TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte

Technical Reports

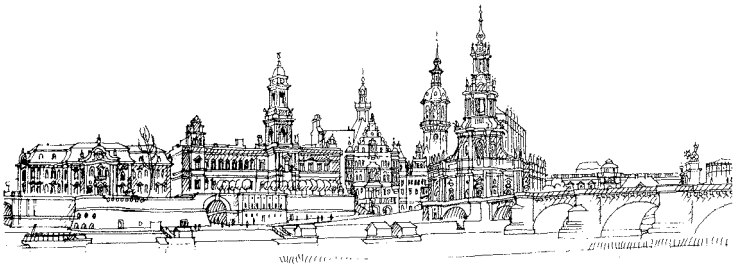
ISSN 1430-211X

TUD / FI 97 / 05 - Mai 1997

Gratz, A.; Schulz, P.; Spallek, R.G.

Institut für Technische Informatik

Entwurf und Realisierung eines Informationssystems zur Prozeß-datenverwaltung und -verarbeitung im durchgängigen Halbleitertechnologieprozeß



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

Entwurf und Realisierung eines Informationssystems zur Prozeßdatenverwaltung und -verarbeitung im durchgängigen Halbleitertechnologieprozeß

Achim Gratz* Patrick Schulz* Rainer G. Spallek*

22.April 1997

Technische Universität Dresden
Fakultät Informatik
Institut für Technische Informatik

* email: gratz@ite.inf.tu-dresden.de, schulz@freia.inf.tu-dresden.de, rgs@ite.inf.tu-dresden.de

Inhaltsverzeichnis

1 Motivation

Wohl kaum ein anderer Industriezweig hat sich in den letzten Jahren so rasant entwickelt wie die Halbleiterindustrie. Ein Ende dieser Entwicklung scheint nicht absehbar. Neue Dimensionen auf dem Weg nach immer höher integrierbaren Schaltungen, höheren Taktraten und größeren Chipflächen wurden und werden erreicht. Dies war und ist nur mit sehr hohen Investitionen in Forschung und Technologieentwicklung möglich. Zusätzlich dazu müssen auch die Produktionsanlagen, Fab genannt, regelmäßig und in vergleichsweise kurzen Zeiträumen vollständig erneuert werden. Die Kosten für diese Art Technik und ihre Entwicklung sind enorm. Trotzdem muß ein Halbleiterunternehmen permanent in Weiterentwicklung der Technologie und Reproduktion investieren, will es den Anschluß an den Weltmarkt nicht verlieren. Neben den hohen Kosten dieser Technologie sind es insbesondere die Produktpreise, die im hart umkämpften Halbleitermarkt über Profit und die langfristige Überlebensfähigkeit des Unternehmens entscheiden.
Es ist daher vordringlichstes Ziel eines jeden Halbleiterherstellers, die Produktion so effektiv wie möglich zu gestalten, um konkurrenzfähige Produkte profitabel verkaufen zu können. Dies erst sichert die Zukunft des Unternehmens auf längere Sicht hin.
Es ist daher nur selbstverständlich, daß in der Halbleiterindustrie von Beginn an in großem Stil Computer eingesetzt werden. Man kann die Chipindustrie ohne Zweifel als durchgängig computerisiert bezeichnen. Die Gründe dafür sind mehrschichtig und liegen in der Eigenart der Branche. Die Dimensionen sind so winzig, daß nur mit computergestützter Technik überhaupt produziert werden kann, der Mensch kann nur noch steuernd eingreifen. Selbst solche Eingriffe betreffen nur Ausnahmesituationen, im Normalfall ist der Mensch ein Störfaktor für den Produktionsprozeß. Daher haben Menschen in einer Fab nur kontrollierende Funktion bzw. führen wenig fehlerintensive Tätigkeiten aus. Von der Produkt- und Technologieentwicklung über die Produktion und deren Steuerung bis hin zur Qualitätskontrolle, überall kommt der Computertechnik eine entscheidende Rolle zu. Bei einer solchen Durchdringung mit Computern stellt sich die Frage nach einer effektiven Nutzung der vorhandenen Möglichkeiten.

1.1 Heterogenität der Datenverarbeitung

Ein integrierender Modellierungsansatz kommt dieser Forderung am ehesten nach, versucht er doch alle anfallenden Daten in all ihren Zusammenhängen darzustellen. Dies setzt jedoch die Integrationsfähigkeit der vorhandenen Technik voraus, eine sehr schwer zu erfüllende Aufgabe. Die Hauptursache dafür liegt in der Heterogenität der Systeme, die ganz zwangsläufig ist. Jeder Hersteller für Fab-Equipment verwendet seine eigene Steuertechnik, verschiedenartige Systeme zur Datenerfassung und -speicherung werden problembezogen beschafft und genutzt. Letztlich findet sich in einer Fab eine reichhaltige Sammlung von Hard- und Softwaresystemen aller Art, deren Zusammenwirken über die Effizienz der Produktion entscheidet. Dies stellt eine denkbar schlechte Ausgangsbasis für die Integration aller Rechentechnik in ein ganzheitliches System dar. In der Praxis findet man deshalb höchstens Ansätze in diese Richtung. In einer Fab werden eine ganze Reihe solcher Lösungen angewendet, schließlich will man den gesamten Produktionsprozeß nicht nur steuern, sondern auch beobachten und regeln, im Idealfall automatisch. Eine frühzeitige Erkennung von Fehlern kann bedingt durch die relativ lange Prozeßdauer ausgesprochen viel Geld sparen helfen.

Die Halbleiterhersteller setzen deshalb eine Vielzahl verschiedener Monitor- und Meßsysteme wie zum Beispiel KLA, INSPEX, WaferSleuth ein, die im Regelfall eigene Datenhaltungs- und Datenverarbeitungstechnik beinhalten.
Daraus resultiert ein Grundproblem, das viele Halbleiterhersteller teilen. Es sind zu jedem Prozeßschritt eine derartige Menge an Daten vorhanden, daß es nur schwer möglich ist, aus der Flut von Informationen die wirklich aussagekräftigen Daten herauszufiltern und aufzubereiten. Die verteilte Datenhaltung verschlimmert dieses Problem weiter. Außerdem kommt so ein weiteres schwerwiegendes Konsistenzproblem hinzu. Daten müssen, obwohl sie semantisch zusammenhängen, unabhängig von einander gehalten werden. Dabei eventuell auftretende Datendopplung erfordert Protokolle zur Konsistenzsicherung der verschiedenen Datenquellen.
Deshalb gibt es Ansätze, diese verteilten Daten zentral zu speichern. Die ungeheure Menge an Daten mit sehr unterschiedlichem Informationsgehalt sowie bestehende Interface- und Lastprobleme verhindern jedoch eine erfolgreiche Realisierung im täglichen Einsatz.

