet-Wert über ein komplexes H-PCTE-Objekt modelliert. Dieses Objekt wird im folgenden als *ProSet-Objekt* bezeichnet und besteht auf einem höheren Abstraktionsniveau aus zwei Komponenten (siehe nachfolgende Abbildung 4-2):

1. Einem *Info-Objekt*, welches die Zugriffsrechte in seiner ACL sowie Auditinginformationen über das ProSet-Objekt innerhalb spezieller Attribute verwaltet. Dieses Objekt wird solange der entsprechende persistente ProSet-Wert existiert nicht gelöscht.
2. Einem *Werte-Objekt,* in dem der Wert des ProSet-Objekts gespeichert wird und anhand dessen Objekttyp man den Typ des persistenten ProSet-Werts ermitteln kann.

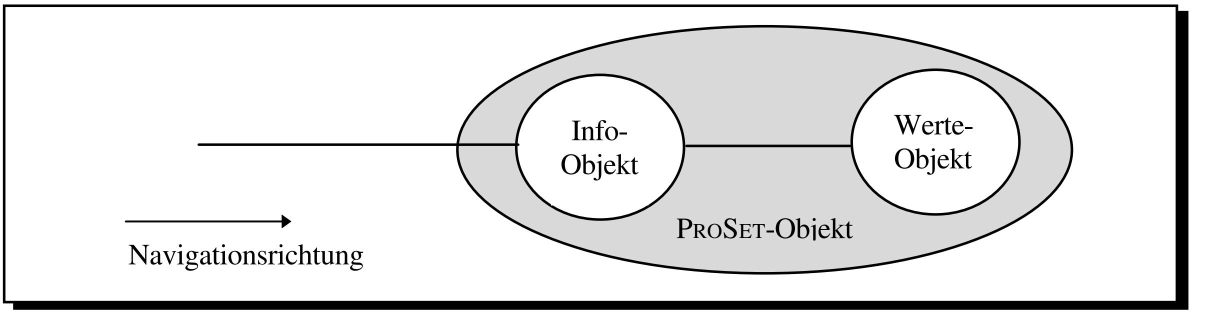


Abbildung 4-2: Aufbau eines ProSet-Objekts

Bei einem Typwechsel eines persistenten ProSet-Werts, wird nun nur das Werte-Objekt des ProSet-Objekts gelöscht und bei einem Neuanlegen mit neuem Objekttyp anhand der ACL seines Info-Objekts die eigene ACL wiederhergestellt.

In ProSet sind P-Files, Sub-P-Files und persistente ProSet-Werte Schutzobjekte. Zur Verdeutlichung: persistente ProSet-Werte sind Werte, die innerhalb eines Programms als persistent deklariert wurden und nicht ProSet-Werte, die nur mitabgespeichert werden müssen, da sie z.B. in der eingefrorenen Umgebung eines persistenten ProSet-Werts vorkommen oder ein Element eines persistenten ProSet-Werts vom Typ `tuple` oder `set` repräsentieren. Solche ProSet-Werte werden im folgenden *interne ProSet-Werte* bzw. die sie repräsentierenden H-PCTE-Objekte *interne ProSet-Objekte* genannt. Zur Speicherung dieser internen PROSET-Objekte benötigt man kein Info-Objekt, sondern nur das Werte-Objekt, da ihre Zugriffsrechte über das PROSET-Objekt verwaltet werden, dessen Komponenten sie sind.

4.2.3 Subjekte als Besitzer

Jeder Benutzer, der ein PROSET-Objekt innerhalb seines PROSET-Programms erzeugt, ist automatisch der Besitzer dieses Objekts. Desweiteren sollten gemäß Anforderung (S9) mehrere Systemadministratoren angemeldet werden können. Aus diesem Grunde wird für jede H-PCTE-Installation eine Benutzergruppe `systemadministrator` kreiert, in der die Gruppen-Administratoren eingetragen werden und die bei Ausführung von Systemadministrator-Funktionen von den entsprechenden Benutzern aktiviert werden muß. So muß man nur für den Besitzer eines PROSET-Objekts bzw. (Sub-)P-File-Objekts und für das Subjekt in Form der Benutzergruppe `systemadministrator` den Zugriffsmodus `CONTROL_DISCRETIONARY` auf erlaubt(+) setzten. Jedem anderen Subjekt wird dieser Zugriffsmodus durch Setzen des Rechtewerts auf undefiniert(?) verboten.

4.2.4 Festlegen von Zugriffsrechten

Die Festlegung der Zugriffsrechte erfolgt implizit bei der Erzeugung eines persistenten PROSET-Werts oder explizit über Operationen der Persistenzschnittstelle.

Um die Zugriffsrechte eines neu erzeugten persistenten PROSET-Werts zu setzen, wird aus einer Umgebungsvariablen `PS_ACCESS_MASK` die Zugriffsrechtemaske für den Benutzer ausgelesen, auf deren Basis die ACL des PROSET-Objekts gesetzt wird. Dieser Mechanismus kann mit dem `umask`-Befehl in UNIX verglichen werden. Über die Zugriffsrechtemaske gibt man allerdings explizit die Rechtewerte für alle PROSET-Zugriffsmodi an, wobei sich die so spezifizierten Zugriffsrechte auf das Benutzer-Subjekt und das zur Zeit aktivierte Benutzergruppen-Subjekt beziehen. Hiermit ist immer die implizit beim Start eines Prozesses oder explizit durch Ausführung einer entsprechenden API-Funktion aktivierte Benutzergruppe gemeint und nicht die dabei ebenfalls aktivierten Obergruppen. In `PS_ACCESS_MASK` muß somit eine aus 4 Zeichen bestehende Zeichenkette übergeben werden, wobei jedes Zeichen einen Rechtewert "+","?" oder "–" darstellt und die Zeichenkette nach folgendem Muster interpretiert wird:

1. Zeichen	2. Zeichen	3. Zeichen	4. Zeichen
Rechtewert für Zugriffsmodus read[V] für den Benutzer	Rechtewert für Zugriffsmodus write[V] für den Benutzer	Rechtewert für Zugriffsmodus read[V] für die aktivierte Benutzergruppe	Rechtewert für Zugriffsmodus write[V] für die aktivierte Benutzergruppe

Tabelle 4-4: Über eine Zeichenkette spezifizierte Zugriffsrechtemaske eines Benutzers

Die ACL eines neu erzeugten PROSET-Objekts wird durch eine bestimmte Verknüpfung einer *Prozeß-Default-ACL* und einer *Default-Rechtemaske* gesetzt (siehe [ECM93b], Kapitel 19.1.4). Die Default-Rechtemaske wird immer implizit so gesetzt, daß sie bei der Verknüpfung keinen Einfluß auf die Prozeß-Default-ACL hat. Deshalb enthält die Prozeß-Default-ACL eines PROSET-Programms immer mindestens drei Einträge: einen für die Benutzergruppe `systemadministrator`, wobei für diese Gruppe alle Zugriffsmodi gesetzt werden, einen für den Besitzer, d.h. den aktiven Benutzer des Programms, und einen für die gerade aktivierte Benutzergruppe. Die beiden letzten Einträge der Prozeß-Default-ACL werden dabei über die in der Umgebungsvariablen gesetzten ProSet-Zugriffsrechte ermittelt. Ist diese Umgebungsvariable nicht definiert, werden die Einträge so gesetzt, daß die beiden Subjekte alle Rechte besitzen.

Darüber hinaus werden Operationen zur Festlegung von Zugriffsrechten angeboten, die später auch einmal über ProSet-Module innerhalb eines ProSet-Programms benutzt werden sollen. Entsprechende Funktionen werden auch für Werkzeuge angeboten. Diese Operationen umfassen:

- Ändern der Zugriffsrechtemaske.
- Setzen von Zugriffsrechten für den Benutzer oder eine Benutzergruppe auf persistente ProSet-Werte.
- Setzen von Zugriffsrechten für den Benutzer oder eine Benutzergruppe auf P-Files oder Sub-P-Files.
- Prüfen eines Zugriffsrechts auf ein (Sub-)P-File oder persistenten ProSet-Wert.

4.2.5 Aktivieren von Subjekten in ProSet

Bei einem Start eines ProSet-Applikationsprozesses, der die Persistenz benutzt, wird automatisch, wie in Abschnitt 2.3.2.5 auf Seite 18 beschrieben, der gemäß der User-ID des UNIX-Prozesses spezifizierte Benutzer als Subjekt aktiviert (genauer bezeichnet, wird die den Benutzer repräsentierende Benutzergruppe aktiviert). Darüber hinaus wird immer implizit die als seine Default-Gruppe in H-PCTE eingetragene Benutzergruppe aktiviert. Damit jeder Benutzer unter den Benutzergruppen, in denen er als Mitglied eingetragen ist, in H-PCTE seine Default-Gruppe eintragen kann, sollte ein entsprechendes Dienstprogramm zur Verfügung gestellt werden. Ein solches Dienstprogramm kann mit dem in UNIX angebotenen `chgrp`-Dienstprogramm verglichen werden. Solange ein solches Werkzeug nicht existiert, kann der Benutzer über eine Umgebungsvariable `PS_ADOPTED_GROUP` seine während des Applikationsprozesses zu adoptierende Benutzergruppe spezifizieren. Ist diese Umgebungsvariable nicht gesetzt, wird automatisch die in H-PCTE für diesen Benutzer definierte Defaultgruppe aktiviert.

Zur Laufzeit sollte man innerhalb eines Programms die Benutzergruppe wechseln können, d.h. anstatt der gerade aktiven Benutzergruppe und all ihrer somit aktiven Obergruppen wird eine andere Benutzergruppe aktiviert und somit all ihre Obergruppen. Aus diesem Grunde sollte die Persistenzschnittstelle eine Funktion, vergleichbar mit dem Dienstprogramm `chgrp` unter UNIX, anbieten, die ein ProSet-Programm dann später über ein ProSet-Modul nutzen kann.

4.2.6 Zugriffsrechte auf Administrationsobjekte

Damit ein Mehrbenutzerbetrieb realisiert werden kann, benötigt jeder Benutzer gewisse Zugriffsrechte auf den Administrationsobjekten.

Damit ein Benutzer über sein Home-Objekt navigieren kann und außerdem private Daten innerhalb von Objekten und Links ausgehend von diesem Objekt speichern kann, müssen folgende H-PCTE-Zugriffsmodi für diesen Benutzer auf erlaubt(+) gesetzt werden:

NAVIGATE, APPEND_LINKS, APPEND_IMPLICIT, WRITE_LINK_ATTRIBUTES, LIST_LINKS, READ_LINK_ATTRIBUTES, APPEND_LINK_ATTRIBUTES, DELETE_LINKS, DELETE_IMPLICIT

Um dem Benutzer ausgehend von einem Home-Objekt zu ermöglichen, über das P-Store-Objekt auf sein P-File-Objekt zu navigieren, muß der NAVIGATE Zugriffsmodus auf das P-Store-Objekt für ihn auf erlaubt (+) gesetzt werden.

Damit ein Benutzer die Zugriffsrechte auf ein PROSET-Objekt oder ein (Sub-)P-File-Objekt verändern kann, muß er die ID des Benutzer- bzw. Benutzergruppenobjekts ermitteln können. Hierzu muß es ihm erlaubt sein, auf sein Benutzer- bzw. das Benutzergruppenobjekt zu navigieren. Aus diesem Grunde muß für jeden Benutzer angefangen vom H-PCTE-Wurzelobjekt "_" über das Gruppenverzeichnisobjekt bis zum entsprechenden Benutzer- bzw. Benutzergruppenobjekt die Zugriffsmodi NAVIGATE, READ_LINK_ATTRIBUTES und LIST_LINKS auf erlaubt(+) gesetzt sein.

Die benötigten Zugriffsrechte auf den Administrationsobjekten werden implizit bei Ausführung einer entsprechenden Funktion der Persistenzschnittstelle ausgeführt. Alle in diesem Zusammenhang nicht benötigten Zugriffsmodi werden für den Benutzer auf undefiniert(–) gesetzt.

Die Benutzergruppe systemadministrator besitzt immer alle Zugriffsrechte auf jeglichen Administrationsobjekten.

4.2.7 Auditingfunktionen

Gemäß Anforderung (S12) wird für PROSET ein Auditing-Schutzmechanismus angeboten. H-PCTE stellt zwar Standardattribute für ein Auditing über den Basisobjekttyp object zur Verfügung, jedoch sind diese nur read-only und können über Funktionen der API von H-PCTE nicht verändert werden. Aus diesem Grund wird ein Objekttyp audit_object eingeführt, dessen Instanzen Attribute zur Verfügung stellen, über denen ein Auditing realisiert werden kann. Von dem Objekttyp audit_object werden dann die Objekttypen der Schutzobjekte abgeleitet und somit mit Auditingattributen versehen. Die zum Auditing notwendigen Informationen werden über zwei Attribute verwaltet: Ein Attribut change_time vom Typ time, in welches gespeichert wird wann zuletzt auf ein Schutzobjekt schreibend zugegriffen wurde. Ein Attribut changed_by vom Typ string, das den Namen des Benutzers verwaltet, der den letzten Zugriff auf das Schutzobjekt ausgeführt hat.

Ein Problem stellt die Sicherheit des Auditings an sich dar: Über das Auditing sollen schließlich sicherheitsrelevante Ereignisse protokolliert werden. Dies setzt jedoch voraus, daß ein Benutzer nicht unbefugt die Auditinginformationen des Schutzobjekts verändern kann. Eine solche Veränderung ist allerdings immer dann möglich, wenn ein Benutzer write-Zugriffsrechte auf ein Schutzobjekt besitzt. Da PCTE jedoch über ein *Notifikationskonzept* [ECM93a], Kapitel 15, im Zusammenspiel mit *message queues* [ECM93a], Kapitel 14, ermöglicht, Operationsausführungen als Ereignisse zu spezifizieren, wobei nach einer solchen Operationsausführung Notifikationsnachrichten versendet und abgefangen werden können, kann man hier die H-PCTE-Objekte, die Auditingattribute besitzen, überwachen lassen. Zur Zeit (Version 2.6) ist jedoch diese Funktionalität von PCTE in H-PCTE noch nicht implementiert. Hier wäre außerdem ein Werkzeug wünschenswert, über das ein Administrator ein Auditing ein- und abschalten kann.

4.2.8 Administrationsfunktionen zur Unterstützung des Schutzkonzepts

Wie schon erwähnt, werden zur Unterstützung des Schutzkonzepts Werkzeuge für eine Benutzer- und Benutzergruppenverwaltung sowie Werkzeuge zum Anzeigen und Setzen von PROSET-Zugriffsrechten benötigt.

Im Rahmen einer Benutzer- und Benutzergruppenverwaltung werden folgende Administrationsfunktionen angeboten:

- Anmelden (im Sinne von „dem System bekanntmachen“) eines neuen Benutzers,
- Löschen eines Benutzers,
- Anlegen einer neuen Benutzergruppe,
- Löschen einer Benutzergruppe,
- Eintragen eines Benutzers in eine Benutzergruppe,
- Austragen eines Benutzers aus einer Benutzergruppe,
- Eintragen einer Benutzergruppe als Untergruppe in einer anderen Benutzergruppe,
- Austragen einer Benutzergruppe als Untergruppe aus einer anderen Benutzergruppe.

4.3 Das Transaktionskonzept von PROSET

Laut Anforderung (T1) soll das Transaktionskonzept von PROSET geschachtelte Transaktionen vorsehen. Innerhalb eines geschachtelten Transaktionsmodells müssen hierbei die ACID-Eigenschaften der Transaktionen zu jedem Zeitpunkt gesichert werden. Da H-PCTE zur Zeit nur ein flaches Prozeßmodell anbietet, hat man keine Möglichkeit, die Wurzel- bzw. Subtransaktionen auf eigenständige Prozesse zu verteilen, um somit die Wurzel- bzw. Subtransaktionen einer Ebene des Schachtelungsbaums eventuell – bei geeigneter Unterstützung durch die unterliegende Hardware – parallel abarbeiten zu können.

Die innerhalb einer Programmeinheit auf persistenten PROSET-Werten ausgeführten Operationen werden entweder innerhalb einer (Wurzel-)Transaktion oder einer Subtransaktion ausgeführt. Dabei wird jeweils zu Beginn einer Programmeinheit, in der persistente PROSET-Werte deklariert sind, eine Transaktion bzw. Subtransaktion gestartet. Entsprechend wird bei Beendigung dieser Programmeinheit die Transaktion bei erfolgreicher Ausführung beendet (Commit) bzw. bei fehlerhafter Ausführung abgebrochen (Abort). Die Wurzeltransaktionen sollen dabei die ACID-Eigenschaften besitzen, wogegen für die Subtransaktionen die Atomaritäts- und Isolations-Eigenschaft gesichert wird. Solche geschachtelten Transaktionen genügen den Mindestanforderungen, die PROSET an Transaktionen stellt. Eine Änderung dieser Eigenschaftsfestlegungen und somit des Transaktionsmodells, wie in [Fra95] beschrieben, kann man nur durch Konstruktion eines separaten Transaktionsmanagers vornehmen, da man hierzu ein geschachteltes Prozeßmodell benötigt. Eine solche Realisierung würde jedoch weit über den Rahmen dieser Diplomarbeit hinausgehen und wird deshalb nicht vorgenommen.

Über die von H-PCTE über eine Aktivitätsklasse angebotenen (flachen) Transaktionen können die Wurzeltransaktionen modelliert werden. Diese weisen bereits die geforderten ACID-Eigenschaften auf. Die Subtransaktionen können mit Hilfe der zur Verfügung gestellten Sicherungspunkte realisiert werden: Beim Start einer Subtransaktion wird intern ein

Sicherungspunkt gesetzt, auf den man beim Abbruch dieser Subtransaktion zurücksetzen kann. Die Atomaritäts-Eigenschaft der Subtransaktion wird somit gesichert. Um die Isolations-Eigenschaft zu bewahren, müssen mit Hilfe der Concurrency Control-Komponente von H-PCTE die in (T2) geforderten Sperren gesetzt werden. Auch dies übernimmt H-PCTE über das bei Transaktionen vorgenommene Synchronisationsprotokoll implizit (siehe Abschnitt 2.3.2.4, Seite 17).

Zur Unterscheidung der in einem ProSet-Programm gestarteten Transaktionen und der über H-PCTE-Aktivitäten angebotenen Transaktionen werden im folgenden entsprechende Transaktionen *ProSet-Transaktionen* und *H-PCTE-Transaktionen* genannt.

4.3.1 Einbettung der Transaktionen in ProSet

In ProSet wird zu Beginn einer Programmeinheit bei der Abarbeitung einer Persistenzdeklaration die entsprechenden persistenten ProSet-Werte geladen. Bei Beendigung der Programmeinheit werden diese wieder in die Objektbank von H-PCTE gespeichert. Dabei wird beim Ladevorgang immer eine Kopie eines persistenten ProSet-Werts auf Basis der aus der Objektbank ausgelesenen Informationen angelegt und innerhalb des Programms auf dieser Kopie gearbeitet. Dementsprechend wird auf Basis der Kopie beim Speichern das ProSet-Objekt in der Objektbank geändert. Die innerhalb einer Programmeinheit ausgeführten Operationen auf persistente ProSet-Werte werden durch eine ProSet-Transaktion geschützt.

4.3.2 Sperren

Nachfolgend wird vorgestellt, welche Sperren bei welchen Operationen und aus welchen Gründen innerhalb von ProSet-Transaktionen gesetzt werden:

- **Laden eines als `constant persistent` deklarierten ProSet-Werts:**

  Will ein Benutzer über ein ProSet-Programm einen persistenten ProSet-Wert laden, auf den er nur lesend zugreifen möchte, so zeigt er dies durch eine `constant persistent`-Deklaration an. In diesem Fall muß dieser persistente ProSet-Wert vor ProSet-Programmen geschützt werden, die diesen Wert laden, um ihn zu verändern. Auf H-PCTE-Ebene darf also der Inhalt eines ProSet-Objekts nicht von einem anderen Prozeß verändert oder gelöscht werden, da sonst eine Parallelitätsanomalie auftreten kann. Alle Funktionen von H-PCTE, die beim Laden eines persistenten ProSet-Werts ausgeführt werden, gehören zur Klasse der Leseoperationen (siehe [Lin91]). Da eine ProSet-Transaktion darüber hinaus mit Hilfe einer H-PCTE-Transaktion realisiert wird, werden die zu einem ProSet-Objekt gehörenden Ressourcen per `READ_PROTECTED` gesperrt. Somit kann nur eine in einem anderen Applikationsprozeß gestartete Aktivität eine weitere `READ_PROTECTED`-, `READ_UNPROTECTED`- und `READ_SEMIPROTECTED`-Sperre setzen. Eine Sperre, die bei Ausführung einer Ladeoperation gesetzt wird, soll im folgenden *Lesesperre* genannt werden.

- **Laden und Speichern eines als `persistent` deklarierten ProSet-Werts:**

  Will ein Benutzer über ein ProSet-Programm einen persistenten ProSet-Wert laden, auf den er lesend und schreibend zugreifen möchte, so zeigt er dies durch eine `persistent`-Deklaration an. In diesem Fall wird der ProSet-Wert zu Beginn der Ausführung seiner Programmeinheit geladen und bei Beendigung derselben gespeichert. Deshalb muß dieser persistente ProSet-Wert vor ProSet-Programmen geschützt werden, die diesen Wert laden, um auf ihn lesend oder schreibend zuzugreifen. Auf H-PCTE-Ebene darf der

Inhalt eines ProSet-Objekts also nicht von einem anderen Prozeß gelesen, verändert oder gelöscht werden, da sonst Inkonsistenzen auftreten können. Beim Speichern werden mehrere H-PCTE-Funktionen ausgeführt, die zur Klasse der Lese-, sowie der Schreib- und Löschoperationen gehören, so daß implizit in einer H-PCTE-Transaktion immer versucht wird, die zu einem ProSet-Objekt gehörenden Ressourcen per `READ_PROTECTED`, `WRITE_TRANSACTIONED` oder `DELETE_TRANSACTIONED` zu sperren. Da ein ProSet-Programm immer auf einer Kopie des persistenten ProSet-Werts arbeitet, genügt es, daß man die beiden zuletzt aufgezählten Sperren erst beim Speichern dieser Kopie setzt. Wie bereits erläutert, genügt das Synchronisationsprotokoll von H-PCTE-Transaktionen somit den obigen Anforderungen an das Setzen von Sperren. Eine Sperre, die bei Ausführung einer Speicheroperation gesetzt wird, soll im folgenden *Schreibsperre* genannt werden.

Die ProSet-Transaktionen stützen sich somit auf die von H-PCTE implizit gesetzten Sperren. Kann eine Sperre aufgrund eines Sperrkonflikts nicht gesetzt werden (siehe [ECM93a], Kapitel 16), wird die entsprechende ProSet-Transaktion in einen Wartezustand versetzt. Darüber hinaus kann jedoch eine ProSet-Transaktion in verschiedene Konfliktsituationen geraten, die im folgenden Abschnitt diskutiert werden.

4.3.3 Deadlocks und Livelocks

In diesem Abschnitt werden gemäß Anforderung (T4) Strategien zur Behandlung von Deadlocks und Livelocks diskutiert und eine geeignete Strategie aufgestellt.

Deadlocks treten häufig bei Umwandlungen von Sperren auf [Wei88], z.B. bei Umwandlung einer Lesesperre in eine Schreibsperre. Dies würde motivieren, daß man bei einer `persistent`-Deklaration eines ProSet-Werts bereits beim Laden des Werts eine Schreibsperre setzt, da sonst bei Ausführung einer ProSet-Transaktion eine Lesesperre in eine Schreibsperre umgewandelt wird. Diese Vorgehensweise würde jedoch die parallele Abarbeitung von ProSet-Transaktionen verschiedener ProSet-Programme zu stark einschränken. Aus diesem Grund werden erst beim entsprechenden Zugriff, also beim Laden oder Speichern, die dazugehörigen Sperren auf die dabei benutzten H-PCTE-Ressourcen gesetzt.

Das Auftreten von Deadlocks kann man entweder über eine geeignete Verhütungsstrategie verhindern oder über geeignete Strategien die aufgetretenen Deadlocks entdecken und beseitigen. Bei ProSet-Transaktionen kann man keine Deadlock-Verhütungsstrategie anwenden, da man erstens eine zentrale Transaktionsverwaltung benötigen würde, welche die Objektbankzugriffe aller gestarteten ProSet-Programme unterhalten und koordinieren müßte. Zweitens müßte diese Transaktionsverwaltung der entsprechenden Komponente von H-PCTE aufgesetzt werden, ohne daß hier Konflikte entstünden. Eine solche Realisierung außerhalb von H-PCTE würde allerdings den Rahmen dieser Diplomarbeit sprengen, so daß dieser Ansatz hier nicht weiter verfolgt wird. Darüber hinaus leistet H-PCTE bereits eine Deadlock-Erkennung, indem die deadlockverursachende API-Funktion mit einem Fehlercode abgebrochen wird, so daß nur noch eine geeignete Strategie zur Deadlock-Beseitigung aufgestellt werden muß. Natürlich könnte man hier die gesamte geschachtelte ProSet-Transaktion abbrechen, deren Sperrversuch einen Deadlock verursachte. Allerdings ist dies bei geschachtelten Transaktionen nicht wünschenswert. Gerade über die geschachtelten Transaktionen möchte man Arbeitseinheiten modellieren, so daß die verrichtete Arbeit nur teilweise rückgängig gemacht werden muß. Da bei jeder Sperranforderung der Sperrenanforderungsgraph von H-PCTE auf Zyklen untersucht wird [Lin91], ist es möglich, nur durch ein Rücksetzen der Subtransaktion (inklusive seiner Sperren), deren Sperranforderung einen Deadlock verursacht hat, die Konfliktsituation aufzulösen. Es sollte zwar immer diejenige an einem Deadlock beteiligte Transaktion zurückgesetzt werden, welche die geringsten Rücksetzkosten verursacht [Gr78], aber H-PCTE gibt nur für die Deadlock-verursachende Transaktion einen Fehlercode zurück, so daß man auch nur diese zurücksetzen kann. Zur Unterstützung der Orthogonalität der Konzepte (P3) wird hier bei einem Deadlock eine `escape`-Ausnahme `p_deadlock_occured` erzeugt.

Wie bereits in Abschnitt 3.3.2 auf Seite 39 erläutert, kann außerdem der Livelock einen Konflikt verursachen. H-PCTE verhindert laut [Lin91] einen Livelock durch die interne Bedingung, daß eine Sperre, die gesetzt werden soll, mit allen noch nicht zugeteilten Sperren, die an die Ressource für darauf wartende Prozesse gesetzt werden sollen, verträglich sein müssen.

4.3.4 Ausnahmen innerhalb von Transaktionen

Wird eine Ausnahme innerhalb einer Transaktion generiert, erhalten die im Rahmen der Ausnahmebehandlung angebotenen Operationen zur Beendigung eines Handlers eine erweiterte Bedeutung: Entweder kann der Benutzer über `return` die Transaktion abbrechen oder er sorgt per `return commit` dafür, daß die Transaktion beendet wird und somit die bis zum Zeitpunkt der Ausnahmegenerierung vorgenommenen Änderungen an persistenten ProSet-Werten in der Objektbank von H-PCTE gespeichert werden.

4.3.5 Recovery in ProSet

In Abschnitt 3.3.2.2 ab Seite 40 wurden Anforderungen an ein transaktionsorientiertes Recovery von ProSet formuliert, die hier nun berücksichtigt werden sollen. Die Anforderung (R1) wird über das Zurücksetzen von geschachtelten Transaktionen im Fehlerfall abgedeckt. H-PCTE deckt über sein Vorwärts- und Rückwärts-Recovery die Anforderungen (R2) und teilweise (R3) ab. Um den Anforderungen (R3) und (R4) vollständig, im Sinne der Fehlertoleranzansprüche einer Prototyping-Sprache, zu genügen, soll hier auf Basis der Backups, die ein UNIX-Systemadministrator täglich über Nacht vornimmt, ein solches Recovery geleistet werden.

4.3.6 Zusammenfassung

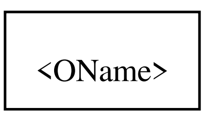
- Es werden mit Hilfe der von H-PCTE angebotenen Transaktionen und Sicherungspunkte geschachtelte Transaktionen realisiert, wobei Lesesperren beim Laden eines persistenten ProSet-Werts und Schreibsperren beim Speichern eines persistenten ProSet-Werts gesetzt werden.
- Die Transaktionen werden für den Benutzer transparent angeboten, d.h. die Transaktionen werden implizit gestartet, beendet und abgebrochen. Der Benutzer kann jedoch über die Ausnahmebehandlung von ProSet in Fehlersituationen selbst entscheiden, ob die Änderungen gesichert werden sollen oder nicht.

4.4 Das Datenbankschema zur Verwaltung persistenter PROSET-Werte

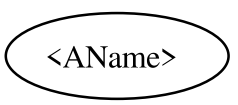
Nachfolgend wird für P-Files und Sub-P-Files, für persistente PROSET-Werte sowie für die zur Realisierung der Persistenzkonzepte benötigten Objekte ein Datenbankschema in H-PCTE aufgestellt, das über ein SDS mit Namen `proset` beschrieben wird (siehe Anhang A). Der Entwurf des Datenbankschemas wird in überschaubaren Teilen mit Hilfe eines auf H-PCTE abgestimmten Schema-Diagramms präsentiert.

4.4.1 Notation des verwendeten Schema-Diagramms

Da das Datenmodell von H-PCTE mit dem *binären Entity-Relationship-Modell* von Chen [Che76] verglichen werden kann, hat sich in der Literatur zu PCTE (z.B. [WJ93]) eine graphische Beschreibungsform über ein Schema-Diagramm durchgesetzt, das mit einem *Entity-Relationship-Diagramm* verglichen werden kann. Im folgenden werden die Datenbankschemata zur Verwaltung persistenter PROSET-Werte über Schema-Diagramme beschrieben. Hierzu wird auf Basis von[WJ93], Anhang B.1, ein auf das Datenmodell von H-PCTE abgestimmtes Schema-Diagramm mit folgenden Notationen benutzt:



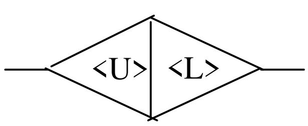
Objekttyp mit Namen <OName>



: <Typp>

Attribut eines Objekttyps mit Attributnamen <AName> und Attributtyp <Typ>

['('<LSName>**:<LSTyp>**')']*.<LBName>



Beziehungstyp

['('<RSName>**:<RSTyp>**')']*.<RBName>

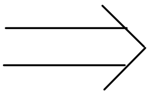
Da ein Beziehungstyp in H-PCTE über Linktyp und Umkehrlinktyp beschrieben wird, werden zusätzliche Angaben gemacht, wobei je nach Leserichtung Link- und Umkehrlinktyp wechseln. Die obigen Angaben werden hier von links nach rechts interpretiert:

- der Name des Linktyps <LBName> und optional eine Folge von Tupeln bestehend aus dem Namen <LSName> des Schlüsselattributs und seinem Attributtyp **<LSTyp>**;
- der Name des Umkehrlinktyps <RBName> und optional eine Folge von Tupeln bestehend aus dem Namen <LSName> des Schlüsselattributs und seinem Attributtyp **<LSTyp>**;
- <U> bzw. <L> beinhalten die Kategorie des Umkehrlinktyps bzw. des Linktyps, wobei dann in einem Diagramm: C für `composition`, E für `existence` und I für `implicit` steht.

Die Kardinalität eines Linktyps wird über einen Pfeil oder Doppelpfeil angegeben. Die Kardinalitäten der Beziehungstypen werden in H-PCTE mit Hilfe der über den Linktypen definierten Schlüsselattribute festgelegt. Dabei sichert der Verzicht auf die Definition eines

Schlüsselattributs im entsprechenden Linktyp, daß nur genau eine solche Beziehung zwischen zwei Objekten bestehen kann, wobei die Definition mindestens eines Schlüsselattributs gewährt, daß mehrere solcher Beziehungen zwischen jeweils zwei Objekten bestehen können. Durch Kombination von Linktypen mit Schlüsselattributen und Linktypen ohne Schlüsselattribute können so alle Kardinalitäten von Beziehungstypen realisiert werden.

In H-PCTE kann ein Objekttyp von anderen Objekttypen abgeleitet werden und ist vom Konzept her mit einer Spezialisierung eines Objekttyps (IS-A-Beziehung) gleichzusetzen. Diese Spezialisierung wird über folgendes Symbol ausgedrückt:



Aus Gründen einer geeigneten Abstraktion innerhalb des Schema-Diagramms, faßt folgendes Symbol einen Teil eines Schema-Diagramms zu einem Sub-Schema-Diagramm zusammen:

4.4.2 Schema für P-Files und Verwaltungsstrukturen

P-Files und Sub-P-Files werden in H-PCTE als komplexe Objekte modelliert. Ein (Sub-)P-File-Objekt besteht, motiviert durch das in Abschnitt 4.1 ab Seite 50 beschriebene Verteilungskonzept, aus einem *Info-Objekt* und einem *Knoten-Objekt*. Das Info-Objekt besteht aus einem Objekt vom Objekttyp `p_file_info` und das Knoten-Objekt aus einem Objekt vom Objekttyp `p_file_node`. Der Objekttyp `p_file_info` wird von dem Objekttyp `basis_object` abgeleitet. Das Attribut `creation_time` vom Typ `time` enthält immer die Zeitangabe, zu der das P-File- bzw. Sub-P-File-Objekt erzeugt wurde. Das Attribut `owner_name` vom Typ `string` verwaltet den Namen des Besitzers des (Sub-)P-Files. In Erweiterung des zuvor definierten (Sub-)P-File-Objekts, wird das Info-Objekt mit dem Info-Objekt seines Sub-P-File-Objekts per `composition`-Link vom Linktyp `has_sub_p_file` verbunden. Der Sub-P-File-Name dient hierbei als Schlüsselwert. Ausgehend vom Knoten-Objekt werden die ProSet-Objekte, die in dem P-File abgelegt wurden, über einen `composition`-Link vom Linktyp `has_p_file_entry` verbunden, wobei der Bezeichner des persistenten ProSet-Werts als Schlüsselwert dient. Die Modellierung eines komplexen (Sub-)P-File-Objekts mit den Einträgen als Komponentenobjekte unterstützt die Arbeit von Werkzeugen, da so z.B. mit Hilfe der von H-PCTE zur Verfügung gestellten API-Funktionen auf komplexe Objekte ein gesamtes P-File und alle seine Einträge auf einmal bearbeitet werden können.

Alle P-File- und Sub-P-File-Objekte sind per `composition`-Link mit mindestens einen *P-Store-Objekt,* ein komplexes H-PCTE-Objekt vom Typ `p_store`, verbunden. Wobei das P-Store-Objekt wiederum mit genau einem *Wurzel-Objekt,* ein H-PCTE-Objekt vom Typ `persistent_root`, verbunden ist. Über dieses Wurzel-Objekt sind alle angelegten P-Files und Sub-P-Files und somit auch alle persistenten ProSet-Werte erreichbar. Ein P-Store-Objekt macht dem Benutzer über sein Home-Objekt einen gewissen Teil der angelegten P-Files verfügbar, da intern immer über den Pfad `"~/"` navigiert wird. Es wird immer genau ein P-Store-Objekt mit dem Home-Objekt per `existence`-Link verbunden. Ein auf diese Art von einem Home-Objekt eines Benutzers verfügbar gemachtes P-Store-Objekt wird im folgenden *Home-P-Store-Objekt* genannt.