1.2 Die Rolle des Fab-Ingenieurs

Die Herstellung von Halbleiterschaltkreisen ist ein komplexer Prozeß, in dem sowohl physikalische als auch chemische Verfahren Anwendung finden. Es ist daher nicht verwunderlich, wenn es neben Elektrotechnikern auch Physiker und Chemiker sind, die als Fab-Ingenieure den Prozeß überwachen und regeln. Diese Aufgaben können sie nur erfolgreich erfüllen, wenn es gelingt, die aus der Fab kommenden Daten aufzufinden, auf das Wesentliche zu reduzieren und richtig zu interpretieren. Somit ist die Datenverwaltung einer der Hauptbeschäftigungen eines Fab-Ingenieurs. Dies entspricht jedoch nur bedingt seiner Qualifikation. Sie ist ein unproduktiver Teil der Tätigkeit eines Fab-Ingenieurs, eine Voraussetzung für die eigentliche Arbeit. Eine komplexe Datenverwaltung, wie sie durch die Heterogenität der Computertechnik fast zwangsläufig ist, erfordert hohe Kosten nicht nur für ihre Unterhaltung, sondern auch für die Schulung der Benutzer. Letztlich gilt es den Anteil der unproduktiven Suche und Verarbeitung von Prozeßdaten möglichst klein zu halten. Nicht selten passiert es daher, daß vorhandene Technik nicht oder nur unzureichend genutzt wird. So hat die Anschaffung teuren Equipments zwar einerseits die Regelbarkeit des Prozesses erhöht, auf der anderen Seite hält sie den Fab-Ingenieur noch länger von seiner eigentlichen Tätigkeit ab, so daß sich der Nutzen letztlich relativiert.
Diesem Problem kann nur entgegengewirkt werden, wenn der Fab-Ingenieur Softwarelösungen erhält, die speziell auf seine Bedürfnisse zugeschnitten sind und die Problematik der Heterogenität und Komplexität verbergen. Solche Lösungen existieren bereits in verschiedenen Entwicklungsstadien oder sind in Entwicklung. Sie sind oftmals Eigenentwicklungen oder entstehen in Zusammenarbeit mit ConsultingFirmen.
Doch auch dieser Ansatz verfolgt das Prinzip der Teillösungen, lediglich die Komplexität sinkt, die Anzahl der für die Arbeit notwendigen Programme nimmt ab. Die einfachsten Lösungen stellen Abfragemasken für Datenbanken dar, welche die Auswahl der benötigten Daten erleichtert. Andere Programme fragen automatisch verschieden Datenquellen ab und präsentieren dem Ingenieur bereits verarbeitete Daten. Solchen Lösungen ist gemeinsam, daß es nicht möglich ist, Zusammenhänge darzustellen, die über das modellierte Teilproblem hinausgehen. Normalerweise ist eine Konfigurierbarkeit durch den Benutzer nur innerhalb der vom Programm erwarteten Parameter möglich.

1.3 Dokumentation als Wissensbasis

Während anfallende Prozeßdaten gespeichert und verarbeitet werden, ist eigentlich die Interpretation und Bewertung der Daten von Interesse. Um dieses Wissen zu bewahren und nachnutzen zu können, muß jede Tätigkeit dokumentiert werden. Da es in der Praxis relativ schwer ist, während der eigentlichen Tätigkeit diese zu dokumentieren, haben fast alle Firmen ein internes Reportwesen aufgebaut. So wird sichergestellt, daß alle relevanten Arbeiten beschrieben werden. Leider haben die Reports genaugenommen einen anderen Zweck. Der Inhalt eines Reports ist in der Regel eine Zusammenfassung der geleisteten Arbeit, gedacht zur Leistungsbewertung durch den Vorgesetzten. Die geleistete Arbeit wird dabei selten ausführlich genug dargestellt und nicht nach den verschiedenen Problemen unterteilt. Es ist im Nachhinein schwer, die Lösung eines Problems nur mittels des angefertigten Reports zu rekonstruieren. Dabei tritt neben der Ausführlichkeit das Dilemma der Archivierung hinzu. Das in Form von Reports dokumentierte Wissen ist zwar zur Leistungsbewertung geeignet, aber zur problemorientierten Suche nicht ausreichend strukturiert. Darüber hinaus ist die Dokumentation meist auch physisch zu weit vom Problem entfernt. Es entsteht ein ähnliches Problem wie bei den Prozeßdaten: Wahrscheinlich gibt es die gewünschte Information in der Fab, sie ist nur sehr ineffizient zu finden und auszuwerten.
Zusammenfassend kann festgestellt werden, daß eine effiziente Gestaltung der Produktion von folgenden Faktoren abhängt:

- Abbildung der Zusammenhänge in der Fab in einem einheitlichen System,
- einfache, schnelle semantische Aufbereitung der problembezogenen Daten,
- strukturierte Dokumentation aller relevanten Tätigkeiten.

Die in der Praxis verwendeten Systeme sind aufgrund ihres Konzepts nicht in der Lage, diese Voraussetzungen zu erfüllen. Aus diesem Grund wird am Institut für Technische Informatik der TU-Dresden seit einiger Zeit über ein System nachgedacht, das einen neuen Ansatz verwirklichen soll. Diese Arbeit beschreibt das ProSpecT-System, sein Konzept, die Spezifikation und eine prototypische Implementierung.

2 Anforderungen

Der Halbleiterherstellungsprozeß ist grundsätzlich sehr dynamisch. Er wird für die Herstellung eines einzigen Produkts entwickelt und an die Verhältnisse in der Fab angepaßt. Hierbei treten bereits erste Modifikationen auf. Während der Prozeß läuft, wird er immer wieder verändert, um eine höhere Ausbeute zu erzielen. Oft dient ein erfolgreich eingesetzter Prozeß als Ausgangsbasis für den Herstellungsprozeß neuer Produkte. Eine Prozeßfamilie entsteht. In der Regel werden mehrere Prozesse gleichzeitig in der Fab genutzt. Es ist für den Ingenieur demzufolge wichtig, seine Sicht auf die Produktion stets an die aktuelle Situation anpassen zu können. Ein dynamisches Prozeßmodell ist notwendig.
Dieses Modell muß alle relevanten Daten in ihren semantischen Zusammenhängen darstellen und auswerten können. Dabei ist insbesondere die heterogene Datenverwaltung zu berücksichtigen. Es müssen verschiedenste Datenquellen auswertbar sein. Diese Funktionalität sollte für den Benutzer transparent sein, schließlich konzentriert sich seine Tätigkeit auf den Herstellungsprozeß. Aus

dieser Tatsache kann ebenfalls die Forderung nach Benutzerfreundlichkeit, Konfigurierbarkeit sowie Erweiterbarkeit des Modells abgeleitet werden. Außerdem wäre es wünschenswert, wenn bereits während der Arbeit diese mitdokumentiert wird. Eine problemorientierte Dokumentation sollte deshalb Teil des Prozeßmodells sein.
Dazu gehört auch, daß dokumentiertes Wissen so lange im System verbleibt, bis es explizit vernichtet wird. So kann auch Jahre später auf bereits gemachte Erfahrung zurückgegriffen werden.
Neben den Herstellungsprozessen sind auch Geschäftsprozesse eng mit dem Funktionieren einer Fab verbunden. So gibt es unter anderem festgelegte Abläufe, nach denen Änderungen am laufenden Prozeß gemacht werden. Dabei werden Vorschläge in verschiedenen Gremien beraten, beschlossen und eventuell deren Umsetzung ausgearbeitet.
Diese Geschäftsprozesse sind eng mit dem Produktionsprozeß verzahnt und sollten auch im Zusammenhang dargestellt werden. An ein neuartiges Informationssystem zur Modellierung einer Fab werden daher zusammenfassend folgende grundlegende Anforderungen gestellt:

- Nachbildung dynamischer Prozesse,
- Zugriff auf verschiedenste Datenquellen,
- Dokumentation als Bestandteil der Tätigkeit,
- Ansammlung und Archivierung von Daten und Dokumentation,
- Transparenz von nicht problembezogener Funktionalität für den Nutzer,
- Darstellung von Herstellungs- und Geschäftsprozessen,
- Benutzerfreundlichkeit,
- Erweiterbarkeit und Konfigurierbarkeit durch den Nutzer.