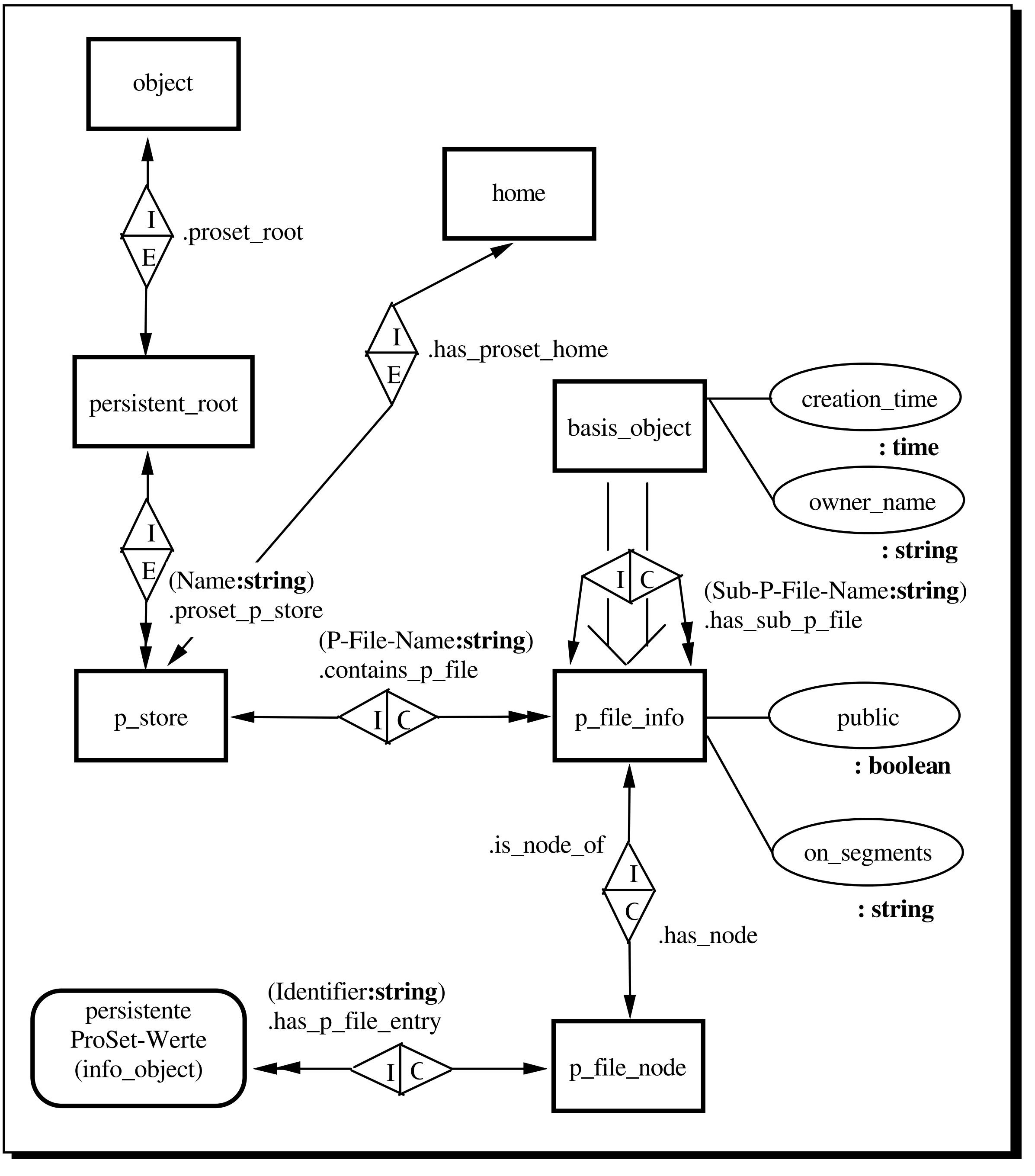


Abbildung 4-3: Schema-Diagramm für P-Files, Sub-P-Files und Administrationsobjekte

Zur Zeit existiert nur ein P-Store-Objekt, nachfolgend *PROSET-P-Store-Objekt* genannt. Das PROSET-P-Store-Objekt ist somit für jeden Benutzer über sein Home-Objekt erreichbar. Prinzipiell hat man aber die Möglichkeit, zusätzliche P-Store-Objekte anzulegen, unter denen man dann eines als Home-P-Store-Objekt auswählen kann. Somit kann man z.B. auf Versionen von (Sub-)P-Files oder persistenten PROSET-Werten arbeiten bzw. dem Benutzer nur einen Ausschnitt der in der Objektbank vorhandenen P-Files (und somit Sub-P-Files und persistenten PROSET-Werten) sichtbar machen. Auf diese Weise kann dem Benutzer also eine Art von Sicht auf Instanzen-Ebene zur Verfügung gestellt werden. Die Abbildung 4-3 stellt innerhalb eines Schema-Diagramms die Modellierung der P-Files, Sub-P-Files und Administrationsobjekte graphisch dar.

4.4.3 Schema für die persistenten PROSET-Werte

Wie bereits in Abschnitt 4.2.2 ab Seite 57 motiviert, wird ein persistenter PROSET-Wert in der Objektbank als ein komplexes H-PCTE-Objekt (PROSET-Objekt) mit den zwei Komponenten Info-Objekt und Werte-Objekt gespeichert.

Das Info-Objekt besteht aus genau einem atomaren H-PCTE-Objekt vom Objekttyp info_object, der von einem selbstdefinierten Objekttyp `audit_object` abgeleitet ist. Somit besitzt das Info-Objekt Attribute, über die ein Auditing realisiert werden kann. Die Attribute `last_change` vom Typ `time` und `changed_by` vom Typ `string` informieren darüber, welcher Benutzer zu welchem Zeitpunkt das PROSET-Objekt zuletzt verändert hat. Da der Objekttyp `audit_object` vom Objekttyp `basis_object` abgeleitet wird, enthält das PROSET-Objekt darüberhinaus die im vorigen Abschnitt bereits erläuterten Attribute.

Das Werte-Objekt kann abhängig vom Typ des PROSET-Werts ein atomares oder komplexes H-PCTE-Objekt sein. Dieses Objekt wird im folgenden *atomares* bzw. *komplexes Werte-Objekt* genannt oder beide Fälle umfassend nur als *Werte-Objekt* bezeichnet.

Wie in Abschnitt 4.2.2 ab Seite 57 erläutert, wird ein PROSET-Wert der innerhalb von persistenten PROSET-Werten mitabgespeichert werden muß, allein über das Werte-Objekt als internes PROSET-Objekt repräsentiert.

4.4.3.1 Schema für primitive PROSET-Werte

Hier wird nun beschrieben, wie die PROSET-Datentypen auf die von H-PCTE angebotenen Attributtypen abgebildet und wie die primitiven PROSET-Werte geeignet modelliert werden können.

Der PROSET-Datentyp `integer` entspricht dem C-Datentyp `integer`. Da ebenfalls der PCTE-Datentyp `integer` auf den C-Datentyp `integer` abgebildet wurde, kann ein PROSET-Wert vom Typ `integer` auf ein PROSET-Objekt mit einem atomaren Werte-Objekt vom selbstdefinierten Objekttyp `ps_integer` mit einem Attribut vom Typ PCTE-`integer` abgebildet werden.

Der PROSET-Datentyp `real` entspricht dem C-Datentyp `double`, über den ein Gleitkommawert mit doppelter Genauigkeit definiert wird. Allerdings wird der Attributtyp PCTE-`float` auf den C-Datentyp `float` abgebildet, über den eine Gleitkommazahl mit einfacher Genauigkeit definiert wird. Eine Abbildung des PROSET-`real` auf den PCTE-`float` würde somit einen Informationsverlust verursachen. Aus diesem Grunde wird ein PROSET-Wert vom Typ `real` über ein PROSET-Objekt mit einem atomaren Werte-Objekt vom selbstdefinierten Objekttyp `ps_real` mit einem Attribut vom Typ PCTE-`string` beschrieben. Intern in der Persistenzschnittstelle wird hier eine entsprechende Konvertierung zwischen PROSET-`real` und PCTE-`string` vorgenommen.

Die Datentypen PROSET-`boolean` und PCTE-`boolean` besitzen wiederum jeweils eine Entsprechung in einer C-Datenstruktur, die zueinander kompatibel ist. Damit kann ein PROSET-Wert vom Typ `boolean` auf ein PROSET-Objekt mit einem atomaren Werte-Objekt vom selbstdefinierten Objekttyp `ps_boolean` mit einem Attribut vom Typ PCTE-`boolean` abgebildet werden.

Die gleiche Aussage trifft auch auf die Datentypen ProSet-`string` und PCTE-`string` zu, so daß hier ein ProSet-Wert vom Typ `string` seine Entsprechung in einem ProSet-Objekt mit einem atomaren Werte-Objekt vom selbstdefinierten Objekttyp `ps_string` mit einem Attribut vom Typ PCTE-`string` findet.

Eine spezielle Bedeutung hat der ProSet-Datentyp `atom`. Seine Instanz soll einen (weltweit) eindeutigen Wert beinhalten. Aus diesem Grunde entschloß man sich, die Einzigartigkeit des `atom`-Werts über eine Beschreibung in Form einer erweiterten Internetadresse zu sichern. Diese Beschreibung ist in ihrer abstrakten Form textuell, so daß es sich anbietet, diese Notation als PCTE-`string` abzuspeichern. Somit kann ein ProSet-Wert vom Typ Atom auf ein ProSet-Objekt mit einem atomaren Werte-Objekt vom Objekttyp `ps_atom` mit einem Attribut vom Typ PCTE-`string` abgebildet werden.

Einen Sonderfall bildet ein `om`-Wert (Omega): Über diesen Wert beschreibt man in ProSet einen typlosen, undefinierten Wert. Da ein solcher Wert jedoch ebenfalls Bürgerrechte erster Klasse besitzt und darüber hinaus auch in einem ProSet-Wert vom Typ `tuple` vorkommen kann, muß dieser auch in dem Datenbankschema berücksichtigt werden. Ein `om`-Wert kann nur genau einen Wert (interpretiert als undefinierten Wert) annehmen, so daß er eine Konstante darstellt. So kann dieser Wert über ein ProSet-Objekt mit einem atomaren Werte-Objekt vom selbstdefinierten Objekttyp `ps_omega` dargestellt werden, das kein Attribut benötigt.

Primitive ProSet-Datentypen	von H-PCTE angebotene Attributtypen
`atom`	
`string`	`string`
`real`	
`integer`	`integer`
`boolean`	`boolean`

Tabelle 4-5: Abbildung der primitiven ProSet-Datentypen in die von H-PCTE angebotenen Attributtypen

Die Tabelle 4-5 präsentiert zusammengefaßt, wie die Datentypen von ProSet in die von H-PCTE angebotenen Attributtypen abgebildet werden.

Die nachfolgende Abbildung 4-4 veranschaulicht innerhalb eines Schema-Diagramms die Modellierung der primitiven persistenten ProSet-Werte.

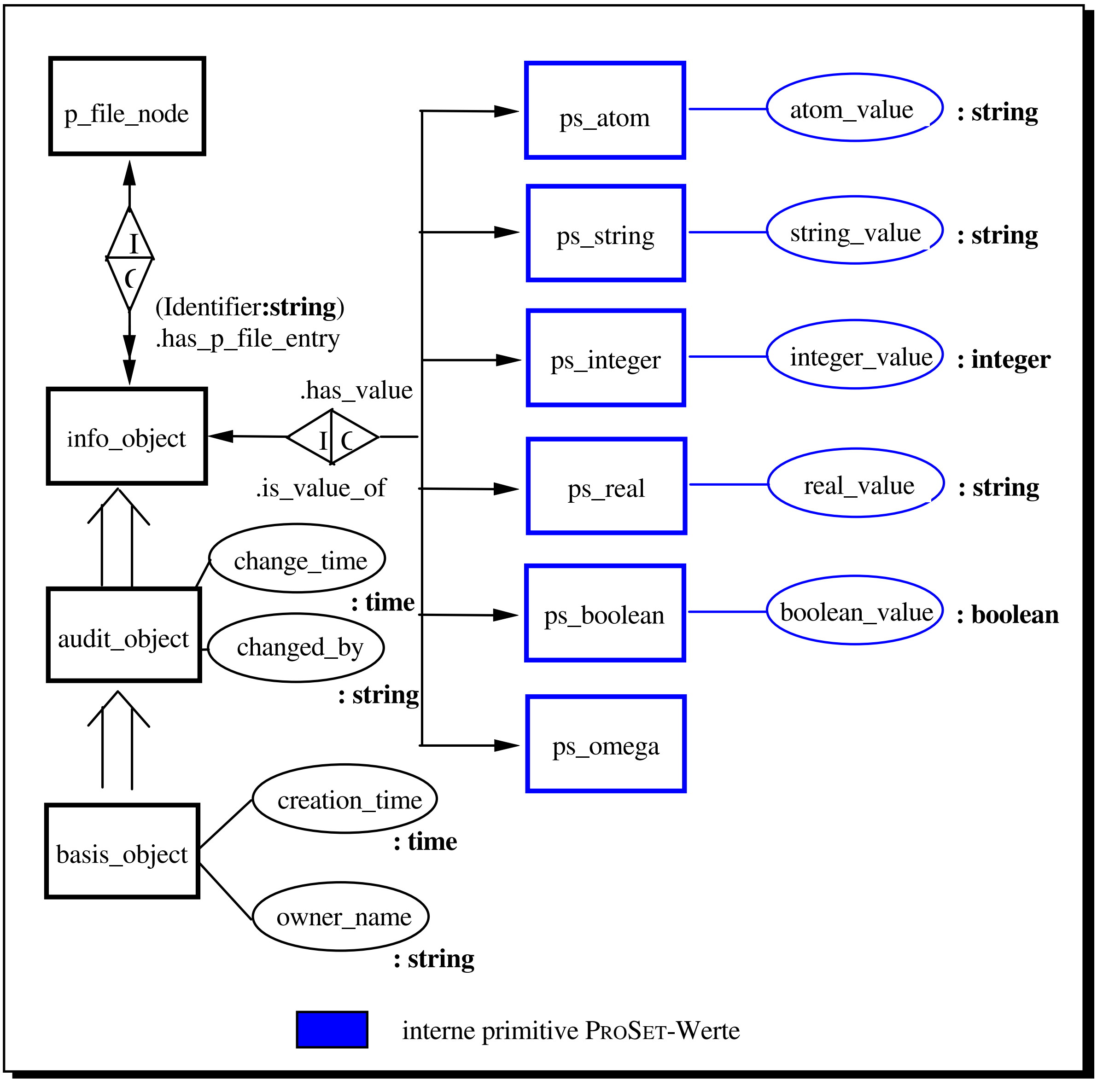


Abbildung 4-4: Schema-Diagramm für primitive persistente PROSET-Werte

4.4.3.2 Schema für zusammengesetzte PROSET-Werte

Ein PROSET-Wert vom Datentyp `set` oder `tuple` kann als Element:

- alle primitiven PROSET-Werte,
- wiederum alle zusammengesetzten PROSET-Werte (also sind geschachtelte Strukturen möglich) und
- alle PROSET-Werte höherer Ordnung enthalten.

Ein PROSET-Wert vom Typ `tuple` (geordnete Menge) kann im Unterschied zum PROSET-Wert vom Typ `set` (ungeordnete Menge) `om`-Werte enthalten. Da auf die Elemente eines `tuple`-Werts eine Ordnung definiert ist, hat die Reihenfolge der im `tuple`-Wert enthaltenen Elemente eine Bedeutung. Deshalb müssen die Positionsangaben mit als Information abgespeichert werden.

Die zusammengesetzten PROSET-Werte `set` und `tuple` können in H-PCTE als ein PROSET-Objekt mit einem komplexe Werte-Objekt vom Objekttyp `ps_set` bzw. `ps_tuple` modelliert werden, wobei ein einziges atomares H-PCTE-Objekt als Wurzel-Objekt des Werte-Objekts mit jedem seiner Elemente über einen `composition`-Link verbunden ist. Der Bezeichner des zusammengesetzten PROSET-Werts dient auch hier als Schlüsselwert für den Link auf das entsprechende PROSET-Objekt, wogegen die Indizes der Elemente des zusammengesetzten PROSET-Werts als Schlüsselwerte für die Links auf die Element-Objekte dienen. Die Modellierung der zusammengesetzten persistenten PROSET-Werte wird in dem Schema-Diagramm in Abbildung 4-5 graphisch visualisiert.

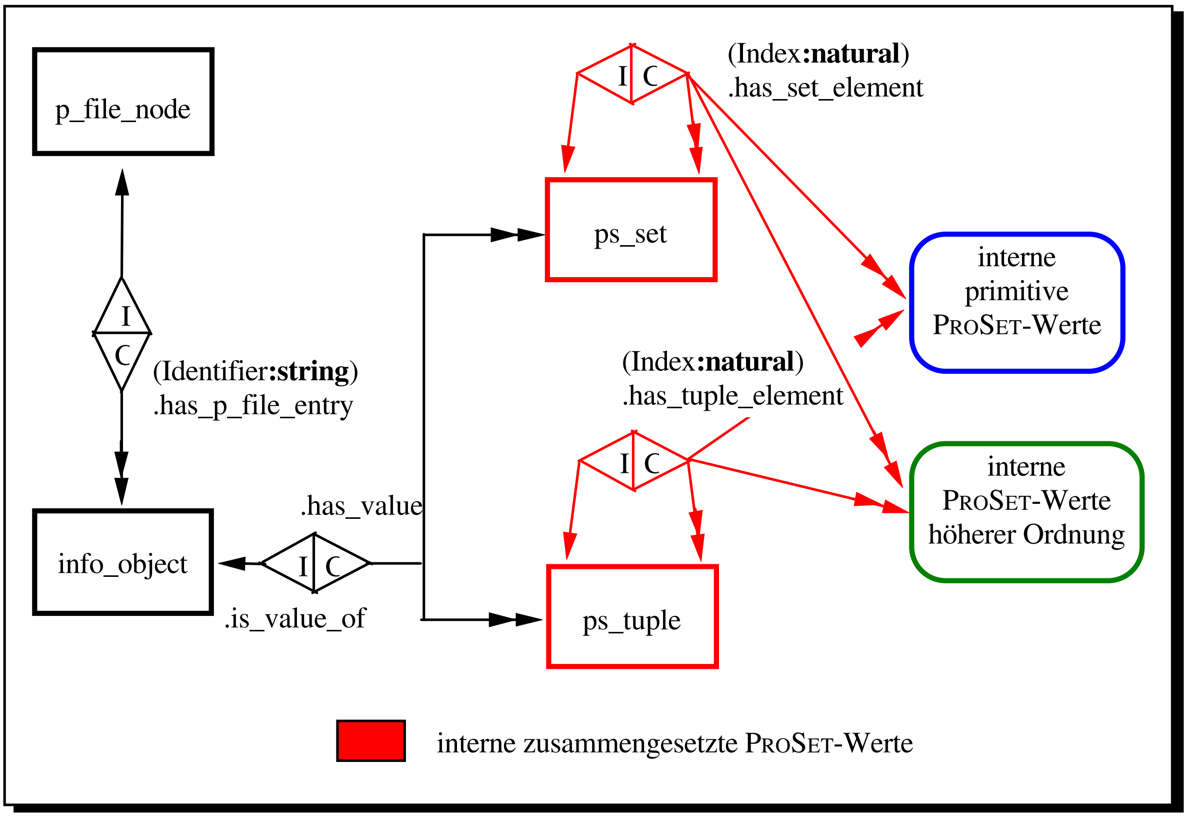


Abbildung 4-5: Schema-Diagramm für zusammengesetzte persistente PROSET-Werte

4.4.3.3 Schema für PROSET-Werte höherer Ordnung

Ein persistenter PROSET-Wert höherer Ordnung kann [DFG+92a]

- vom Datentyp `function` sein, der per `closure`-Konstrukt aus einer `procedure` gebildet wurde,
- vom Datentyp `modtype` sein, der per `closure`-Konstrukt aus einem `module` erzeugt wurde, oder
- vom Datentyp `instance` sein, der per `instantiate`-Konstrukt aus einem `modtype` gebildet wurde.