3 Konzept

Wie in Abschnitt 1 (siehe Seite 1ff) bereits erwähnt, sind fast alle Voraussetzungen für ein umfassendes Modellierungskonzept bereits in der Fab vorhanden. Ein Bottom-up Ansatz könnte zum Entwurf des Systems genutzt werden. Diesen wenden bereits existierende Lösungen an, jedoch modellieren sie nie das Problem im ganzen, sondern beschränken sich auf die Lösung von Teilproblemen.

Der Ansatz, der im folgenden beschrieben und benutzt wird, hat integrativen Charakter. Seit einiger Zeit verfolgt Dipl. Ing. Achim Gratz die Idee, eine Fab als Ganzes zu modellieren, da nur so alle Zusammenhänge erkannt und nachgebildet werden können [GS-96].

3.1 Der integrative Modellierungsansatz

Die Modellierung der Arbeit einer gesamten Fab kann relativ leicht begründet werden. Einem Außenstehenden stellt sich eine Fab zunächst ziemlich einfach dar, sieht er doch schließlich nur ihre Bestandteile. Equipment, Personal, Infrastruktur, Produkte (Wafer) sowie CIM- Technik sind alle notwendigen Voraussetzungen, um Halbleiterschaltkreise zu fertigen. Die komplexen Zusammenhänge zwischen diesen „Grundbausteinen“ machen letztlich erst eine Fab aus. Jeder Mitarbeiter hat eine spezielle Sicht auf das Funktionieren der Fab, je nachdem, welche Position er ausfüllt. Diese Folge der Arbeitsteilung mag zu dem falschen Schluß führen, es gebe abgeschlossene, unabhängige Aufgabengebiete. Letzten Endes stellen sie aber nur Teile des großen Netzwerks aus Zusammenhängen dar, dem Modell der Fab. Bildet man die gesamte Fab nach, lassen sich auch alle Zusam-

menhänge modellieren. Die Benutzer des Systems bekommen ihre Sicht auf das Modell, aber tatsächlich arbeiten alle mit denselben Daten und Zusammenhängen.
Der integrative Modellierungsansatz versucht, sowohl die Grundbestandteile als auch ihre Zusammenhänge zu abstrahieren. Die modellierten Grundbausteine einer Fab werden als Basiskomponenten bezeichnet, deren Zusamenhänge als Modellkomponenten. Das Gesamtmodell besteht aus einer Hierarchie von Teilmodellen, wobei jede Hierarchieebene ein höheres Abstraktionsniveau darstellt. So wird die Komplexität der Zusammenhänge überschaubar.
Im Gegensatz dazu steht der distributive Modellierungsansatz. Er verringert nach dem Prinzip „Teile und Herrsche“ die Problemkomplexität, indem das Problem in Teilprobleme zerlegt wird, die besser überschaubar sind. Dabei geht sowohl der Gesamtüberblick verloren, als auch die Möglichkeit, das Abstraktionsniveau zu verändern. Der integrative Ansatz hingegen bietet die Möglichkeit, jederzeit das gewählte Abstraktionsniveau zu verlassen, um die Komplexität des Problems weniger oder mehr überschaubar zu machen.
Den Vorteilen des integrativen Ansatzes stehen nicht zu vernachlässigende Nachteile gegenüber. Das dabei wohl schwierigste Problem stellt die Komplexität des Modells dar. Obwohl es ein möglichst genaues Abbild des Originals sein soll, werden doch bewußt Einschränkungen gemacht, die das Modell auf das Wesentliche reduzieren sollen. Dabei kann es vorkommen, daß Zusammenhänge nicht modelliert werden, die sich gegebenenfalls als unverzichtbar erweisen. Andererseits birgt eine zu genaue Nachbildung große Probleme hinsichtlich der Realisierung und Fehlerhaftigkeit. Auch der Zeitfaktor ist hierbei nicht zu vernachlässigen, gerade da es sich um ein dynamisches Modell handelt. Trotzdem bildet der integrative Ansatz als einziger die Möglichkeit, das Problem umfassend nachzubilden und dessen Komplexität begreifbar zu machen.

3.2 Modellierung

Wie bereits erläutert, geht der integrative Ansatz davon aus, die Fab als Ganzes nachzubilden. Dies ist ein komplexer Vorgang, der zusätzlich durch den dynamischen Charakter des Modells erschwert wird. Andererseits stellt dies kein Problem dar, das nur in der Halbleiterindustrie existiert. In vielen Bereichen der Wirtschaft sind komplexe Abläufe zu beherrschen, es müssen Modelle dafür geschaffen werden. Betrachtet man diese Tatsache genauer, stellt man fest, daß nicht die Modellbildung das eigentliche Problem ist, sondern das System, das dazu verwendet wird.
Man möchte ein Computersystem entwerfen, das alle gewünschten Vorgänge in der Fab nachbilden kann. Das Modell einer Fab ist also letztlich ein laufendes Computerprogramm. Dieses Computerprogramm muß selbst entworfen, modelliert werden. Am Ende entsteht meist ein System, das eine Fab zum Zeitpunkt der Problemanalyse repräsentiert. Um eine gewisse Flexibilität zu gewährleisten, hat man unter Umständen Erweiterungsschnittstellen vorgesehen, die später genutzt werden sollen, um eine Anpassung des Modells an das Original zu erreichen. Dies ist ein zwar erweiterbarer, aber nicht flexibler Ansatz. Bei größeren, nicht vorhersehbaren Änderungen des Herstellungsprozesses kommt man um eine Modifikation des Modells, also des Computerprogramms, nicht umhin. Damit ist eine zeit- und personalintensive Betreuung des Programms durch den Hersteller notwendig.
Betrachtet man das zu modellierende Problem, die Fab, genauer, so stellt man fest, daß alle Zusammenhänge zwischen den Grundbestandteilen selbst wieder Modelle sind. Jede Sicht, die Mitarbeiter auf ihre Arbeit haben, ist wieder eine Abstraktion der Realität. So stellt die Personal-

struktur der Mitarbeiter letztlich ein Ordnungsmodell[1] dar, den Produktionsprozeß kann man als ein Flußgraphenmodell[2] begreifen. Für fast alles schafft sich der Mensch Modelle, mit denen er die Realität begreift. Was liegt also näher, als ein System zur Modellierung der Realität zu entwerfen, mit dessen Hilfe man alle Teilmodelle einer Fab nachbildet?
Genau diesen Gedanken verfolgt das hier vorgestellte Projekt. Es ist ein Programmsystem, das alle Basisfunktionalität bereitstellt, die für die Modellierung eines Problems benötigt wird. Es ist im Ausgangszustand folglich nicht geeignet, ein komplexes Problem nachzubilden. Wohl aber kann es um Modelle erweitert werden, um letztlich alle geforderte Funktionalität bereitzustellen.

3.3 Basiskomponenten

Ein Gesamtmodell zur Nachbildung eines komplexen Problems besteht oft aus zwei Komponenten, wie in Abbildung 3-1 gezeigt:

- die Nachbildungen der Grundbestandteile des Problems (Basiskomponenten),
- Zusammenhänge zwischen diesen Atomen (Modellkomponenten).

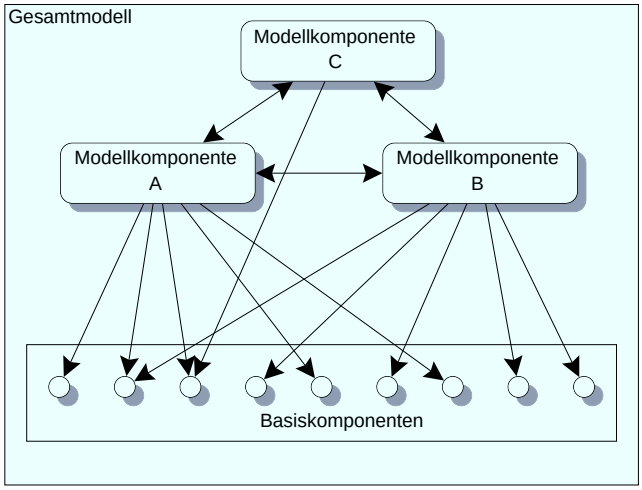


Abbildung 3-1: Darstellung eines Gesamtmodells als Gesamtheit von Teilmodellen

Basiskomponenten abstrahieren Bestandteile der Realität und korrespondieren mit ihnen. Darüber hinaus lösen sie das Problem der Datendopplung und Konsistenz, da eine Basiskomponente nicht alle zugehörigen Daten selbst enthalten muß, sondern stattdessen Verweise auf andere Datenquellen, die diese enthalten. Die Modellierung der Basiskomponenten macht damit eine transparente Datenverwaltung möglich.
Bei genauer Betrachtung fällt auf, daß diese beiden Komponenten einander ähnlich sind. Sie stellen beide Abstraktionen der Realität dar, strukturell sind sie gleichartig. Basiskomponenten können von Modellkomponenten referenziert werden, aber nicht selbst auf andere Komponenten verweisen. Man kann das Modellsystem auch als Graph auffassen, bei dem Knoten die Komponenten und die Kanten deren Zusammenhänge darstellen (Abbildung 3-2).

[1] analytisches Modell zur Strukturierung der Realität

[2] algorithmisches Modell zur Ablaufbeschreibung

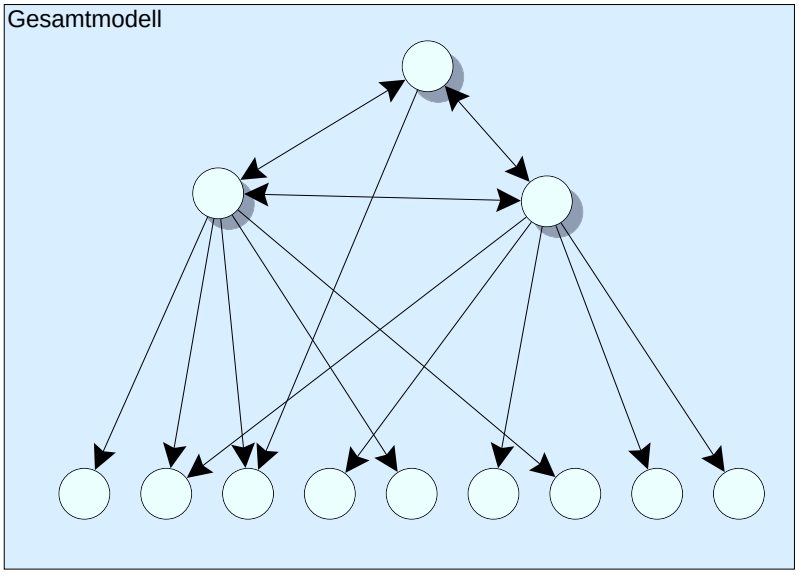


Abbildung 3-2: Darstellung eines Gesamtmodells als Graph

So gesehen, gibt es für das Modellsystem nur gleichartige Objekte, die Teilmodelle darstellen. Beide können prinzipiell in gleicher Weise verwaltet werden. ProSpecT nutzt dazu eine Repräsentation, das abstrakte ProSpecT-Objekt. Es enthält alle Funktionalität, die zu seiner Verwaltung benötigt wird, und muß zur Repräsentation eines Modells funktionell erweitert werden. Das bedeutet, daß ProSpecT sowohl hinsichtlich der Nachbildung der Basiskomponenten als auch der gewünschten Modellkomponenten erweitert werden muß. So wird eine weitreichende Flexibilität erreicht, die den Einsatz in nahezu allen Bereichen des Lebens möglich macht. Seine eigentliche Funktionalität erzielt ProSpecT letztlich nur durch seine Erweiterungen. Darauf soll später noch ausführlicher eingegangen werden.Damit beschränkt sich die Funktionalität des Basissystems auf:

- Grundfunktionalität des ProSpecT-Objektes,
- Verwaltung der ProSpecT-Objekte,
- Erweiterbarkeit um Modellkomponenten,
- Erweiterbarkeit um Basiskomponenten,
- Benutzerverwaltung.

3.4 Leistungsumfang

3.4.1 Das abstrakte ProSpecT-Objekt

Als abstrakte Repräsentation von Basis- und Modellkomponente stellt das ProSpecT-Objekt nur elementarste Funktionalität bereit. Diese wird zum einen zur Verwaltung innerhalb des Objektverwaltungssystems benötigt; Hauptaufgabe des ProSpecT-Objektes ist, als Basis für Spezialisierungen zu dienen. In der Sprache der Softwaretechnologie stellt es ein abstraktes Klassenobjekt dar, das durch Vererbung spezialisiert wird und Instanzen produzieren kann. Es besitzt Raum für Attribute und Methoden, die später eingefügt werden können. Dazu stehen Schnittstellen zur Verfügung (Abbildung 3-3). Obwohl das abstrakte ProSpecT-Objekt alle Eigenschaften einer

Klasse besitzt, handelt es sich für ProSpecT um ein Objekt, das von anderen ProSpecT-Objekten nicht unterscheidbar ist.
Neben Attributen und Methoden nimmt jedes ProSpecT-Klassenobjekt auch Verweise auf andere ProSpecT-Klassenobjekte auf, von denen es referenziert wird. Damit ist bei Änderungen bzw. beim Erzeugen eines neuen Instanzobjekts eine automatische Einordnung in bestehende Zusammenhänge möglich. Außerdem unterstützt das ProSpecT-Objekt das Konzept des Dokumentierens während des Arbeitens. So enthält es neben einem Namensfeld ebenfalls Raum für seine verbale Beschreibung. So kann bei jeder Modifikation bzw. Neuerzeugung eines Objekts eine Dokumentation gefordert werden.
Jedes ProSpecT-Objekt besitzt außerdem einen Klassenbezeichner, der die Zugehörigkeit zu seiner Klasse herstellt. Bei der Spezialisierung wird eine neue Klasse kreiert; innerhalb des ProSpecT Systems existiert somit eine hierarchische Klassenanordnung, ähnlich dem objektorientierter Sprachen.