Jeder persistente PROSET-Wert höherer Ordnung wird über ein PROSET-Objekt mit einem komplexen Werte-Objekt vom entsprechenden selbstdefinierten Objekttyp dargestellt. Der „Basis-Objekttyp", von dem alle diese selbstdefinierten Objekttypen abgeleitet werden, ist `unit`. Darüber hinaus repräsentiert eine Instanz vom Typ `unit` einen statischen Nachfolger eines (geschachtelten) persistenten oder internen ProSet-Werts höherer Ordnung. Eine solche Instanz enthält nur als wesentliche Information den Objekt-Code der im ProSet-Wert höherer Ordnung geschachtelten Programmeinheit, sowie wiederum die statischen Nachfolger dieses internen ProSet-Werts. Auf diese Weise verwaltet jede in einem ProSet-Wert höherer Ordnung geschachtelte Programmeinheit ihren Objekt-Code selbst. Somit wird der Objektcode in dem `object_code`-Attribut vom Typ `string` des entsprechenden H-PCTE-Objekts gespeichert, wobei dieses Attribut bei einem Zugriff auf das Objekt immer als Longfield-Attribut geöffnet wird. Das Attribut `code_size` vom Typ `integer` verwaltet die Größe des Objekt-Codes. In dem Attribut `extern_name` vom Typ `string` wird der Name der Programmeinheit gespeichert, von der per `closure` ein ProSet-Wert höherer Ordnung erzeugt wurde. Das Attribut `is_dld_code` vom Typ `boolean` zeigt an, ob der Objekt-Code dazu geeignet ist, um per DLD geladen zu werden, oder ob er als Shared Library erzeugt wurde. Diese Thema wird in Abschnitt 5.5.1.1 ab Seite 82 aufgegriffen. Das Parameter-Profil der Programmeinheit wird in einem Attribut `profile` vom Typ `string` als Zeichenkette codiert abgelegt.

Von dem Objekttyp `unit` werden die Objekttypen, deren Instanzen einen ProSet-Wert zweiter Klasse darstellen, direkt abgeleitet. Da ProSet-Werte vom Typ `function`, `modtype` und `instance` sowohl gemeinsame als auch typspezifische Eigenschaften aufweisen, wird ein Objekttyp `closure_unit` zwischengeschaltet. Die Identität eines jeden ProSet-Werts höherer Ordnung wird über einen ihm zugeordneten ProSet-Wert vom Typ `atom` gesichert. So *muß* mit einem H-PCTE-Objekt vom Typ `closure_unit` immer genau ein (also 1:1-Beziehung) H-PCTE-Objekt vom Typ `ps_atom` in Beziehung `has_identity` stehen. Diese 1:1-Beziehung wird über einen Link ohne Schlüsselwert realisiert. Weiterhin *kann* jeder ProSet-Wert höherer Ordnung eine *eingefrorene Umgebung* besitzen. Eine solche Umgebung enthält alle per `closure`-Konstrukt eingefrorenen nicht-lokalen Werte und macht diese Werte somit bei einem späteren Laden des ProSet-Werts verfügbar. Aus diesem Grunde besteht eine (0,1:N)-Beziehung `has_frozen_env` zwischen dem ProSet-Wert höherer Ordnung und allen ProSet-Werten mit Bürgerrechten erster Klasse. Darüber hinaus kann eine solche Beziehung ebenfalls zu ProSet-Werten, die keine Bürgerrechte erster Klasse besitzen, bestehen. Diese Relation wird über einen Link mit einem Schlüsselwert, der den Index des ProSet-Werts in der eingefrorenen Umgebung enthält, beschrieben (`<Index>.has_frozen_env`). Dieser Index wird gemäß der internen Repräsentation in der Laufzeitbibliothek von ProSet übernommen und ist aus diesem Grunde normalerweise nicht fortlaufend. Damit die eingefrorene Umgebung des persistenten ProSet-Werts während des Ladevorgangs wiederhergestellt werden kann, benötigt man die Größe der Umgebung seines (damaligen) statischen Vorgängers. Deshalb unterhält eine Instanz vom Typ `closure_unit` ein Attribut `pred_env_size` vom Typ `integer`.

Von dem Objekttyp `closure_unit` können nun die Objekttypen `ps_function` und `ps_modtype` abgeleitet werden, deren Instanzen einen ProSet-Wert vom Typ `function` bzw. `modtype` persistent speichern. Da zur Zeit intern für beide Typen von ProSet-Werten die gleichen Informationen persistent abgelegt werden müssen, werden auf dieser Typhierarchiestufe keine zusätzlichen Attribute benötigt. Da sich jedoch grundsätzlich ProSet-Werte vom Typ `function` von ProSet-Werten vom Typ `modtype` unterscheiden, wurde in die Ableitungshierarchie der Objekttyp `closure_unit` eingefügt.

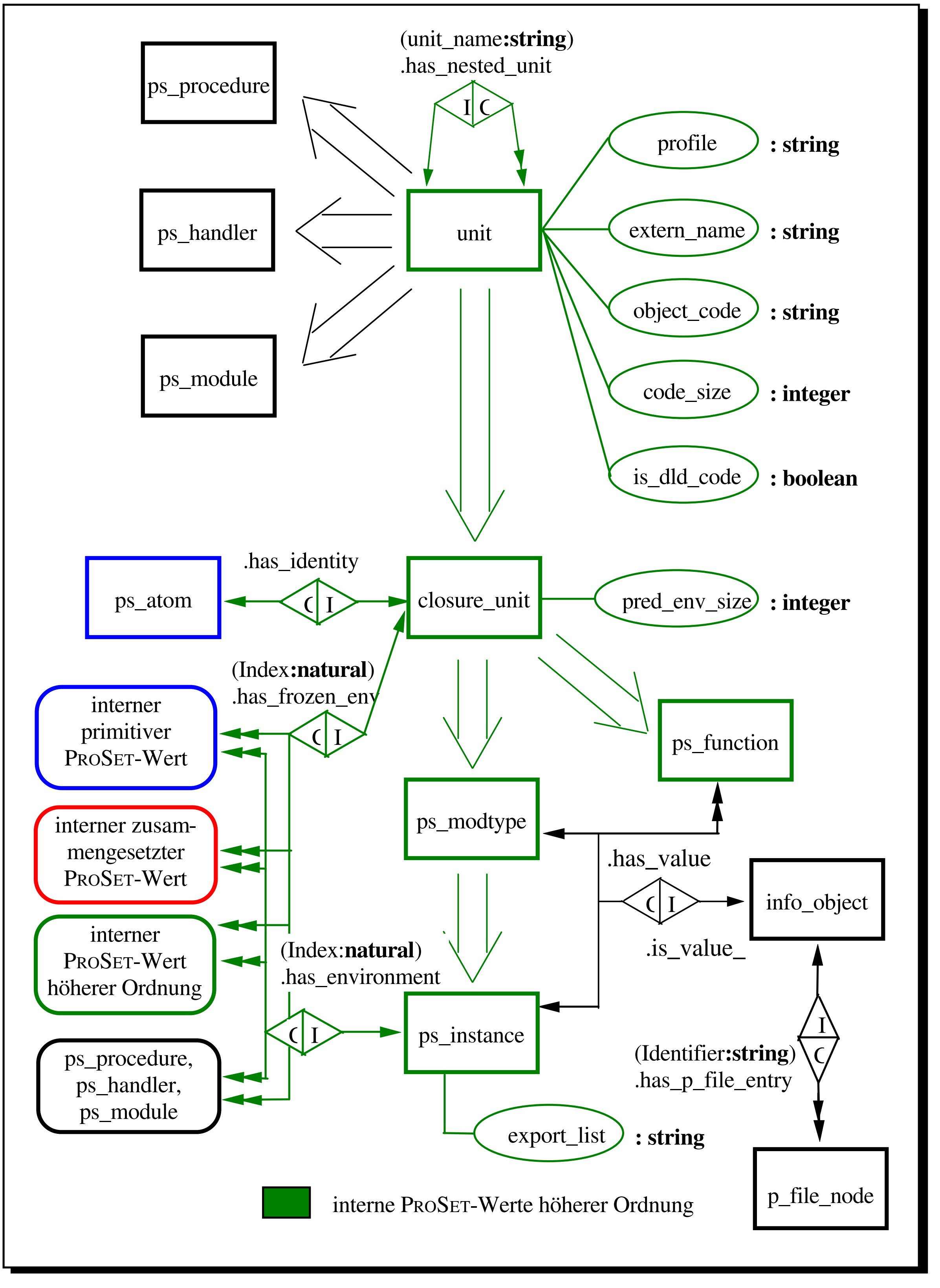


Abbildung 4-6: Schema-Diagramm für persistente ProSet-Werte höherer Ordnung

Von dem Objekttyp `ps_modtype` kann nun der Objekttyp `ps_instance`, dessen Instanz einen ProSet-Wert vom Typ `instance` speichert, abgeleitet werden. Zur Speicherung eines solchen Werts werden zusätzlich zum `modtype`-Wert noch die Exportliste und die Umgebung der Modulinstanz gespeichert. Aus diesem Grunde besteht eine (1:N)-Beziehung `has_environment` zwischen dem ProSet-Wert vom Typ `instance` und den ProSet-Werten seiner Umgebung. In der Umgebung können potentiell ProSet-Werte von jeglichem Typ vorkommen. Eine solche Relation wird über einen Link mit einem Schlüsselwert, der den Index des ProSet-Werts in der Umgebung enthält, beschrieben. Dieser Index wird gemäß der internen Repräsentation in der Laufzeitbibliothek von ProSet übernommen und ist aus diesem Grunde normalerweise nicht fortlaufend. In dem in obiger Abbildung 4-6 präsentierten Schema-Diagramm wird die Modellierung der persistenten ProSet-Werte höherer Ordnung veranschaulicht.

4.5 Architektur der Persistenzschnittstelle

Nachfolgend wird nun die Architektur der Persistenzschnittstelle beschrieben, wobei die in Abschnitt 3.5 ab Seite 48 gestellten Anforderungen berücksichtigt werden.

Alle C-Module der Persistenzschnittstelle beginnen mit dem Präfix `"ps_"` als Akronym für ProSet oder persistent. Die Architektur der Schnittstelle weist auf einem höheren Abstraktionsniveau zwei Schichten auf:

1. Eine erste Schicht, welche die von H-PCTE über seine API angebotenen Funktionen geeignet zusammenfaßt und kapselt. Diese Schicht soll im folgenden in Anlehnung an [Kel94] mit *Common Services-Schicht* umschrieben werden, da sie allgemein benötigte Dienste der ihr überliegenden Schicht anbietet.
2. Eine zweite Schicht, die aufbauend auf den Common Services verschiedene auf den Bedarf von ProSet zugeschnittene Schnittstellen-Funktionen anbietet. Diese Schicht soll deshalb *ProSet-Dienste-Schicht* genannt werden.

Die Common Services-Schicht umfaßt folgende Module:

- `ps_hpcte`: Dieses Funktionsmodul kapselt die von H-PCTE angebotenen allgemeinen Dienste. Dies sind im wesentlichen Funktionen zum Erzeugen, Löschen und Ändern von Objekten, Links und Attributen. Dieses Modul wurde in Gemeinschaftsarbeit mit Jörg Heinrich erstellt [Hei95]. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"h_"`.
- `ps_security`: Dieses Funktionsmodul bietet Funktionen zur Realisierung des Schutzkonzepts von ProSet. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"sec_"`.
- `ps_auditing`: Dieses Funktionsmodul bietet Funktionen zur Realisierung eines Auditings. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"adg_"`.
- `ps_transact`: Dieses Funktionsmodul bietet Funktionen zur Realisierung des Transaktionskonzepts von ProSet. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"ta_"`.
- `ps_distrib`: Dieses Funktionsmodul bietet Funktionen zur Realisierung des Verteilungskonzepts von ProSet. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"dtb_"`.

Die PROSET-Dienste-Schicht umfaßt folgende Module:

- `ps_primitive`: Dieses Funktionsmodul stellt Funktionen zum Laden und Speichern von primitiven persistenten PROSET-Werten zur Verfügung. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"pr_"`.
- `ps_complex`: Dieses Funktionsmodul bietet Funktionen zum Laden und Speichern von komplexen persistenten PROSET-Werten, womit zusammengesetzte PROSET-Werte und PROSET-Werte höherer Ordnung gemeint sind.. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"cp_"`.
- `ps_dynlink`: Dieses Funktionsmodul bietet Funktionen zur Unterstützung des Ladens von persistenten PROSET-Werten höherer Ordnung, d.h. konkret, es werden diverse Funktionen zum dynamischen Linken des Objekt-Codes der PROSET-Werte höherer Ordnung zur Verfügung gestellt. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"dl_"`.
- `ps_debug`: Dieses Modul stellt verschiedene Funktionen zur Unterstützung des Debuggings beim Laden und Speichern der persistenten PROSET-Werte zur Verfügung. Diese Funktionen liefern insbesondere Informationen über die interne sehr komplexe Struktur der persistenten PROSET-Werte höherer Ordnung.
- `ps_lang` (lang steht als Akronym für language): Dieses Funktionsmodul bietet auf den Bedarf von PROSET und PROSET-Werkzeugen abgestimmte Funktionen zum Laden und Speichern von persistenten PROSET-Werten an. Dabei wird das Schutzkonzept von PROSET inklusive der Auditingfunktionen integriert. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"lg_"`.
- `ps_p_file`: Dieses Funktionsmodul bietet Funktionen zum Erzeugen und Löschen von P-Files an. Dabei wird das Verteilungskonzept von PROSET integriert. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"pf_"`.
- `ps_persistent`: Dieses Funktionsmodul bietet speziell auf den Bedarf des PROSET-Compilers abgestimmte Funktionen zum Laden und Speichern von persistenten PROSET-Werten an. Dabei wird das Transaktionskonzept von PROSET integriert. Die Funktionen dieses Moduls besitzen unterschiedliche auf die Laufzeitbibliothek von PROSET abgestimmte Präfixe.
- `ps_command`: Dieses Funktionsmodul bietet Funktionen, die in einem PROSET-Modul gekapselt und so später in einem PROSET-Programm benutzt werden können. Die Parameterschnittstellen dieser Funktionen sind auf Kosten der Effizienz minimal und von der Funktionalität möglichst mächtig gehalten und können deshalb auch innerhalb von PROSET-Werkzeugen benutzt werden. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"cmd_"`.
- `ps_install`: Dieses Funktionsmodul bietet Administrationsfunktionen für eine Installation der zur Realisierung der Persistenz benötigten PROSET-Administrationsobjekte an. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"ins_"`.
- `ps_tool`: Dieses Modul bietet mächtige Funktionen für Werkzeuge zur Unterstützung der Persistenzkonzepte von PROSET an. Hier werden insbesondere Funktionen zur Benutzer und Benutzergruppenverwaltung, zum Setzen von Zugriffsrechten auf persistente

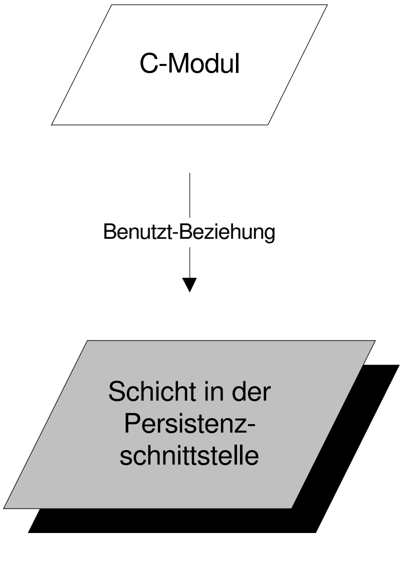
PROSET-Werte und (Sub-)P-Files, sowie zur Unterstützung des Verteilungskonzepts angeboten. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"tl_"`.

Um die Portierbarkeit der Persistenzschnittstelle so gut wie möglich zu unterstützen, werden folgende Module angeboten, wobei ps_util, ps_konst und ps_types in Zusammenarbeit mit Jörg Heinrich konstruiert wurden.:

- `ps_util`: Dieses Funktionsmodul bietet diverse Hilfsfunktionen an. Hierzu zählen auch Funktionen, welche die über die C-Standard-Bibliotheken zur Verfügung gestellten Ein- und Ausgabefunktionen kapseln. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"u_"`.
- `ps_error`: Dieses Funktionsmodul stellt verschiedene Funktionen sowie selbstdefinierte Fehlercodes zur Realisierung einer geeigneten Fehlerbehandlung innerhalb der Schnittstelle und für Werkzeuge, welche die Funktionen dieser Schnittstelle benutzen, zur Verfügung. Alle Funktionen dieses Moduls beginnen mit dem Präfix `"err_"`.
- `ps_konst`: Dieses Modul definiert einzig verschiedene Konstanten, wie z.B. für alle in der Persistenzschnittstelle verwendeten Objekttypen, Linktypen, Attributnamen, usw.
- `ps_types`: Dieses Modul definiert für die in der Persistenzschnittstelle benötigten H-PCTE-Typen neue Typen und stellt darüber hinaus weitere in der Schnittstelle benötigte Typen zur Verfügung.
- `ps_storage`: Kapselt die über C-Standard-Bibliotheken zur Verfügung gestellten Speicherallokationsfunktionen.

Diese zuletzt genannten Module, sowie die Module der PROSET-Bibliothek werden von allen Modulen der beiden Schichten benutzt. Um die in Abbildung 4-7 vorgestellte Modularchitektur übersichtlicher zu gestalten, wurden diese Module weggelassen.