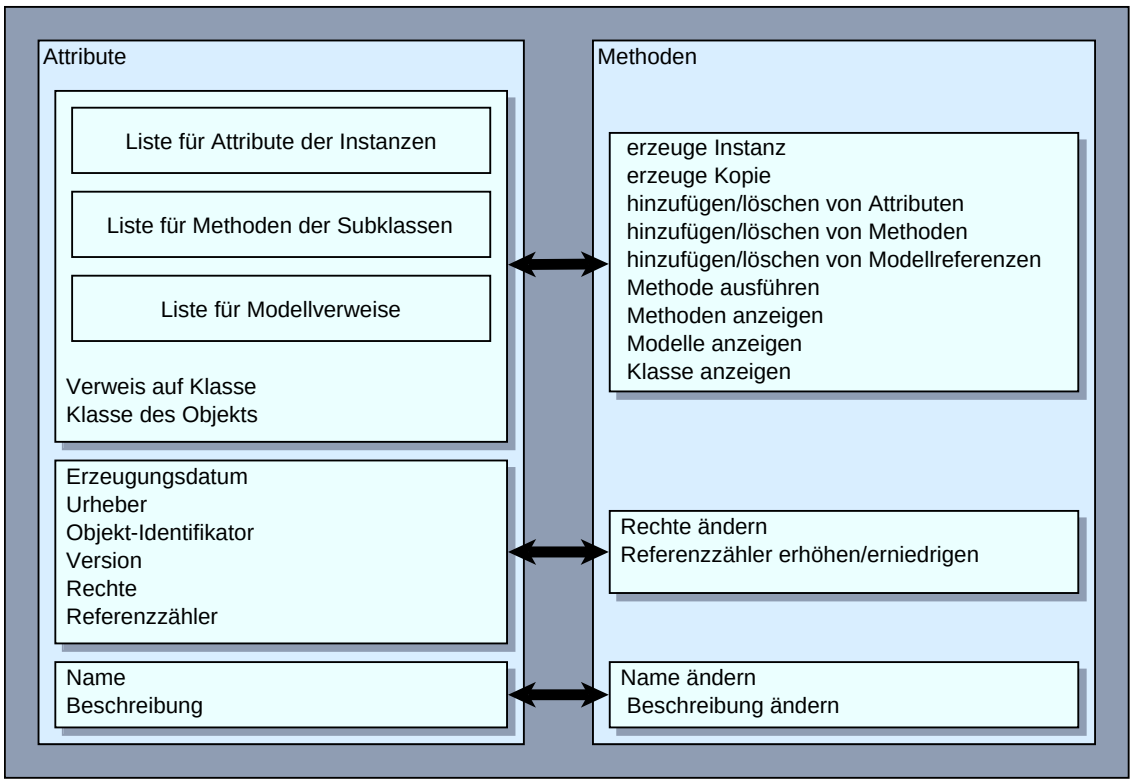


Abbildung 3-3: Eigenschaften und Struktur des abstrakten ProSpecT-Objekts

3.4.2 Verwaltung der ProSpecT-Objekte (Objektverwaltung)

Da im System ausschließlich Datenobjekte vorkommen, kann durchaus von einer Objektverwaltung gesprochen werden. Hierbei treten viele Anforderungen auf, wie sie auch an Datenbanksysteme gestellt werden [LKK-93]:

- Konsistenz,
- Persistenz,
- Lese- und Schreiboperationen / Transaktionen,

- Datenunabhängigkeit und Offenheit der Daten,
- Ausfallsicherheit,
- Mehrnutzerbetrieb / Konkurrenz

Da die ProSpecT-Objektverwaltung konzeptuell nur auf Verwaltung einzelner, isolierter Datenobjekte ausgelegt ist, können viele der oben genannten Anforderungen relativ leicht erfüllt werden. Sie könnte auch als vereinfachtes Datenbank-Modell ansehen werden. Damit wäre es durchaus denkbar, bestehende Datenbanken als Objektverwaltung zu nutzen, was aber nicht vorrangiges Ziel des Konzepts ist. Vielmehr verfolgt die ProSpecT-Objektverwaltung Ansätze, die von existierenden DBMS nicht unterstützt werden. So ist eine automatische Versionsverwaltung aller Datenobjekte ebenso vorgesehen wie eine halbautomatische Freispeicherverwaltung.

3.4.3 Versionsverwaltung

Dynamische Modelle unterliegen ständigen Veränderungen. Diese können zum einen in einer Erweiterung oder Verkleinerung der Abbildung der Realität bestehen, andererseits sind es sehr oft nur Modifikationen bestehender Modellkomponenten. Dadurch entsteht semantisch betrachtet, eine neue Version dieser Komponente. Dieser Fakt ist für den Benutzer naheliegend, schließlich hat er das Modell modifiziert. Es ist also durchaus folgerichtig, auch eine Beziehung zwischen dem bisherigen Objekt und dessen neuerem Nachfolger innerhalb der Objektverwaltung herzustellen. Dies ist selbstverständlich nur sinnvoll, wenn das veraltete Objekt in der Objektverwaltung verbleibt. Eine der Anforderungen an ProSpecT ist, die Ansammlung von Daten über einen längeren Zeitraum zu ermöglichen. Diese Daten dienen als Wissensbasis und müssen erst gelöscht werden, wenn sie veraltet sind.
So entstehen zwangsläufig verschiedene Versionen eines Objekts - jede Modifikation eines Objekts erzeugt automatisch eine neue Version. Die Verwaltung dieser Versionen wird durch die ProSpecT-Objektverwaltung konzeptuell unterstützt. Dabei ist sowohl ein Zugriff auf das individuelle Objekt, als auch die Gruppe seiner Versionen realisiert. Man kann sich hierbei die verschiedenen Versionen des gleichen Objekts als Vektor vorstellen, bei dem die neuere Version jeweils einen Verweis auf die ältere Version besitzt. Damit ist sichergestellt, daß alle Versionen jederzeit erreichbar sind. Somit existiert eine netzwerkweite Konsistenz der Daten.

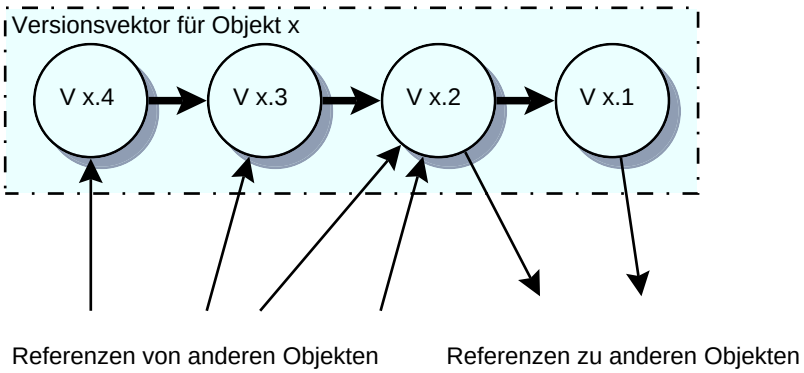


Abbildung 3-4: Objektvektor - logischer Zusammenhang verschiedener Versionen

Ansätze dieser Art sind nicht neu, so besaß schon das Dateisystem von VMS eine Versionsverwaltung für Dateien. Auch bei Datenbanken versucht man, die Geschichte von Informationen zu bewahren. Objektorientierte Datenbanken wie zum Beispiel das GOOD-Modell [GPV+-90] nutzen ähnliche Referenzierungstechniken zur Versionsverwaltung der Datenobjekte .

3.4.4 Freispeicherverwaltung

ProSpecT besitzt neben seinem Charakter als Modellierungssystem auch Ähnlichkeiten mit einer Datenbank. Erklärtes Ziel des Konzepts ist, durch Kombination dieser Möglichkeiten, eine strukturierte Wissensbasis aufzubauen. Modellierte Zusammenhänge bleiben gespeichert, bis sie nicht mehr benötigt werden. Da jede Änderung eines Objektes zu einer neuen Version führt, ist in kurzer Zeit mit einer großen Datenmenge zu rechnen. Dabei handelt es sich um verschiedene Versionen von Basiskomponenten und Modellkomponenten. Da jedes Objekt und seine Versionen von mehreren anderen Objekten referenziert werden können, ist schwer entscheidbar, wann ein Objekt entbehrlich, also nicht mehr referenziert ist. Dies ist nur feststellbar, wenn alle Modelle hinsichtlich ihrer Referenzen betrachtet werden, ein sehr zeitintensives und schwer zu realisierendes Verfahren.
ProSpecT nutzt deshalb eine automatisch funktionierende Freispeicherverwaltung, um diese Probleme zu lösen. Die Funktionsweise dieser Technik läßt sich durchaus mit der des menschlichen Gehirns vergleichen. Dort wird eine Information im Zusammenhang mit anderen Informationen gespeichert und ist nur über diese Zusammenhänge zu erreichen. Werden diese Zusammenhänge aufgehoben, also lange nicht genutzt, ist die Information nicht mehr zugänglich und wird gelöscht, bzw. „vergessen“.
Um diese Funktionalität zu erreichen, nutzt ProSpecT eine Referenzzählungstechnik mit automatischer Freigabe. Ähnliche Techniken existieren als „reference counting“ und „autoreleasing“ in Klassenbibliotheken [NEX-94] bzw. Freispeicherverwaltungen objektorientierter Programmiersprachen [JON-96]. Die „hard links“ des 4.3BSD UNIX Dateisystems nutzen ebenfalls dieses Verfahren und machen seine Funktionsweise recht verständlich [LMK-88].
Dabei besitzt jedes Objekt einen Referenzzähler, der die Anzahl der auf es verweisenden Objekte angibt. Beim ersten Einspeichern wird dieser Zähler initialisiert. Wird ein neuer Verweis auf dieses Objekt erzeugt, bekommt es die Aufforderung, seinen Referenzzähler zu inkrementieren. Umgekehrt wird bei jeder gelöschten Referenz der Zähler dekrementiert. Erreicht der Referenzzähler eines Objekts den Wert Null, ist es nicht mehr referenziert. Es meldet dem ProSpecT-Objektverwaltungssystem, daß es gelöscht werden kann, und wird daraufhin aus der Objektverwaltung entfernt.
Besondere Bedeutung kommt dem Initialisierungswert des Referenzzählers zu. Wird ein Objekt bereits mit einem Zählerstand größer Null erzeugt, ist es praktisch nicht mehr löschbar, da mehr Verweise gelöscht werden müßten, als tatsächlich existieren. ProSpecT nutzt diesen Umstand, um möglichst viele Zusammenhänge zu speichern und trotzdem nicht benötigte Objekte zu löschen. Alle Modellkomponenten, die ihrerseits Basiskomponenten referenzieren, werden mit einem Zählerstand von Eins initialisiert. Sie sind daher nicht durch Entfernen anderer Verweise löschbar. Das wirft die Frage auf, wie Modellkomponenten zugreifbar sind, wenn keine Verweise zu ihnen existieren. Da jedes Objekt stets Teil seines Versionsvektors ist, kann auch ein nicht referenziertes Objekt erreicht werden. Erhält es dann eine Aufforderung, seinen Referenzzähler zu dekrementieren, wird es gelöscht. Dabei werden die Zähler aller von ihm referenzierten Objekte ebenfalls dekrementiert und diese dadurch eventuell gelöscht.
Basiskomponenten werden grundsätzlich mit Referenzzählerstand Null erzeugt. Wird auf sie nicht mehr verwiesen, werden sie gelöscht und sind auch im Versionsvektor nicht mehr enthalten. Diese

Technik kann allerdings nur funktionieren, wenn bei jedem Auf- und Abbau eines Verweises das Zielobjekt davon informiert wird, um seinen Referenzzähler entsprechend zu ändern. Dies ist durch die Implementation des ProSpecT-Kerns zu realisieren, damit keine externe Fehlerquelle das Objektverwaltungssystem in einen inkonsistenten Zustand bringen kann.
Das ProSpecT-Konzept legt nicht fest, wie die nachfolgenden Anforderungen erfüllt werden, sondern überläßt dieses der Implementierung:

- Persistenz der Objekte,
- Konkurrenz und gegenseitiger Ausschluß im Mehrbenutzerbetrieb,
- Migration der Objekte bei verteilter Objektverwaltung,
- Ausfallsicherheit.

Damit kann die Implementation sich auf die Realisierung der Versionsverwaltung und Freispeicherverwaltung konzentrieren und ansonsten auf die Funktionalität existierender Objektverwaltungssysteme[1] oder Datenbanken zurückgreifen.

3.4.5 Modellierung - Erweiterbarkeit

Wie bereits ausgeführt, ist das Kernsystem von ProSpecT ohne Funktionalität bezüglich der Modellierung von Problemen. Erst durch Erweiterung kann es der gedachten Aufgabe gerecht werden. Wie in Abschnitt 3.2 beschrieben, stellt ein Objekt entweder grundlegende Problembestandteile oder deren Zusammenhänge dar. Daher muß das allgemeine ProSpecT-Objekt in beiden Bereichen erweitert werden. So entstehen prinzipiell zwei Spezialisierungsrichtungen:

1) Nachbildung der verschiedenen grundlegenden Problembestandteile,
2) Modellierungsarten.

zu 1
Bezogen auf eine Fab müssen zunächst Abstraktionen für Dinge wie Equipment (Stepper, Implanter, Lithografie, Waferhandling,...), Mitarbeiter, Infrastruktur (Chemikalien, Energie,...), Produkte (Wafer) sowie CIM-Technik (compute server, services,...) gefunden werden. Es werden also alle relevanten Eigenschaften jedes Bestandteils zu Attributen seines Modells. So ist zum Beispiel ein Ofen durch Parameter wie Ofengruppe, Typ, Hersteller, usw. gekennzeichnet. Die so erhaltene Basiskomponente macht diese Attribute nur durch Zugriffsmethoden zugänglich.
Hierbei kommt das offene Konzept von ProSpecT voll zum Tragen. In aller Regel sind diese Attribute bereits als Daten in einer der zahlreichen Datenquellen in der Fab gespeichert. Beim Aufruf einer Zugriffsmethode kann diese die Datenquelle abfragen, welche die entsprechende Information enthält. Die Basiskomponente enthält letztlich gar kein Attribut, sondern nur einen Verweis auf die eigentlichen Daten. Die Realisierung der Zugriffsmethode stellt dabei die Schnittstelle zwischen ProSpecT und bereits existierenden Datenquellen dar. So ist für den Benutzer nicht erkennbar, woher die verschiedenen Daten stammen, die ihm das System bereitstellt. Diese Transparenz ist erklärtes Ziel des Konzepts. Informationen können so konzentriert werden, obwohl sie tatsächlich verteilt gespeichert sind. Als Folge dessen speichert ProSpecT nur Daten, die noch nirgendwo sonst gespeichert sind.

[1] englisch: persistent object store

zu 2
Die Spezialisierung des abstrakten ProSpecT-Objekts hinsichtlich Modellierungseigenschaften schafft nun die Möglichkeit, die Basiskomponenten in ihrem Zusammenhang darzustellen. Es werden primär Verweise auf diese geschaffen, die unterschiedlichen Charakter haben können. So lassen sich Komponenten strukturieren, was ein Ordnungsmodell darstellt. Andererseits sind Abläufe, Prozesse zwischen Objekten modellierbar.

3.5 Schnittstellen

Das bisher geschilderte Konzept versucht durch Integration aller möglichen Informationsquellen ein Gesamtmodell des Problems zu realisieren. Dabei erscheint der ProSpecT-Kern als eine zentrale Datenquelle. Hierbei wird das Client-Server Prinzip ausgenutzt [TAN-92]. Der Benutzer kommuniziert mit dem System von seinem Arbeitsplatzrechner aus, auf dem ein ProSpecT-Client läuft, während alle Daten bzw. Verweise auf Daten zentral auf einem Server vorgehalten werden. Abbildung 3-5 stellt die Basisarchitektur dar.