In der Modularchitektur werden folgende Symbole benutzt:



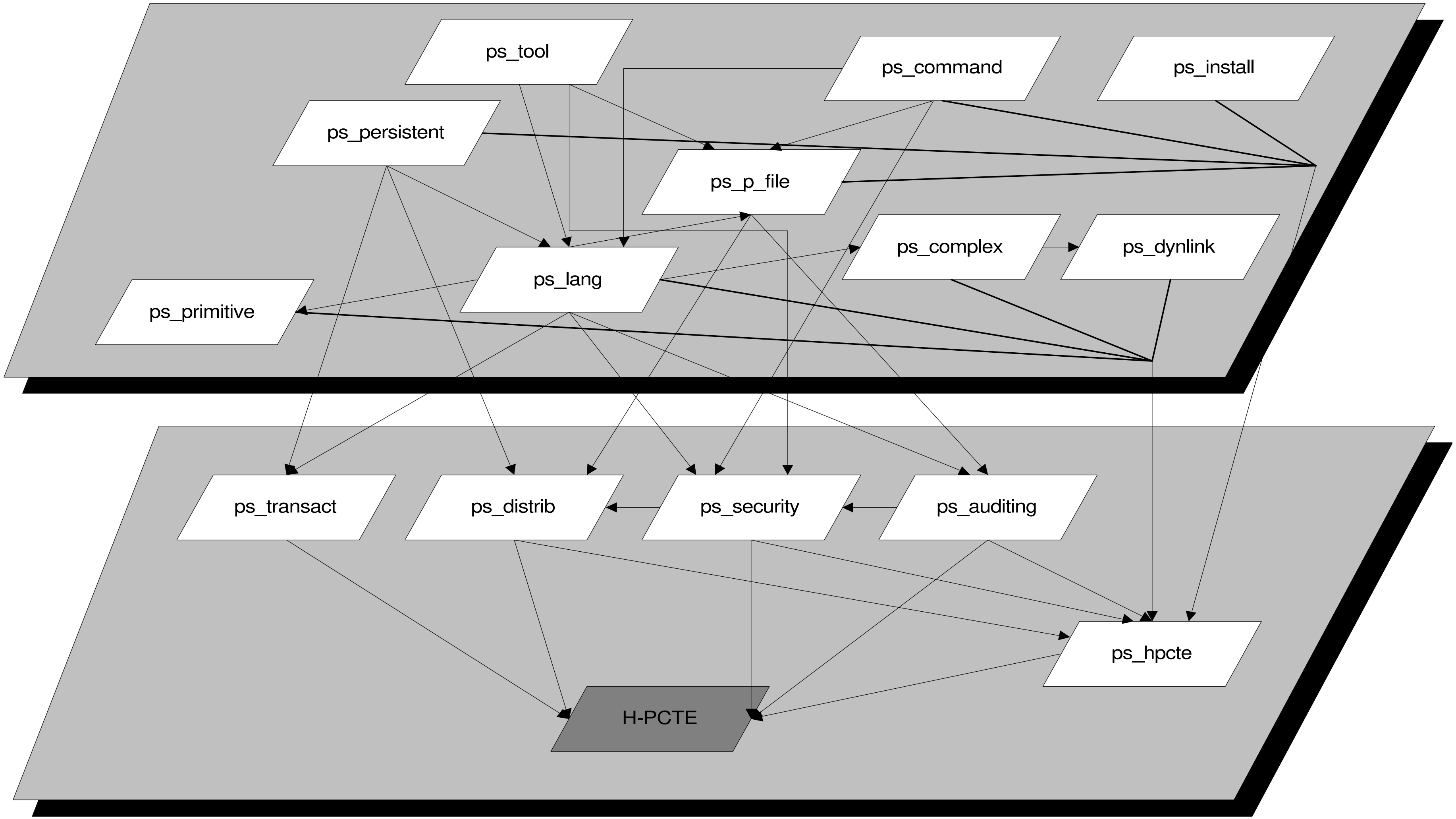


Abbildung 4-7: Modularchitektur der Persistenzschnittstelle

5 Implementierung

Im folgenden wird nun beschrieben, wie die einzelnen Persistenzkonzepte implementiert wurden. Hierzu wird erläutert, wie eine H-PCTE-Installation darauf vorbereitet wird, als Repository für persistente ProSet-Werte zu dienen, wie Benutzer und Benutzergruppen in H-PCTE angelegt werden, wie persistente ProSet-Werte geladen und gespeichert werden und wie der ProSet-Compiler die über die Persistenzschnittstelle angebotenen Funktionen in das ProSet-Programm einfügt. Außerdem werden die wichtigsten Details bei der Implementierung der Persistenzkonzepte erklärt.

5.1 Installation

Bevor man die Persistenz von ProSet nutzen kann, muß eine H-PCTE-Installation vorgenommen werden. Diese Installation sollte der Benutzer vornehmen, der als Systemadministrator fungieren soll. Denn die ACL aller bei der Installation erzeugten Objekte, die u.a. die Metadaten von H-PCTE verwalten, enthält initial nur einen ACL-Eintrag für den Benutzer, der die Installation vorgenommen hat, wobei für ihn alle H-PCTE-Rechte gesetzt sind. Nach der H-PCTE-Installation (siehe [See94b]) müssen dann die Verwaltungsstrukturen zur Realisierung der Persistenz von ProSet angelegt werden. Dabei müssen die Benutzergruppe `systemadministrator` in die Benutzergruppenverwaltung von H-PCTE eingetragen, ein ProSet-Administrationssegment erzeugt und die ProSet-Administrationsobjekte auf diesem Segment angelegt werden. Hierzu wird über dem Modul `ps_install` die Funktion `ins_initial_objects_create` angeboten. Innerhalb dieser Funktion wird:

- die Benutzergruppe `systemadministrator` in H-PCTE angelegt und der Benutzer, der die Installation vornimmt, als erster Systemadministrator in diese Gruppe eingetragen. Die ACL jedes im folgenden erzeugten Objekts enthält einen ACL-Eintrag für die Benutzergruppe `systemadministrator`. Auf diese Objekte kann dann nämlich jeder Systemadministrator zugreifen.
- Schließlich werden das ProSet-Administrationssegment und die ProSet-Administrationsobjekte (siehe Abschnitt 4.1.3 ab Seite 53) erzeugt.

Auf Basis dieser Installation kann nun zumindest der Systemadministrator die Persistenz von ProSet nutzen, d.h. P-Files bzw. Sub-P-Files anlegen und persistente ProSet-Werte speichern und laden.

5.2 Anlegen von Benutzern und Benutzergruppen

Um nun aber auch einen Mehrbenutzerbetrieb zu ermöglichen, müssen weitere Benutzer in die H-PCTE-Installation eingetragen werden. In diesem Abschnitt werden deshalb die Funktionen vorgestellt, die zur Benutzer- und Benutzergruppenverwaltung angeboten werden

Das Modul `ps_tool` stellt verschiedene Funktionen zur Verfügung für eine Benutzer- und Benutzergruppenverwaltung. Diese Funktionen sollte nur ein Systemadministrator ausführen, da er bestimmte Zugriffsrechte auf die Administrationsobjekte innerhalb der Objektbank benötigt.

- `tl_user_create`: Eintragen eines neuen Benutzers in die Benutzerverwaltung von H-PCTE. Da man sich auf die Identifikations- und Authentifikationsmechanismen von UNIX abstützt, muß dieser neue Benutzer unter der gleichen Benutzerkennung in Form seines Namens eingetragen werden, unter der er auch in der Benutzerverwaltung von UNIX eingetragen ist. Innerhalb dieser Funktion wird auch automatisch ein Home-Objekt auf einem eigenen Segment für den Benutzer angelegt und dieses mit einem P-Store-Objekt verbunden. Außerdem werden benötigte Rechte implizit auf das Home-Objekt und das P-Store-Objekt zur Realisierung eines Mehrbenutzerbetriebs gesetzt (siehe Abschnitt 4.2.6 ab Seite 60).
- `tl_group_create`: Eintragen einer neuen Gruppe in die Benutzergruppenverwaltung von H-PCTE.
- `tl_group_add_user`: Eintragen eines Benutzers in eine Benutzergruppe.
- `tl_group_add_subgroup`: Eintragen einer Benutzergruppe als Untergruppe einer anderen Benutzergruppe.

Es werden keine Funktionen zum Austragen von Benutzern und Benutzergruppen aus der Benutzer- bzw. Benutzergruppenverwaltung von H-PCTE sowie zum Entfernen von Benutzern und Benutzergruppen aus einer Benutzergruppe angeboten, da die API von H-PCTE hierzu keine Funktionen anbietet. Ein Systemadministrator, der die internen Verwaltungsstrukturen von H-PCTE kennt, kann allerdings über den von H-PCTE zur Verfügung gestellten Browser `hbrowse` manuell entsprechende Objekte und Links löschen.

Wenn Benutzer und Benutzergruppen gemäß der organisatorischen Umgebung angelegt wurden, sollte man nun für jeden Benutzer P-Files und Sub-P-Files anlegen.

5.3 P-Files und Sub-P-Files

P-File-Objekte darf nur ein Systemadministrator ausgehend von einem P-Store-Objekt erzeugen und löschen. Aus diesem Grunde werden bei der Erzeugung eines P-Store-Objekts implizit für die `systemadministrator`–Benutzergruppe entsprechende Rechte auf das P-Store-Objekt gesetzt. Funktionen zum Anlegen und Löschen von P-Files, die von einem Werkzeug benutzt werden können, werden im Modul `ps_tool` zur Verfügung gestellt:

- `tl_p_file_create`: Erzeugt ausgehend von einem P-Store-Objekt ein P-File- oder Sub-P-File-Objekt für einen Benutzer, wobei das Info-Objekt auf dem PROSET-Administrationssegment angelegt wird und das Knoten-Objekt auf einem eigenen Segment. Dem Besitzer werden initial alle Rechte (read[V], write[V]) auf das P-File-Objekt zugeteilt. Außerdem werden für den Besitzer die benötigten Zugriffsrechte auf das P-Store-Objekt gesetzt.
- `tl_p_file_delete`: Löscht ein P-File- oder Sub-P-File-Objekt.

Entsprechende Funktionen, die später von einem Systemadministrator über ein PROSET-Modul genutzt werden sollen, werden über die Funktionen `cmd_p_file_create` und `cmd_p_file_delete` des Moduls `ps_command` angeboten. Zusätzlich werden noch die Funktionen `cmd_p_file_entries_list` und `cmd_sub_p_files_list` zur Verfügung gestellt.. Über diese Funktionen kann der Inhalt eines (Sub)P-Files innerhalb eines PROSET-Werts vom Typ `set` aufgelistet werden. Die erste Funktion liefert nur die Bezeichner der persistenten PROSET-Werte, während die zweite Funktion die P-File-Pfadnamen der in dem P-File bzw. Sub-P-File enthaltenen Sub-P-Files auflistet. Somit kann mit Hilfe dieser zwei Funktionen rekursiv der Inhalt eines Sub-P-Files ausgelesen werden.

Damit ein Werkzeug spezifizieren kann, ob ein Segment eines P-Files oder Sub-P-Files lokal oder global geladen werden soll, werden folgende Funktionen über das Modul `ps_tool` angeboten:

- `tl_load_locally_on`: Das Segment des (Sub-)P-Files wird lokal geladen, wenn ein PROSET-Programm auf einen der in dem (Sub-)P-File abgelegten persistenten PROSET-Werte zugreift.
- `tl_load_locally_off`: Das Segment des (Sub-)P-Files wird global geladen, wenn ein PROSET-Programm auf einen der in dem (Sub-)P-File abgelegten persistenten PROSET-Werte zugreift.

5.4 Setzen von PROSET-Zugriffsrechten

Um nun für die angelegten Benutzer und Benutzergruppen die Zugriffsrechte auf P-Files und persistente PROSET-Werte setzen zu können, werden über das Moduls `ps_tool` diverse Funktionen angeboten:

- `tl_value_access_rights_set`: Setzt für einen Benutzer oder eine Benutzergruppe PROSET-Zugriffsrechte auf einen persistenten PROSET-Wert.
- `tl_pf_access_rights_set`: Setzt für einen Benutzer oder eine Benutzergruppe PROSET-Zugriffsrechte auf ein P-File oder Sub-P-File.
- `tl_unix_value_access_rights_set`: Setzt für einen Benutzer oder eine Benutzergruppe PROSET-Zugriffsrechte mit einer zu UNIX kompatiblen zweiwertigen Rechtelogik auf einen persistenten PROSET-Wert.
- `tl_unix_pf_access_rights_set`: Setzt für einen Benutzer oder eine Benutzergruppe PROSET-Zugriffsrechte mit einer zu UNIX kompatiblen zweiwertigen Rechtelogik auf ein P-File oder Sub-P-File..

Darüber hinaus werden innerhalb des Moduls `ps_command` entsprechende auf die PROSET-Laufzeitbibliothek abgestimmte Funktionen angeboten.

5.5 Laden und Speichern eines persistenten PROSET-Werts

Im folgenden wird nun erläutert, wie die persistenten PROSET-Werte geladen und gespeichert werden. Da die persistenten PROSET-Werte höherer Ordnung eine sehr komplexe interne Struktur aufweisen, wird der Lade- und Speichervorgang bei diesen Werten nur übersichtsartig erläutert. Detailliertere Informationen würden den Rahmen der schriftlichen Ausarbeitung dieser Diplomarbeit sprengen. Der interessierte Leser kann die Details hierzu der Dokumentation der Persistenzschnittstelle in Form von ausführlichen Kommentaren bzw. der Dokumentation der PROSET-Laufzeitbibliothek entnehmen.

5.5.1 Laden eines persistenten ProSet-Werts

In der Persistenzschnittstelle werden folgende Funktionen angeboten, um einen persistenten ProSet-Wert zu laden:

- `lg_proset_value_read`: Lädt einen persistenten ProSet-Wert und führt dabei auch ein Auditing durch, falls das entsprechende Flag der Schnittstelle aktiviert wurde. Diese Funktion soll in erster Linie von Werkzeugen benutzt werden.
- `ppl_read`: Ruft implizit `lg_proset_value_read` auf und wird vom ProSet-Compiler benutzt. Die Funktion generiert in einem Fehlerfall Ausnahmen, die innerhalb eines ProSet-Programms behandelt werden können.

Innerhalb eines ProSet-Programms wird immer auf eine Kopie eines persistenten ProSet-Werts gearbeitet, d.h. es wird aufgrund der in H-PCTE gespeicherten Informationen und mit Hilfe der über die Laufzeitbibliothek von ProSet angebotenen Funktionen eine Kopie des in H-PCTE abgelegten persistenten ProSet-Werts erzeugt. Diese Kopie wird dann mit Hilfe der über die ProSet-Laufzeitbibliothek zur Verfügung gestellten effizienten Manipulationsoperationen bearbeitet.

5.5.1.1 Laden eines persistenten ProSet-Werts höherer Ordnung

Da persistente ProSet-Werte höherer Ordnung eine Programmeinheit mit Bürgerrechten erster Klasse in ProSet darstellen und H-PCTE nur strukturell-objektorientiert ist, muß der Objekt-Code des entsprechenden ProSet-Werts mit Hilfe eines dynamischen Linkers zum Programm geladen werden. Darüber hinaus müssen einige interne Strukturen rekonstruiert werden.

Unter Solaris 1 (SunOS 4.1) wird der Objekt-Code über den Dld von Gnu zum ProSet-Programm geladen (siehe [WO90] und [Wil91]). Allerdings wird der Dld nicht weiterentwickelt, so daß insbesondere eine lauffähige Version für Solaris 2 fehlt. Da sowohl Solaris 1 als auch Solaris 2 *Shared Libraries* zur Verfügung stellen, wird ein Hinzuladen von Objekt-Code über diese Shared Librarys ermöglicht. Darüber hinaus unterstützen die Shared Librarys auch ein *Multithreading*, d.h. leichtgewichtige Prozesse. Somit dient eine solche Vorgehensweise einer späteren Erweiterung der Konzepte bzgl. der Parallelprogrammierung von ProSet.

Da das Laden von persistenten ProSet-Werten in Bezug auf den Hauptspeicherplatz speicherintensiv ist, wird implizit über Funktionen des Moduls `ps_storage` bei Speicherplatzmangel geprüft, ob geladene persistente ProSet-Werte höherer Ordnung existieren, die nicht mehr benötigt werden (ihr Referenzzähler ist dann auf 0 gesetzt). In diesem Fall wird der Speicherplatz mit Hilfe spezieller Funktionen des Dlds bzw. der Shared Library-Systembibliotheken freigegeben. Detailliertere Informationen zum Laden von persistenten ProSet-Werten höherer Ordnung kann man den Kommentaren in dem Modulen `ps_dynlink` und `ps_complex` entnehmen.

5.5.2 Speichern eines persistenten ProSet-Werts

In der Persistenzschnittstelle werden folgende Funktionen angeboten, um einen persistenten ProSet-Wert zu speichern:

- `lg_proset_value_write`: Speichert einen persistenten ProSet-Wert und führt dabei auch ein Auditing durch, falls das entsprechende Flag der Schnittstelle aktiviert wurde. Diese Funktion soll in erster Linie von Werkzeugen benutzt werden.
- `ppl_write`: Wird vom ProSet-Compiler benutzt und generiert in einem Fehlerfall Ausnahmen, die innerhalb eines ProSet-Programms behandelt werden können.

5.5.2.1 Ermitteln des Speicherungsmodus

Laufzeitanalysen der H-PCTE-Forschungsgruppe haben ergeben, daß die laufzeitintensivsten Funktionen das Erzeugen und Löschen von H-PCTE-Objekten sind. Um eine möglichst effiziente Speicherung erzielen zu können, um also insbesondere nicht unnötigerweise Objekte zu löschen und wieder zu erzeugen, wird deshalb vor der Speicherung von ProSet-Werten eine Funktion `lg_creation_mode_get` aufgerufen, die nach dem in Abbildung 5-1 vorgestellten Ablaufplan einen *Speicherungsmodus* ermittelt.

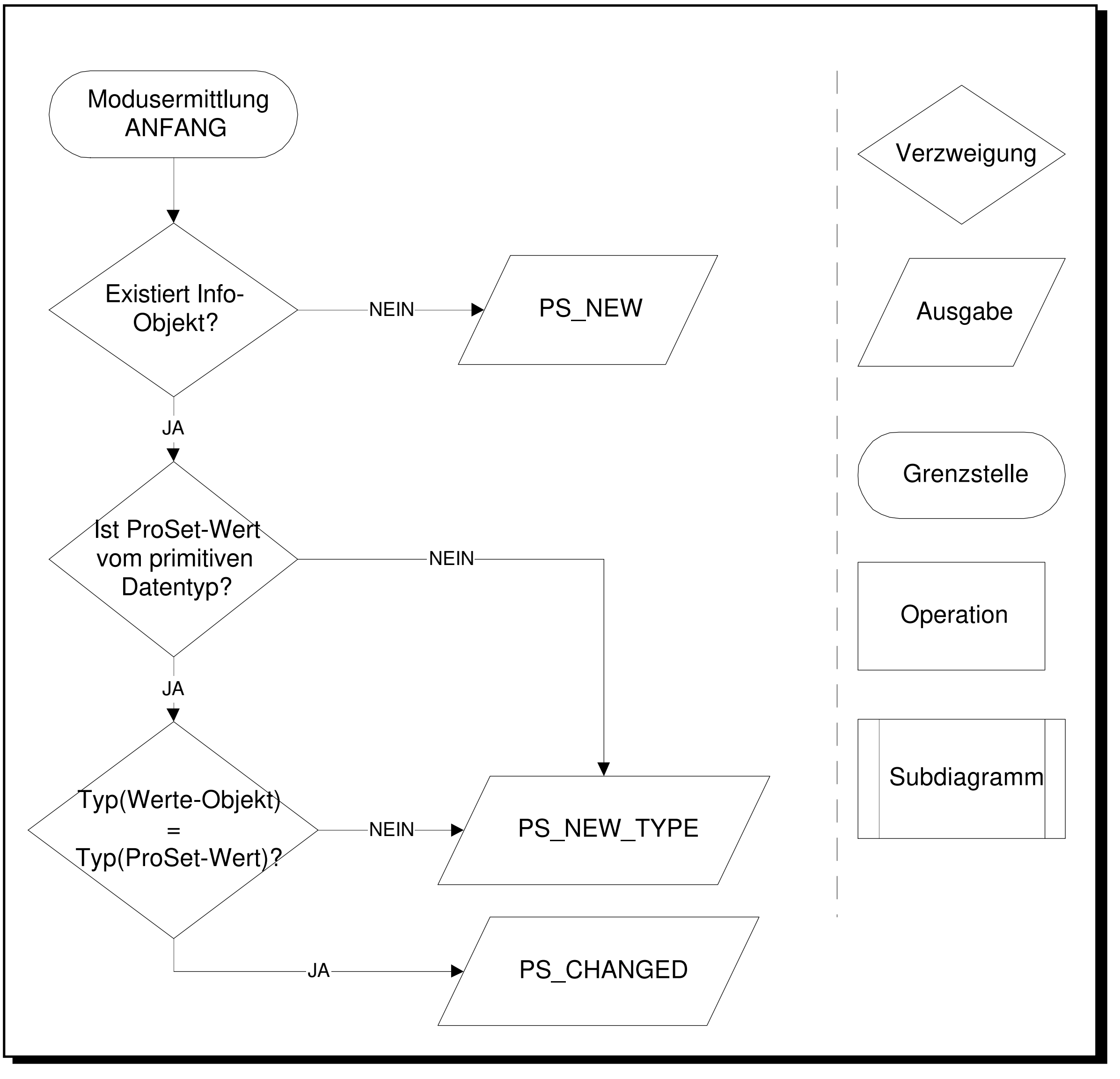


Abbildung 5-1: Ablaufplan zur Ermittlung eines Speicherungsmodus

Hierbei zeigen die von der Funktion zurückgegebenen Werte folgendes an:

- `PS_NEW`: Das PROSET-Objekt existiert noch nicht in der Objektbank, d.h. später muß sowohl das Info-Objekt als auch das Werte-Objekt des PROSET-Objekts gespeichert werden.
- `PS_NEW_TYPE`: Das PROSET-Objekt existiert zwar, aber sein Typ hat sich geändert, d.h. später kann das Info-Objekt bestehen bleiben aber das Werte-Objekt muß gelöscht, ein Werte-Objekt mit dem neuen Typ angelegt und sein Attribut neu gesetzt werden.
- `PS_CHANGED` (nur für primitive persistente PROSET-Werte): Das PROSET-Objekt existiert mit dem gleichen Typ, d.h. später muß nur das Attribut seines Werte-Objekts neu gesetzt werden.

Ein Kritikpunkt ist sicherlich, daß dieser Prüfvorgang im Worst-Case einen gewissen Overhead mit sich bringt, der dazu führen kann, daß diese Vorgehensweise einen, wenn auch nur geringen Effizienzverlust verursacht. Dieser Worst-Case tritt auf, wenn nach einem zusätzlichen Auslesen des Objekttyps des existierenden Werte-Objekts festgestellt wird, daß sich der Typ des persistenten PROSET-Werts geändert hat und somit doch das Werte-Objekt gelöscht werden muß. Hierbei wird jedoch insbesondere auch aufgrund der in der Praxis erzielten Ergebnisse der am Lehrstuhl Software-Technologie der Universität Dortmund durchgeführten Projektgruppe [PG95] davon ausgegangen, daß die Wahrscheinlichkeit eines Typwechsels eher gering ist.

5.5.2.2 Speichern von primitiven PROSET-Werten

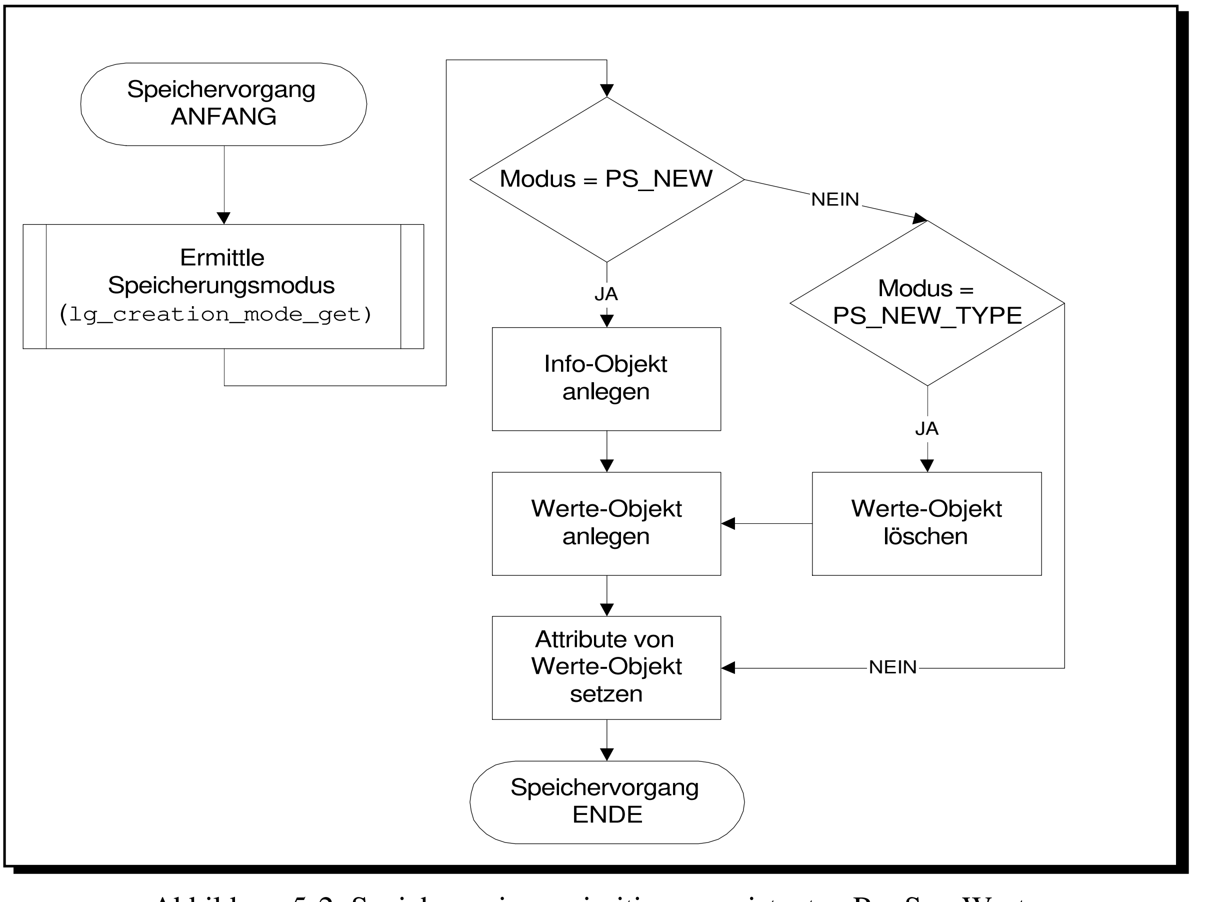


Abbildung 5-2: Speichern eines primitiven persistenten PROSET-Werts

Das Werte-Objekt eines primitiven PROSET-Objekts besteht aus einem atomaren H-PCTE-Objekt. Aus diesem Grunde kann man hier die oben formulierte Strategie wie über den Ablaufplan in beschrieben Abbildung 5-2 realisieren.

5.5.2.3 Speichern von komplexen persistenten PROSET-Werten

Ein komplexer persistenter PROSET-Wert umfaßt zusammengesetzte PROSET-Werte und PROSET-Werte höherer Ordnung. Das Werte-Objekt eines komplexen PROSET-Objekts besteht aus einem komplexen H-PCTE-Objekt. Es wäre auch hier möglich, die oben formulierte Strategie zu realisieren, jedoch wird die Strategie zur Zeit nur abgeschwächt angewandt: Wenn festgestellt wird, daß das PROSET-Objekt existiert, bricht die Funktion `lg_creation_mode_get` (siehe 5.5.2.1) ab und liefert den Modus `PS_NEW_TYPE`, so daß immer das Werte-Objekt zwischenzeitlich gelöscht wird.

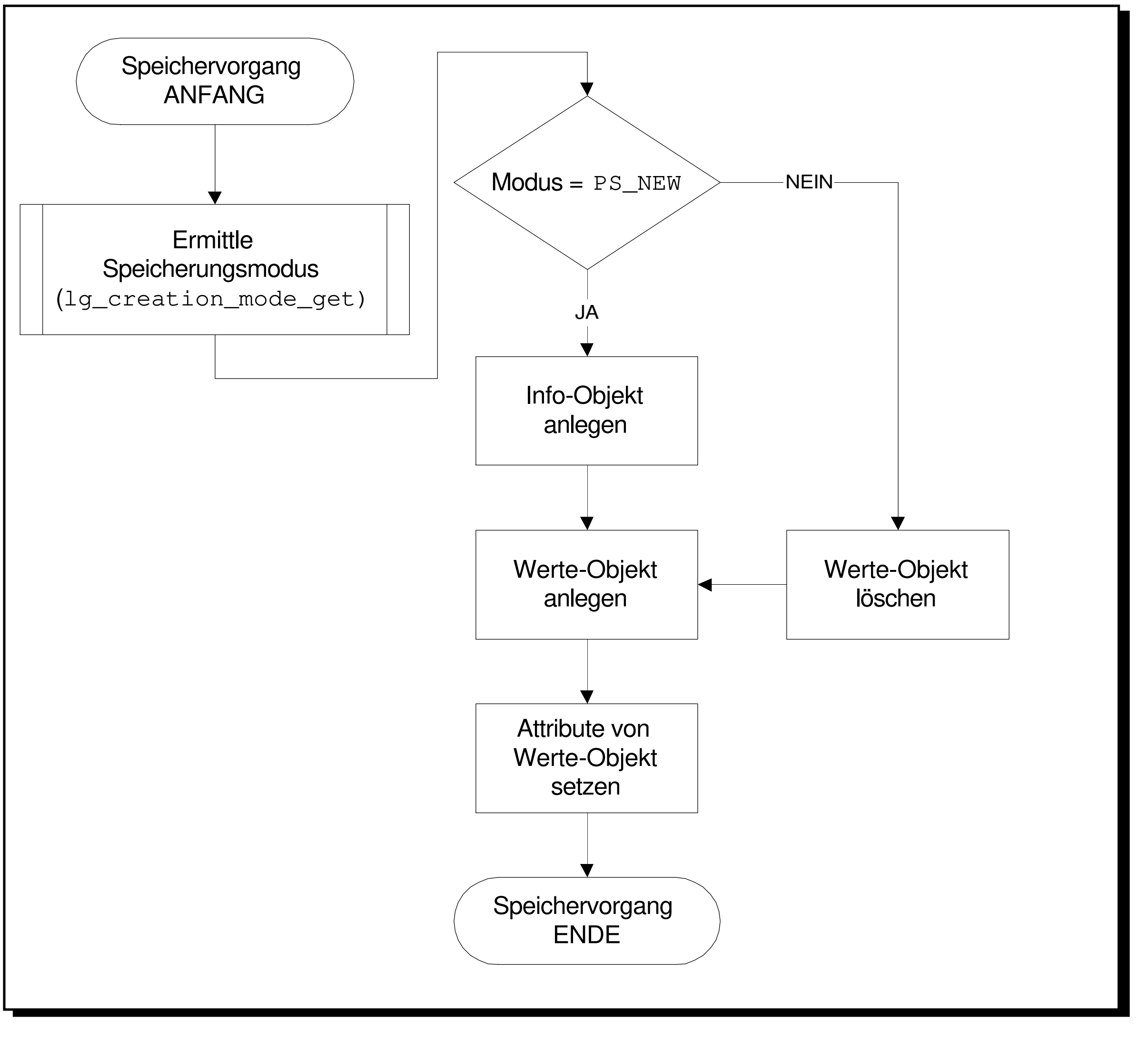


Abbildung 5-3: Speichern eines komplexen persistenten PROSET-Werts

Die obige Strategie wird abgeändert, da die Wahrscheinlichkeit bei zusammengesetzten persistenten PROSET-Werten und insbesondere bei PROSET-Werten höherer Ordnung sinkt, daß die Komponentenobjekte unverändert übernommen werden können. Allerdings haben die praktischen Erfahrungen, welche die oben erwähnte Projektgruppe [PG95] mit PROSET gesammelt hat, ergeben, daß zusammengesetzte ProSet-Werte meistens benötigt werden, um Record-Strukturen nachzubilden. Somit weisen sie eine reguläre Struktur auf, die der Strategie entgegenkäme. In einem solchen Fall kann vorausgesetzt werden, daß an bestimmten Positionen im zusammengesetzten ProSet-Wert nur immer ein ProSet-Wert von einem bestimmten (sich nicht ändernden) Typ vorkommen würde. Eine Erweiterung der Strategie in dieser Hinsicht ist jedoch implementierungstechnisch zu aufwendig, so daß dies als mögliche Erweiterung der Diplomarbeit angesehen werden kann. Der Speichervorgang eines komplexen ProSet-Werts läuft dann wie in Abbildung 5-3 beschrieben ab.

5.5.2.4 Gemeinsame Komponenten

In ProSet können in bestimmten Situationen und abhängig vom jeweiligen Datentyp identische Werte auftreten, die dann zur Minimierung der Speicherplatzkosten zu *gemeinsamen Komponenten (Shared Components)* zusammengefaßt werden sollten. Gemeinsame Komponenten können durch entsprechende Zuweisungen oder durch Ausführung bestimmter Operationen nur im Verlaufe genau eines ProSet-Programms entstehen.

Eine erste Klasse von gemeinsamen Komponenten entsteht bei Zuweisungen, wie anhand der Abbildung 5-4 verdeutlicht wird:

```
program demo;
  persistent A, B : "PFile1";
begin
     A := B;
end demo;
```

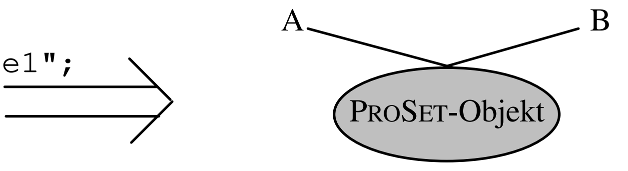


Abbildung 5-4: Gemeinsame Komponenten

Aufgrund der Wertsemantik der Zuweisungen in ProSet, kann eine Speicherung zweier persistenter ProSet-Werte A und B, wobei einem ProSet-Wert der andere zugewiesen wurde, problematisch sein. Bei Änderung eines Zugriffsrechts auf A durch seinen Besitzer wird somit implizit auch das Zugriffsrecht auf B geändert. In einem solchen Fall erweist es sich allerdings als vorteilhaft, daß ein ProSet-Objekt aus zwei Komponenten besteht. Eine Lösung wäre hier, nur das Werte-Objekt als gemeinsame Komponente zu modellieren und die Zugriffsrechte im jeweiligen Info-Objekt zu verwalten. Aber auch diese Vorgehensweise ist unter dem Gesichtspunkt des Schutzkonzepts von ProSet problematisch, weil man erstens so einen Zugriffsschutz durch Gebrauch der H-PCTE-API umgehen kann und zweitens ein Benutzer aufgrund des Zugriffsrechts, das ihm das erste Info-Objekt gewährt, das Werte-Objekt verändern kann, obwohl das zweite Info-Objekt ihm diesen Zugriff verwehrt. Eine Speicherung von gemeinsamen Komponenten kann also nur erfolgreich sein, wenn dies für den Benutzer transparent ist, was bzgl. des Schutzkonzepts nicht der Fall ist.

Aus diesen Gründen werden keine identischen ProSet-Werte, die durch Zuweisungen entstanden sind, über eine gemeinsame Komponente gespeichert. Diese Thematik kann in einem anderen Licht erscheinen, wenn ProSet eine Referenzsemantik einführt und darüber hinaus für jeden persistenten ProSet-Wert die Identität speziell sichert, was zur Zeit noch nicht der Fall ist (siehe Abschnitt 3.2.3 ab Seite 28).

Eine zweite Klasse von gemeinsamen Komponenten entsteht bei der Speicherung von ProSet-Werten vom Typ `instance`. Da es bei diesen ProSet-Werten nötig ist, die in der Umgebung des Moduls vorhandenen ProSet-Werte mit abzuspeichern und die statischen Nachfolger des Moduls zu speichern, können hierbei ebenfalls gemeinsame Komponenten auftreten. Dieser Fall kann eintreten, wenn in einem Modul mehrere Prozeduren definiert sind, und sie sowohl als statische Nachfolger des Moduls als auch innerhalb seiner Umgebung gespeichert werden. Eine solche Speicherung von gemeinsamen Komponenten ist für den Benutzer transparent, so daß hier ohne Probleme Speicherplatz eingespart werden kann.

5.6 Die Schnittstelle zum ProSet-Compiler

Es werden jeweils zu Beginn der Ausführung einer Programmeinheit alle in dieser Programmeinheit deklarierten persistenten ProSet-Werte aus H-PCTE geladen und nach Beendigung der entsprechenden Programmeinheit alle ihre persistenten ProSet-Werte wieder zurück nach H-PCTE gespeichert, wobei die innerhalb einer Programmeinheit ausgeführten Operationen über eine Transaktion geschützt werden.