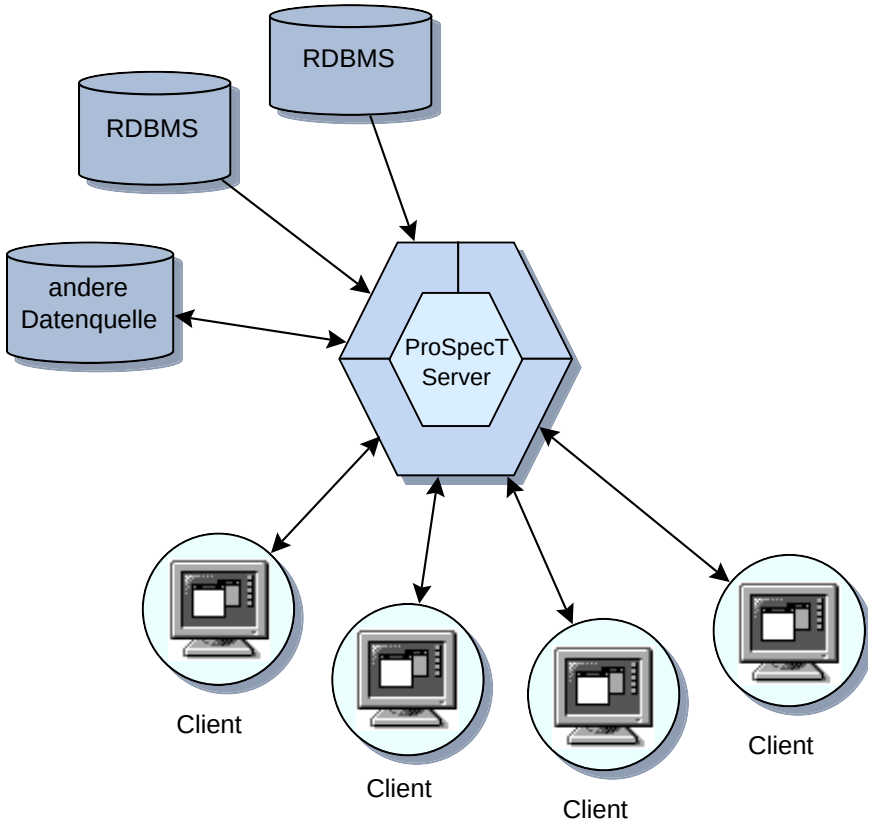


Abbildung 3-5 : ProSpecT Basisarchitektur

3.5.1 Server

Der ProSpecT-Server besteht neben Objektverwaltung, ProSpecT-Objekten bzw. deren Spezialisierungen auch aus einer Clientschnittstelle. Diese behandelt alle eintreffenden Anfragen von verschiedenen Clients und leitet sie weiter an die entsprechenden Objekte bzw. die Objektverwaltung. Darüber hinaus kann der Server eine Schnittstelle zu anderen ProSpecT-Servern haben. Eine einziger Server als zentrale Datenquelle stellt schnell sowohl hinsichtlich der Geschwindigkeit als auch bezüglich der Ausfallsicherheit ein Problem dar. Die Erfahrungen aus der Mainframe-Ära haben die Defizite dieses Konzepts offensichtlich gemacht. Außerdem ist es nur logisch, die Daten möglichst nahe dem Ort zu speichern, an dem sie benötigt werden. All dies macht eine verteilte Architektur nötig, in der mehrere ProSpecT-Server zusammenarbeiten. Damit entsteht eine erweiterte Architektur. Die Vorzüge eines solchen Designs sind:

- Lastverteilung auf mehrere Rechner bzw. Netzwerke,
- höhere Ausfallsicherheit und Möglichkeit der Redundanz der Datenbasis,
- Skalierbarkeit/Anpaßbarkeit hinsichtlich der Problemgröße,
- Nutzbarkeit vorhandener Rechentechnik.

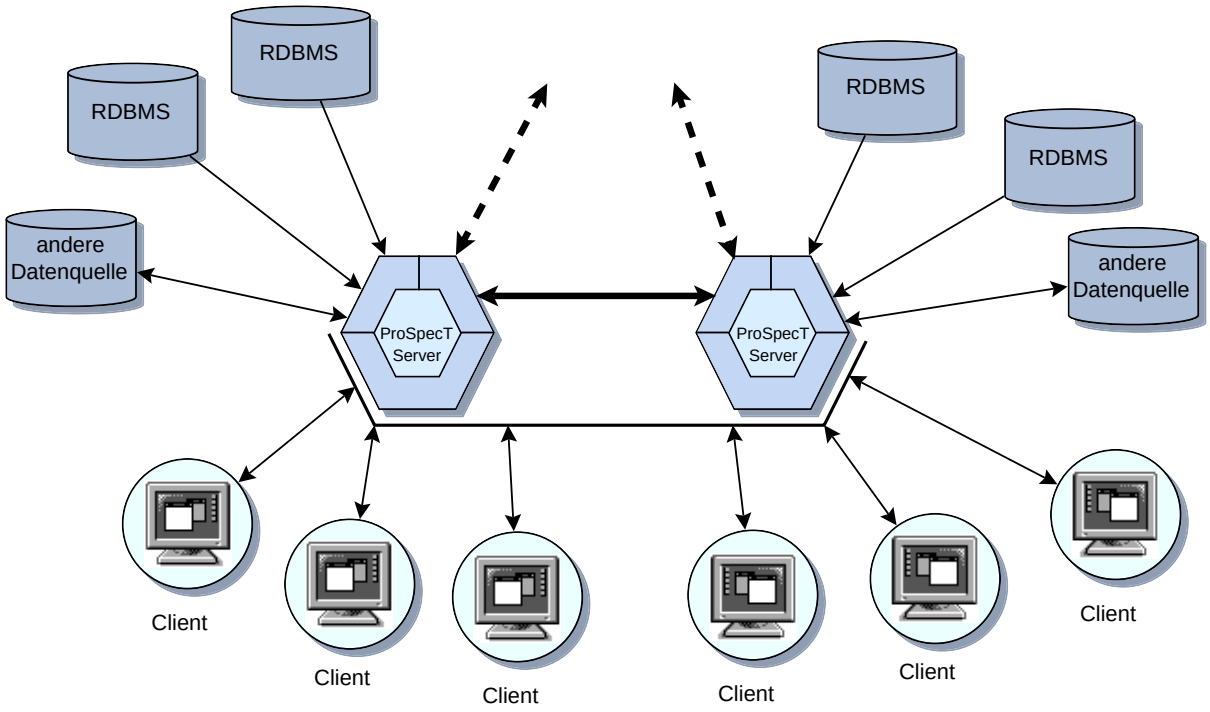


Abbildung 3-6: verteilte ProSpecT-Architektur

Wie Abbildung 3-6 zeigt, existieren damit drei Arten von Schnittstellen:

- zu anderen Datenquellen z.B. relationalen Datenbanken,
- zu ProSpecT-Client Programmen,
- zu weiteren ProSpecT-Servern.

- Datenbankschnittstellen

Wie unter Abschnitt 3.4.5 (siehe Seite 11ff) bereits ausgeführt, stellen die Basiskomponenten die Schnittstelle zu anderen Datenquellen für den Benutzer bereit. Dazu wird das abstrakte ProSpecT-Objekt entsprechend erweitert. Durch die Transparenz dieser Lösung kann die Schnittstelle zwischen Basiskomponente und externer Datenquelle exakt an die gegebenen Erfordernisse angepaßt werden. Es gibt keinerlei Beschränkung diesbezüglich. So wird nahezu jede Datenquelle für ProSpecT nutzbar, die Heterogenität der vorhandenen Systeme stellt kein Hindernis dar.