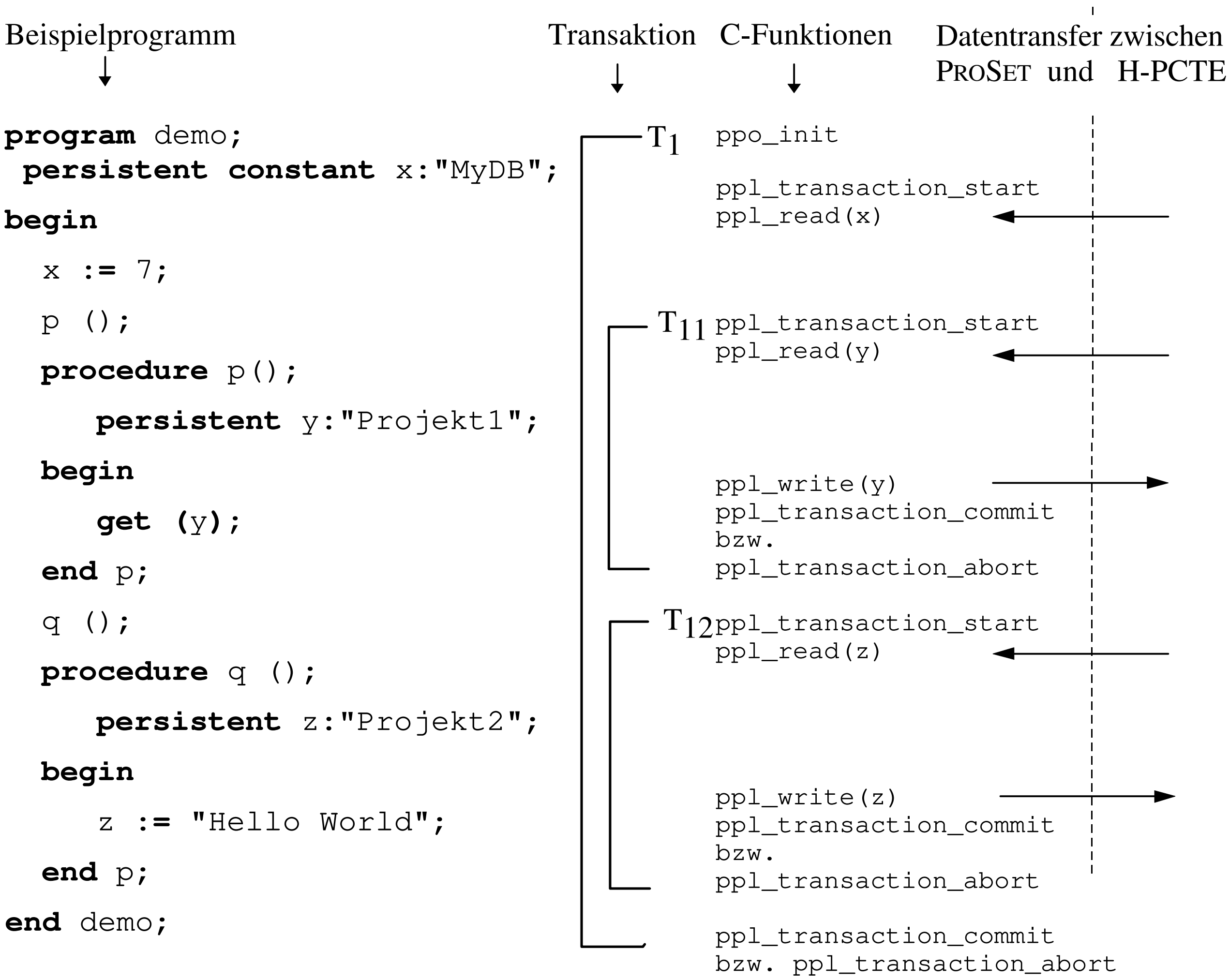


Abbildung 5-5: Ablauf eines ProSet-Programms, das die Persistenz benutzt

Der Compiler von ProSet generiert zu dem ANSI-C-Code Funktionsaufrufe von Operationen, die über die Persistenzschnittstelle zur Verfügung gestellt werden. Dabei werden folgende Funktionen des Moduls `ps_persistent` benutzt:

- `ppo_init`: Startet einen H-PCTE-Prozeß, der mit dem H-PCTE-Server kommuniziert und führt diverse Initialisierungen durch.
- `ppl_scan`: Prüft die lexikalische Korrektheit des P-File-Namens und die Existenz des P-Files.
- `ppl_read`: Lädt einen persistenten ProSet-Wert in den Applikationsprozeß.
- `ppl_write`: Speichert einen persistenten ProSet-Wert nach H-PCTE.
- `ppl_transaction_start`: Startet eine Transaktion
- `ppl_transaction_commit`: Beendet die letzte gestartete (Sub-)Transaktion.
- `ppl_transaction_abort`: Bricht die letzte gestartete (Sub-)Transaktion ab.

Bei Ausführung einer Programmeinheit wird als erster Funktionsaufruf per `ppl_transaction_start` eine Transaktion gestartet, wobei intern entweder eine H-PCTE-Transaktion gestartet oder für eine Subtransaktion ein Sicherungspunkt gesetzt wird. Bei Beendigung einer Programmeinheit wird als letzte Funktion `ppl_transaction_commit` oder `ppl_transaction_abort` aufgerufen, abhängig davon, ob bei Ausführung von `ppl_read` oder `ppl_write` ein Fehler aufgetreten ist. Die obige Abbildung 5-5 illustriert, wie die Funktionen der Persistenzschnittstelle in das C-Programm eingebunden werden. Die rechte Spalte veranschaulicht graphisch, wie die persistenten ProSet-Werte dabei von H-PCTE nach ProSet hin und her transferiert werden.

Bei einem Fehler wird zur Unterstützung der Orthogonalität der Konzepte von ProSet eine Ausnahme erzeugt, auf die der Programmierer durch entsprechend definierte Handler reagieren kann (siehe 4.3.1 ab Seite 63).

Wie in Abschnitt 2.1.2 ab Seite 3 erläutert, wird ein ProSet-Programm in einer ersten Stufe in ein C-Programm übersetzt. Um hier das Transaktionskonzept geeignet realisieren zu können, werden jeweils am Ende einer C-Funktion, in die eine ProSet-Programmeinheit übersetzt wurde, zwei Sprungmarken `COMMIT` und `ABORT` definiert. Zur Sprungmarke `COMMIT` wird verzweigt, wenn der Handler, der für eine innerhalb einer Transaktion generierten Ausnahme definiert wurde, mit `return commit` beendet wurde oder die Transaktion ordnungsgemäß ausgeführt werden konnte. Zur Sprungmarke `ABORT` wird verzweigt, wenn ein entsprechender Handler mit `return` beendet wurde. Ab einer solchen Marke werden dann Funktionen für diverse Aufräumarbeiten, wie das Freigeben von nicht mehr benötigten Speicher, aufgerufen und letztendlich wird die passende Funktion `ppl_transaction_commit` oder `ppl_transaction_abort` ausgeführt.

5.7 Implementierung des Schutzkonzeptes

In diesem Abschnitt wird erläutert, wie das Schutzkonzept von ProSet implementiert wurde. Zur Realisierung des Schutzkonzepts von ProSet stellt das Modul `ps_security` verschiedene Funktionen zur Verfügung.

5.7.1 Initialisierungen beim Applikationsstart

Beim Start des Applikationsprozesses wird über eine Funktion `sec_application_init` das Modul `ps_security` initialisiert. Unter anderem werden dabei

- über spezielle Betriebssystemfunktionen von UNIX der Name des Benutzers ermittelt, in dessen Auftrag der Prozeß arbeitet, um so z.B. das `owner_name`-Attribut eines neu erzeugten ProSet-Objekts setzen zu können;
- die Umgebungsvariable `PS_ADOPTED_GROUP` ausgelesen und die entsprechende Benutzergruppe als Subjekt aktiviert (ist die Umgebungsvariable nicht gesetzt, wird die in H-PCTE spezifizierte Default-Gruppe des Benutzers aktiviert);
- die Prozeß-Default-ACL mit folgenden Einträgen belegt:
  1. einen Eintrag für den Benutzer des Prozesses auf Basis der in der Umgebungsvariable `PS_ACCESS_MASK` für den Benutzer spezifizierten ProSet-Rechtemaske,
  2. einen Eintrag für die adoptierte Benutzergruppe auf Basis der in der Umgebungsvariable `PS_ACCESS_MASK` für die Gruppe spezifizierte ProSet-Rechtemaske, und
  3. einen Eintrag für die Benutzergruppe `systemadministrator`, für die immer alle Rechte gesetzt sind.

5.7.2 Typwechsel eines persistenten ProSet-Werts

Wie bereits in Abschnitt 4.2.2 ab Seite 57 beschrieben, ensteht ein Problem, wenn der persistente ProSet-Wert seinen Typ ändert. In diesem Fall wird das Werte-Objekt des ProSet-Objekts gelöscht und mit dem entsprechenden neuen Typ neu angelegt. Das Problem ist die Rekonstruktion der ACL, die beim Löschen des Werte-Objekts verloren geht. Aus diesem Grunde wird, nachdem ein Speicherungsmodus `PS_NEW_TYPE` festgestellt wurde, die ACL des Info-Objekts ausgelesen und die ACL jedes neu erzeugten Komponentenobjekts mit dieser ACL belegt. Um diesen Vorgang effizienter zu gestalten, wird die aktuelle Prozeß-Default-ACL in einer modullokalen Variablen gespeichert und alle Zugriffsrechte in der Prozeß-Default-ACL auf undefiniert(?) gesetzt. Eine Ausnahme bildet hier der Eintrag des Besitzers. Dieser bleibt bestehen, da sonst die Prozeß-Default-ACL selbst nicht mehr geändert werden kann. Dann werden auf Basis der ACL des Info-Objekts entsprechende Einträge in die Prozeß-Default-ACL vorgenommen. Somit werden die neu erzeugten Komponentenobjekte automatisch bei ihrer Erzeugung mit der entsprechenden ACL versehen. Nach Beendigung des Speichervorgangs wird dann die Prozeß-Default-ACL auf ihren ursprünglichen Inhalt zurückgesetzt.

5.7.3 Adoptieren einer Benutzergruppe

Man hat prinzipiell drei Möglichkeiten eine Benutzergruppe zu aktivieren:

1. Man trägt vor dem Start eines ProSet-Programms den entsprechenden Gruppennamen in die Umgebungsvariable PS_ADOPTED_GROUP ein.
2. Man setzt über die Funktion `tl_default_group_change` den von H-PCTE speziell zur Verfügung gestellten Link auf die neue Defaultgruppe um. Diese Benutzergruppe wird dann beim Start eines Applikationsprozesses dieses Benutzers implizit adoptiert. Voraussetzung ist dann aber, daß die obige Umgebungsvariable nicht gesetzt ist.

3. Die Funktion `cmd_group_change` kann über ein ProSet-Modul angeboten werden, so daß man die Möglichkeit besitzt, explizit durch Aufruf dieser Funktion die Benutzergruppe zu wechseln.

5.7.4 Auditingfunktionen

Das Auditing kann über zwei Funktionen `adg_auditing_on` und `adg_auditing_off` des Moduls `ps_auditing` an- und ausgeschaltet werden. Diese Funktionen sollte jedoch nur ein Systemadministrator ausführen dürfen.

5.8 Fehlerbehandlung in der Persistenzschnittstelle

Da die Persistenzschnittstelle Funktionen sowohl für ProSet-Programme als auch für verschiedene Werkzeuge anbietet, werden erst in der obersten Schicht der Schnittstelle zum ProSet-Programm bei einer Fehlersituation Ausnahmen generiert. Denn nur ein ProSet-Programm kann auf die diversen Ausnahmen geeignet reagieren, wogegen Werkzeuge eher unabhängig von der Ausnahmebehandlung von ProSet arbeiten wollen. Deshalb werden, angefangen von der untersten Schicht, die von H-PCTE bei Ausführung einer seiner API-Funktionen erzeugten Fehlercodes und die innerhalb der Persistenzschnittstelle erzeugten Fehlercodes in die überliegenden Schichten propagiert, bis letztendlich in der obersten Schicht in Form des Moduls `ps_persistent` die ProSet-Ausnahmen generiert werden. Hier können dann die in Tabelle 5-2 und aufgelisteten Ausnahmen auftreten:

Ausnahme	Modus	erzeugt in	Grund	Parameter
p_illegal_p_file_name	escape	ppl_scan	String, der P-File identifiziert, verletzt die lexikalische Konvention	(RD p_file_path)
p_missing_p_file	escape	ppl_scan	P-File existiert nicht in der Objektbank	(RD p_file_path)
p_type_mismatch	escape	ppl_scan	falscher Typ des Namenparameters	
p_missing_constant	escape	ppl_read	persistenter ProSet-Wert ist als Konstante deklariert, aber nicht in der Objektbank vorhanden	(RD identifier)
p_missing_name	notify	ppl_read, ppl_write	persistenter ProSet-Wert existiert nicht in der Objektbank, bei Beendigung des Handlers mit `resume <value>` wird ein ProSet-Wert `value` implizit angelegt.	(RD identifier)

Tabelle 5-1 Übersicht über die in der Persistenzschnittstelle generierten Ausnahmen (Teil 1)

Ausnahme	Modus	erzeugt in	Grund	Parameter
p_deadlock_occured	escape	ppl_read, ppl_write	Die innerhalb der (Sub-) Transaktion gesetzten Sperren haben ein Deadlock verursacht.	keine
p_fatal_error	escape (return)	ppo_init, ppl_read, ppl_write	• Initialisierung oder Start eines H-PCTE-Clients ist fehlgeschlagen • Lesefehler in H-PCTE oder Fehler beim Laden des persistenten ProSet-Werts per DLD oder Shared Library • Schreibfehler in H-PCTE	keine
p_access_denied	escape (return)	ppl_read, ppl_write	Recht zur Ausführung der entsprechenden Operation wurde den im Auftrag des Benutzers agierenden Subjekten eines ProSet-Programms verwehrt.	(RD identifier)
p_segment_unloadable	escape (return)	ppl_scan	Das Segment des P-Files konnte nicht geladen werden	(RD p_file_path)

Tabelle 5-2: Übersicht über die in der Persistenzschnittstelle generierten Ausnahmen (Teil 2)

6 Schlußbemerkungen

In diesem Abschnitt werden eine Zusammenfassung der Arbeit und mögliche Erweiterungen der Diplomarbeit dargestellt.

6.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurden Persistenzkonzepte für den Schutz- und die Verteilung persistenter ProSet-Werte aufgestellt. Darüber hinaus wurde das Transaktionskonzept von ProSet integriert und somit ein Mehrbenutzerbetrieb auf persistente ProSet-Werte realisiert. Hier sei angemerkt, daß die Funktionen der Persistenzschnittstelle noch nicht im Rahmen eines Mehrbenutzerbetriebs getestet werden konnten, weil die von H-PCTE zur Verfügung gestellten API-Funktionen zum Setzen von ACL-Einträgen noch fehlerhaft sind. Eine korrekte Version konnte bisher noch nicht zur Verfügung gestellt werden. Desweiteren können alle ProSet-Werte mit Bürgerrechten erster Klasse persistent gemacht werden. Spezielle Operationen, die in einem ProSet-Modul gekapselt werden können, werden innerhalb der Persistenzschnittstelle angeboten. Außerdem bietet die Persistenzschnittstelle auf Administrationswerkzeuge abgestimmte Funktionen zur Benutzer- und Benutzergruppenverwaltung, zur Unterstützung des Verteilungskonzepts, sowie zur Verwaltung der persistenten ProSet-Werte an.

Obwohl die über die Persistenzschnittstelle angebotenen mächtigen Funktionen bereits werkzeugähnlichen Charakter haben, sind noch diverse Werkzeuge zur geeigneten Unterstützung der Persistenzkonzepte notwendig.

Im folgenden wird erläutert, welche Erweiterungen der Diplomarbeit vorgenommen werden sollten bzw. können.

6.2 Erweiterungen

Innerhalb der einzelnen Abschnitte wurden bereits mögliche Erweiterungen der Persistenzkonzepte bzw. der Implementierung angesprochen. Zusammengefaßt sind dies folgende Erweiterungen:

- Es werden Werkzeuge benötigt, welche die Benutzung des Schutzkonzepts und des Verteilungskonzepts von ProSet weiter automatisieren.
- Die Persistenzkonzepte sollten auf parallele ProSet-Programme erweitert werden. Dies betrifft primär das Transaktionskonzept von ProSet.
- Es sind Funktionen zur Versionierung von persistenten ProSet-Werten wünschenswert.
- Zur Formulierung von Integritätsbedingungen sollten weitere Persistenzkonzepte aufgestellt werden.
- Zur Verbesserung der Speicherplatzausnutzung sollten gemeinsame Komponenten eingeführt werden. Eine solche Erweiterung ist jedoch schwierig, da sie das Schutzkonzept und das Verteilungskonzept von ProSet beeinflussen. Ebenfalls muß dann die Identität aller ProSet-Werte gesichert werden.

- Um insbesondere die Verwaltung persistenter ProSet-Werte und (Sub-)P-Files zu unterstützen, sollten Funktionen angeboten werden, um bestimmte Anfragen auf den Objektbankinhalt ausführen zu können. Da inzwischen H-PCTE über NTT [Haa94] und P-OQL [Hen95] zwei Anfragesprachen anbietet, stellt dies eine praktikable Erweiterung dar.
- Die innerhalb des schriftlichen Teils der Diplomarbeit dargelegten Optimierungsmöglichkeiten bzgl. des Laufzeitverhaltens der Funktionen der Persistenzschnittstelle sowie der Speicherplatzausnutzung sollten realisert werden. Dies betrifft insbesondere die Speicherung von gemeinsamen Komponenten. Hierzu sollten jedoch zuerst diverse Messungen der Laufzeit und des Speicherplatzbedarfs vorgenommen werden.

6.3 Wünschenswerte Erweiterungen von H-PCTE

Die über H-PCTE angebotenen Funktionen haben sich als sehr leistungsstark und flexibel bei der Implementierung der Persistenzkonzepte von ProSet erwiesen. Jedoch wäre insbesondere in Bezug auf das Transaktionskonzept von ProSet und eine spätere Erweiterung der Persistenzkonzepte auf parallele ProSet-Programme wünschenswert, daß ein geschachteltes Prozeßmodell angeboten wird. Aus Gründen einer Reduzierung der Speicherplatzkosten wäre es ebenfalls hilfreich, eine Funktion angeboten zu bekommen, über die nicht mehr benötigte Sicherungspunkte von H-PCTE wieder freigegeben werden können. Desweiteren werden effiziente Funktionen auf komplexe Objekte benötigt, wie z.B. das Setzen der ACLs der Komponentenobjekte eines komplexen Objekts oder das Kopieren von komplexen Objekten.

Anhang A. Das SDS zur Verwaltung persistenter ProSet-Werte

```
sds proset:
--**************************************************
-- Importierte Objekt- und Attributtypen
--**************************************************
import objecttype security-user_group as user_group;
import objecttype security-user as user;
import objecttype system-object as object;
import objecttype hpcte-home as home;
import attributetype system-system_key;
import attributetype system-name as name;
import attributetype system-number as number;

--**************************************************
-- Attribute der primitiven Datentypen
--**************************************************
attributetype boolean_value : boolean := false;
attributetype integer_value : integer := +0;
attributetype string_value  : string := "";
attributetype real_value    : string := "0.0";
attributetype atom_value    : string := "";

--**************************************************
-- PS_OBJECT
--**************************************************
-- Basisobjekttyp fuer persistente ProSet-Werte
objecttype ps_object : child type of object;

--**************************************************
-- BASIS_OBJECT
--**************************************************
objecttype basis_object : child type of object
with
  attribute
     creation_time      : time;
     owner_name         : string := "";
end basis_object;
--**************************************************
-- AUDIT_OBJECT
--**************************************************
objecttype audit_object : child type of basis_object
with
  attribute
     change_time        : time;
     last_change_by     : string := "";
end audit_object;

-- ************************************************
-- Author:       J. Heinrich, Ch. Kappert
-- Description:  PRIMITIVE PROSET-WERTE
-- ************************************************
-- PS_BOOLEAN
objecttype ps_boolean : child type of ps_object
with
  attribute
    boolean_value;
end ps_boolean;
```

```
-- PS_INTEGER
objecttype ps_integer : child type of ps_object
with
  attribute
    integer_value;
end ps_integer;

-- PS_STRING
objecttype ps_string : child type of ps_object
with
  attribute
    string_value;
end ps_string;

-- PS_REAL
objecttype ps_real : child type of ps_object
with
  attribute
    real_value;
end ps_real;
-- PS_ATOM
objecttype ps_atom : child type of ps_object
with
  attribute
    atom_value;
end ps_atom;

-- PS_OMEGA
objecttype ps_omega : child type of ps_object;

-- ***************************************************
-- Author:       Ch. Kappert
-- Description:  PROSET-WERTE ZWEITER KLASSE
-- ***************************************************

-- ***************************************************
-- UNIT
-- ***************************************************
linktype has_nested_unit : composition link (number) to unit;

objecttype unit : child type of ps_object
with
  attribute
      extern_name  : string  := "";
      profile      : string  := "";
      is_dld_code  : boolean := false;
      code_size    : integer := +0;
      object_code  : string  := "";
  component
     has_nested_unit;
end unit;

-- ***************************************************
-- HANDLER
-- ***************************************************
objecttype ps_handler : child type of unit;

-- ***************************************************
-- PROCEDURE
-- ***************************************************
objecttype ps_procedure : child type of unit;
```

```
-- ************************************************
-- MODULE
-- ************************************************
objecttype ps_module : child type of unit;

-- ************************************************
-- Author:      Ch. Kappert
-- Description: PROSET-WERTE HOEHERER ORDNUNG
-- ************************************************

-- ************************************************
-- CLOSURE_UNIT
-- ************************************************
-- Eingefrorene Umgebung
linktype has_frozen_env : composition link (name)
        to  ps_set, ps_tuple, ps_boolean, ps_integer,
            ps_string, ps_real, ps_atom, ps_omega,
            ps_procedure, ps_handler, ps_module,
            ps_function, ps_modtype, ps_instance;

linktype has_identifier : composition link to ps_atom;

-- Objekttyp closure_unit
objecttype closure_unit : child type of unit
with
  attribute
        pred_env_size : integer := +0;
  component
        has_identifier;
        has_frozen_env;
end closure_unit;
-- ************************************************
-- FUNCTION
-- ************************************************
objecttype ps_function : child type of closure_unit;

-- ************************************************
-- MODTYPE
-- ************************************************
objecttype ps_modtype : child type of closure_unit;

-- ************************************************
-- INSTANCE
-- ************************************************
linktype has_environment : composition link (number)
             to   ps_set,
                  ps_tuple,
                  ps_boolean,
                  ps_integer,
                  ps_string,
                  ps_real,
                  ps_atom,
                  ps_omega,
                  ps_procedure,
                  ps_handler,
                  ps_module,
                  ps_function,
                  ps_modtype,
                  ps_instance;
```

```
-- Objekttyp ps_instance
objecttype ps_instance : child type of ps_modtype
with
  attribute
        export_list : string := "";
  component
        has_environment;
end ps_instance;



-- ***********************************************
-- Author:      J. Heinrich, Ch. Kappert
-- Description: ZUSAMMENGESETZTE PROSET-WERTE
-- ***********************************************

-- ***********************************************
-- SET
-- ***********************************************
objecttype ps_set : child type of ps_object;

linktype set_element : composition link (number)
      to    ps_set, ps_tuple, ps_boolean, ps_integer,
            ps_string, ps_real, ps_atom, ps_function,
            ps_modtype, ps_instance;

extend objecttype ps_set
with
  component
     set_element;
end ps_set;

-- ***********************************************
-- TUPLE
-- ***********************************************
objecttype ps_tuple : child type of ps_object;

linktype tuple_element : composition link (number)
      to    ps_set, ps_tuple, ps_omega, ps_boolean,
            ps_integer, ps_string, ps_real,
            ps_atom, ps_function, ps_modtype, ps_instance;

extend objecttype ps_tuple
with
  component
    tuple_element;
end ps_tuple;

-- ***********************************************
-- Author:        Ch. Kappert
-- Description:   INFO-OBJEKT
-- ***********************************************
linktype is_value_of : implicit link to info_object
                   reverse has_value;

linktype has_value : composition link
      to    ps_set, ps_tuple, ps_omega, ps_boolean, ps_integer,
            ps_string, ps_real, ps_atom, ps_function,
            ps_modtype,ps_instance
     reverse is_value_of;
```

```
objecttype info_object : child type of audit_object
with
  component
      has_value;
end info_object;

extend objecttype ps_set
with
      link is_value_of;
end ps_set;

extend objecttype ps_tuple
with
      link is_value_of;
end ps_tuple;

extend objecttype ps_omega
with
      link is_value_of;
end ps_omega;

extend objecttype ps_boolean
with
      link is_value_of;
end ps_boolean;

extend objecttype ps_integer
with
      link is_value_of;
end ps_integer;

extend objecttype ps_real
with
      link is_value_of;
end ps_real;

extend objecttype ps_string
with
      link is_value_of;
end ps_string;

extend objecttype ps_function
with
      link is_value_of;
end ps_function;


extend objecttype ps_modtype
with
      link is_value_of;
end ps_modtype;

extend objecttype ps_instance
with
      link is_value_of;
end ps_instance;

extend objecttype ps_atom
with
      link is_value_of;
end ps_atom;
```

```
-- ***********************************************
-- Author:      Ch. Kappert
-- Description: P-Files und Sub-P-Files
-- ***********************************************

-- ***********************************************
-- PFILE
-- ***********************************************
-- Eintrag eines P-Files
linktype has_p_file_entry : composition link (name) to  info_object
                             reverse is_p_file_entry_of;

linktype is_p_file_entry_of : implicit link to p_file_node
                             reverse has_p_file_entry;

-- P-File (INFO-OBJEKT)
objecttype p_file_info : child type of basis_object
with
  attribute
        public        : boolean := true;
        on_segments   : string  := "";
end p_file_info;

-- P-File (KNOTEN-OBJEKT)
objecttype p_file_node : child type of object
with
  component
      has_p_file_entry;
end p_file_node;

-- ProSet-Objekt (INFO-OBJEKT) erweitern
extend objecttype info_object
with
  link
    is_p_file_entry_of;
end info_object;

linktype is_p_file_of : implicit link
      to p_store reverse contains_p_file;

linktype is_sub_p_file_of : implicit link
      to p_file_info reverse has_sub_p_file;

-- ***********************************************
-- Sub-P-File
-- ***********************************************
linktype has_sub_p_file : composition link (name)
      to p_file_info reverse is_sub_p_file_of;

linktype has_node : composition link to p_file_node
                    reverse is_node_of;

linktype is_node_of : implicit link to p_file_info reverse has_node;

-- Erweiterung des P-File-Knoten-Objekts
extend objecttype p_file_node
with
  link
    is_node_of;
end p_file_node;
```

```
-- Erweiterung des P-Files
extend objecttype p_file_info
with
  link
    is_sub_p_file_of;
    is_p_file_of;
  component
    has_sub_p_file;
    has_node;
end p_file_info;

-- ***************************************************
-- Author:      Ch. Kappert
-- Description: Administrationsobjekte
-- ***************************************************
-- Home-Objekt
-- ***************************************************
linktype has_proset_home : existence link to p_store
                     reverse is_proset_home_of;

linktype is_proset_home_of : implicit link (system_key) to home
                     reverse has_proset_home;

extend objecttype home
with
  link
    has_proset_home;
end home;

linktype contains_p_file : existence link (name) to p_file_info
                    reverse is_p_file_of;

-- ***************************************************
-- P-Store-Objekt
-- ***************************************************
objecttype p_store : child type of object
with
  attribute
    ps_adm_seg_id : string := "";
  link
    is_proset_home_of;
    contains_p_file;
end p_store;

-- ***************************************************
-- persistentes Wurzelobjekt
-- ***************************************************
objecttype persistent_root : child type of object;

-- Linktyp contains_p_store -- verbindet alles mit allem
linktype contains_p_store : existence link (name) to p_store
                  reverse is_p_store_of;

linktype is_p_store_of : implicit link to persistent_root
                  reverse contains_p_store;

extend objecttype persistent_root
with
  link
      contains_p_store;
end persistent_root;
```

```
extend objecttype p_store
with
    link
      is_p_store_of;
end p_store;


-- **************************************************
-- Directory fuer temporaere Objekte
-- **************************************************
linktype contains_tmp_object : existence link (name) to user_group,
                                                            user;

objecttype tmp_directory : child type of object
with
    link
      contains_tmp_object;
end tmp_directory;

linktype contains_tmp_dir : existence link to tmp_directory;

linktype proset_root : existence link to persistent_root;

extend objecttype object
with
    link
        contains_tmp_dir;
        proset_root;
end object;

end proset;
```

Literaturverzeichnis

[ABM88] M. P. Atkinson, P. Buneman, R. Morrison (Eds.). *Data Types and Persistence.* Springer Verlag, 1988.

[AFH94] O. Agesen, S. Frolund, M. Hoffmann Olsen. *Persistent object concepts.* In [KLLM94], pages 193-197, 1994.

[Atk91] M. P. Atkinson. *A vision of persistent systems.* In Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases. Lecture Notes in Computer Science, pages 453-459, Springer-Verlag, Berlin, 1991.

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven. *Prototyping - An Approach to Evolutionary System Development.* Springer-Verlag, Berlin, 1992.

[CD88] G. F. Coulouris, J. Dollimore. *Distributed Systems – Concepts and Design*, Addison-Wesley, 1988.

[Che76] P. P.-S. Chen. *The Entitiy Relationship Model: Toward a Unified View of Data,* ACM Transactions on Database Systems, Vol.1, pages 9-36, 1976.

[DDLC] The Data Definition Language Compiler, Dokumentation von H-PCTE.

[DF89] E.-E. Doberkat, D. Fox. *Software-Prototyping mit SETL.* Leitfäden und Monographien der Informatik, Teubner Verlag, 1989.

[DFG+92a] E.-E. Doberkat, W.Franke, U. Gutenbeil, W. Hasselbring, U. Lammers und C. Pahl. *ProSet - Prototyping with Sets: Language Definition.* Informatik-Bericht 02-92, Universität Essen, 1992.

[DFG+92b] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, C. Pahl. ProSet - *A Language for Prototyping with Sets.* In N. Kanopoulos (Ed.), *Proceedings of the Third International Workshop on Rapid System Prototyping*, Research Triangle Park, NC, pages 235-248, IEEE Computer Society Press, June 1992. (Auch erhältlich als Software Engineering Memo-15, Universität Essen, 1992).

[DFKS93] E.-E. Doberkat, W. Franke, U. Kelter, W. Seelbach. *Verwaltung persistenter Daten in einer Prototyping-Umgebung.* In H. Züllighoven, W. Altmann, E.-E. Doberkat (Hrsg.), *Requirements Engineering '93: Prototyping*, Berichte des German Chapter of the ACM, 41, Seiten 147-164, Stuttgart, Teubner Verlag, 1993.

[Dit86] K.R. Dittrich. *Object-Oriented Database Systems: the Notion and the Issues.* In K.R. Dittrich U. Dayal (Hrsg.): *Proceedings of the 1986 International Workshop on Object-Oriented Database Management Systems,* Pacific Grove, California (USA), 1986. IEEE, Computer Science Press.

[DMFV90] D. DeWitt, D. Maier, P. Futtersack, F. Velez. *A Study of three alternative Workstation-Server Architectures for object-oriented Systems.* In Proceedings of the 16th International Conference on Very Large Data Bases, pages 107-121, 1990.

[Dob92] Ernst-Erich Doberkat. *Integrating Persistence into a set-oriented Prototyping-Language.* In: Structured Programming, 13(3), Seiten 137-153, 1992.

[DTBS94] Asuman Dogac, M. Tamer Özsu, Alexandros Biliris, Timos Sellis. *Advances in Object-Oriented Database Systems.* Springer Verlag, NATO ASI Series, Series F: Computer and Systems Sciences, Vol.130, 1994.

[ECM93a] European Computer Manufacturers Association (Ed.). *Standard ECMA-149 – Portable Common Tool Environment (PCTE)*, Abstract Specification, Vol. 1, 2nd Edition, June 1993.

[ECM93b] European Computer Manufacturers Association (Ed.). *Standard ECMA-149 – Portable Common Tool Environment (PCTE)*, Abstract Specification, Vol. 2, 2nd Edition, June 1993.

[ECM93c] European Computer Manufacturers Association (Ed.). *Standard ECMA-158 – Portable Common Tool Environment (PCTE)*, C Programming Language Binding, 2nd Edition, June 1993.

[EN89] R. Elmasri, S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, 1989.

[ES89] G. Engels, W. Schäfer. *Programmentwicklungsumgebungen - Konzepte und Realisierung.* Reihe: Leitfäden der Angewandten Informatik. B.G. Teubner Stuttgart, 1989.

[Fra95] Wolfgang Franke. *Geschachtelte Transaktionen*, Internes Arbeitspapier des Lehrstuhls X, Software-Technologie, Universität Dortmund, 1995.

[Gr78] J.N. Gray. *Notes on Data Base Operating Systems.* in: Operating Systems – An Advanced Course, LNCS 60, 1978.

[GSD93] A. Geppert, S. Scherrer, K.R. Dittrich. *Transaction Management in CoOMS.* Technical Report 93.10, Forschungsbereich Datenbanktechnologie, Universität Zürich, April 93.

[Gus94] A. Gustavsson. *Persistence.* In [OOE94], pages 187-192, 1994.

[H94] Die H-PCTE-Forschungsgruppe. *H-PCTE vs. PCTE, Version 2.6*, Dokumentation zu H-PCTE, November 1994.

[Haa94] Oliver Haase. *NTT–Eine mengenorientierte algebraische Anfragesprache für PCTE.* Internes Memorandum 94/3, Fachbereich Elektrotechnik und Informatik, Universität-Gesamthochschule Siegen, Oktober, 1994.

[Has94] Wilhelm Hasselbring. *Prototyping Parallel Algorithms with a Set-Oriented Language*. Verlag Dr. Kovac, 1994.

[Hei95] Jörg Heinrich. *Entwurf und Integration verschiedener Werkzeuge zur Analyse und Verwaltung persistenter Daten einer Prototyping-Umgebung*, Diplomarbeit, Universität Dortmund, 1995.

[Hen95] Andreas Henrich. *P-OQL: an OQL-oriented Query Language for Pcte*, Dokumentation zu H-PCTE Version 2.6, 1995.

[Heu92] Andreas Heuer. *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme.* Addison-Wesley, 1992.

[Kel88] Udo Kelter: *Anforderungen an Zugriffsschutzmechanismen in Objektmanagementsystemen mit autonomen Benutzergruppen*, Internes Memorandum des Lehrstuhls Software-Technologie, Memo. 31, Nov. 1988.

[Kel90] Udo Kelter. *Group-Oriented Discretionary Access Controls for Distributed Structurally Object-Oriented Database Systems*, in *Proc. European Symposium on Research in Computer Security*, Toulouse, October 24-26, 1990; AFCET; 1990.

[Kel91] Udo Kelter. *Betriebssysteme- Kurseinheit 6: Sicherheit*. Unterlagen zur Vorlesung Betriebssysteme, Fernuniversität Gesamthochschule Hagen, 1991.

[Kel91c] Udo Kelter. *CASE*. Informatik-Spektrum, S.215ff, Bd. 14, 1991.

[Kel94] Udo Kelter. *Notizen zur Einführung in PCTE.* PCTE-Kurs vom 11.-15.7.1994, Universität-Gesamthochschule Siegen.

[KLLM94] I.L. Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson. *Object-oriented Environments: The Mjølner Approach.* Prentice Hall, 1994.

[Lin91] Frank Lindert. *Eine Prozeß- und Sperrenverwaltung für das Objektmanagementsystem H-PCTE.* Diplomarbeit, Universität Dortmund, 1991.

[LS87] P.C. Lockemann, J.W. Schmidt (Hrsg.). *Datenbank-Handbuch,* Springer Verlag, 1987.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Cambridge, MA: MIT-Pess, 1985.

[PG95] Projektgruppe 240. *Scotland Yard – Evaluation und Entwurf von Werkzeugen für eine Entwicklungsumgebung zum Prototyping*, Abschlußbericht der Projektgruppe 240, Software-Technologie Lehrstuhl X,, Universität Dortmund, März, 1995.

[Pla91] Dirk Platz. *Eine Recoverykomponente für das Objektmanagementsystem H-PCTE.* Diplomarbeit, Universität Dortmund, 1991.

[Reu87] Andreas Reuter. *Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen.* Kapitel 4 in [LS87], 1987.

[RSL78] D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis. *System Level Concurrency Control for Distributed Database Systems.* TODS Vol.3 No.2, 1978.

[SDDS86] J.T. Schwartz, E. Dubinsky, R. Dewar, E. Schonberg. *Programming with Sets, An Introduction to SETL.* Springer-Verlag, New-York, 1986.

[See93] Wolfgang Seelbach. *Adreßraum-Partitionierung*, Internes Memorandum 93/2, Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 1993.

[See94a] Wolfgang Seelbach. *Long Field Attributes and Discretionary Access Controls in H-PCTE,* Dokumentation zu H-PCTE Version 2.6, 1994.

[See94b] Wolfgang Seelbach. *H-PCTE Installation Environment.* Dokumentation zu H-PCTE Version 2.6, Juli 1994.

[SPG91] A. Silberschatz, J. Peterson P. Galvin. *Operating System Concepts.* Addison-Wesley, 3rd edition, 1991.

[VG93] Gottfried Vossen, Margret Groß-Hardt. *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, 1993.

[Vos94] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*, 2. Auflage, Addison-Wesley, 1994.

[Wec84] G. Weck. *Datensicherheit,* aus der Reihe: Leitfäden der angewandten Informatik, B.G. Teubner Verlag Stuttgart, 1984.

[Wei88] Gerhard Weikum. *Transaktionen in Datenbanksystemen – fehlertolerante Steuerung paralleler Abläufe.* Addison-Wesley, 1988.

[Wei89] Gerhard Weikum. *Das aktuelle Schlagwort – Geschachtelte Transaktionen.* Informatik-Spektrum, S. 102-106, Bd. 12, 1989.

[Wil91] W. Wilson Ho. *Dld – A Dynamic Link/Unlink Editor, Version 3.2.3*. Programmdokumentation, 1991.

[WJ93] Lois Wakeman, Jonathan Jowett. *PCTE - The Standard for Open Repositories.* For the PIMB Association, Prentice Hall, 1993.

[WO90] W. Wilson Ho, R.A. Olsson. *An Approach to Genuine Dynamic Linking.* Division of Computer Science, University of California, Davis, U.S.A.