- Server - Server - Schnittstelle

Für die Kommunikation mehrerer Server ist lediglich eine Funktion des ProSpecT-Kerns von Interesse, die Objektverwaltung. Wird ein Objekt angesprochen, das von einem anderen Server verwaltet wird, so wird eine Kopie zum entsprechenden Server migriert. Danach erfüllt diese die gewünschte Anfrage und wird anschließend vernichtet. Dabei ist ein Konsistenzprotokoll einzuhalten. Zwischen zwei Servern werden nur bereits in der Objektverwaltung gespeicherte Objekte bzw. deren Kopie ausgetauscht. Die hierzu notwendigen Schnittstellen können recht einfach gehalten werden:

- Kopie von Objekt anfordern,
- Objekt in entfernte Objektverwaltung speichern,
- Objektkopie nicht länger genutzt,
- Objekt löschen.

3.5.2 Client

Aufgabe des Clientprogramms ist es, dem Benutzer eine für seine Aufgaben zweckmäßige Schnittstelle zu geben. Die modellierten Zusammenhänge werden grafisch repräsentiert und sind vom Benutzer manipulierbar. Für den Benutzer scheint der Client die Funktionalität des gesamten ProSpecT-Systems zu verkörpern. Die Kommunikation zwischen Client und Server verläuft für den Benutzer transparent. Dabei kann man zwei Arten unterscheiden, eine Client-Server- sowie eine Server-Client-Kommunikation. Als Client-Server-Nachrichtenaustausch ist dabei die Anfrage-Ergebnis-Kommunikation zu verstehen. Sie ähnelt dem Funktionsaufruf einer Programmiersprache und ist durch Anweisungsname, Eingangsparameter und Ergebnis charakterisiert. Der Client setzt eine Anfrage an den Server ab und bekommt daraufhin das Ergebnis.
Die Server-Client-Kommunikation wird notwendig, wenn der Server Anfragen bzw. Meldungen an den Client sendet. Dies kann zum einen im Fehlerfall geschehen, andererseits kann so der Server unabhängig von der Absicht des Benutzers Informationen anfordern. Ein Beispiel dafür ist die Modelleinordnung eines neuen Objektes. Erzeugt ein Benutzer ein neues Objekt einer Klasse, so wird er dieses in den ihm sichtbaren Zusammenhang einordnen wollen. Objekte dieser Klasse können aber durchaus auch anderen Modellen angehören. So kann es nützlich sein, daß der Server die Einordnung des neuen Objekts in die anderen Modelle vom Client fordert.
Wie bereits ausgeführt, gleicht die Client-Server-Kommunikation einem Funktionsaufruf. Dabei stellt sich die Flexibilität des Servers als Problem für die Schnittstellendefinition dar. So können weder Anweisungsnamen noch deren verschiedenen Argumente und Ergebnisse im voraus definiert werden, da sie jederzeit veränderbar sind. Lediglich die Funktionalität des ProSpecT-Objekts steht

fest. So ist es möglich, zur Laufzeit alle Methoden eines spezialisierten ProSpecT-Objektes zu erfragen und sie anschließend aufzurufen. Dabei können auch Argumente und Ergebnisse verbal beschreiben werden. Es kann also lediglich ein Format definiert werden, das die Anweisungen an den Server darstellt.

Abbildung 3-7: Format einer Serveranfrage

Dabei dienen Argument- und Ergebniscontainer der Aufnahme vielfältiger Datentypen. Eine Prüfung auf Richtigkeit von Anzahl und Typ der übergebenen Werte ist später durch Client und Server selbst zu realisieren. Der Objekt-Identifikator kann neben dem Bezeichner des konkreten Objekts auch eine Klasseninformation enthalten, mit der die Funktionalität des Objekts maskiert werden kann.
Die Funktionalität der Server-Client-Kommunikation wird durch den Client realisiert, eine Definition der Schnittstellen ist erst bei dessen Design möglich. Die Notwendigkeit, das Clientprogramm speziell für den späteren Einsatz zu entwerfen, führt zu einer Vielzahl verschiedenartiger Clientlösungen, die dem Server verschiedene Schnittstellen bieten. Die Kommunikation erfolgt dabei zwischen spezialisierten ProSpecT-Objekten und dem Clientprogramm. Beide müssen die gleichen Schnittstellen definieren und können dies auch, da sie unabhängig vom ProSpecT-Kern später hinzugefügt werden. Trotzdem sind einige Anweisungen bereits durch den Kern als minimale Schnittstelle definiert:

- Warnung ausgeben,
- Eingaben entgegennehmen,
- Client neue Rechte zuweisen.

4 Implementation

Ziel dieser Arbeit sollte es sein, ein leistungsfähiges Prozeßmodellierungssystem für den Einsatz in der Halbleiterindustrie zu entwerfen und implementieren. Wie in Abschnitt 3 (siehe Seite 4ff) ausgeführt, ist ProSpecT aufgrund seiner offenen Architektur in der Lage, alle Anforderungen an ein solches System zu erfüllen. Voraussetzung dafür ist jedoch eine Anpassung und Spezialisierung an das Aufgabengebiet. Notwendig wären:

- die Nachbildung aller Basiskomponenten einer Fab,
- die Schaffung von Modellkomponenten für algorithmische und strukturelle Modelle,
- den Entwurf von Clientprogrammen für die jeweilige Arbeitsaufgabe.

Dies ist nur möglich, wenn man eine konkrete Fab nachbilden kann. Die verschiedenen Datenquellen, Datenformate und bereits existierenden Softwaresysteme sind ohne Vorbild ebensowenig nachzuempfinden, wie die Tätigkeitsfelder der Benutzer.

4.1 Plattform- und Systemunabhängigkeit

Eine der wichtigsten Anforderungen an ProSpecT stellt die Plattformunabhängigkeit dar. Betrachtet man die Probleme, die mit ProSpecT nachgebildet werden sollen, so handelt es sich meist um vernetzte Systeme verschiedenster Rechner. Es ist ein Fakt, daß in den meisten Einsatzorten von Computertechnik eine Heterogenität von Hard- und Softwaresystemen nicht zu umgehen ist. Dies ist nicht verwunderlich, schließlich kann es keine einheitliche Lösung für die täglich anfallenden Probleme geben. Eine Vereinheitlichung der eingesetzten Computertechnik zur Senkung der Betriebskosten ist zwar wünschenswert, will aber ebenso gut überlegt sein. Eine nicht am Problem orientierte Lösung kann schnell teurer kommen, als die eingesparten Kosten, auch wenn man Lösungen mit großem Marktanteil verwendet.
Daher gibt es Überlegungen, wenigstens die Schnittstellen sowie deren Semantik (Protokolle) zwischen verschiedenen Systemen zu vereinheitlichen. Das betrifft sowohl die Hardware- als auch die Softwareseite. Standards für den Datenaustausch zwischen verschiedenen Systemen erleichtern deren Kooperation, ohne deren Wirkungsweise zu vereinheitlichen. Beispiele für solche Standards sind z. B. EtherNet, SCSI, TCP/IP, SQL, CORBA.
Somit wird es möglich, Daten über Hardware- und Betriebssystemgrenzen hinweg auszutauschen, das Problem der Heterogenität scheint gelöst. In der Praxis setzen sich Standards nur sehr schwer durch, in einigen Fällen geschieht dies gar nicht. Die Ursachen dafür liegen zumeist in der Entstehung des Standards bzw. seiner Auslegung. Ein Standard stellt eine Normierung einer Problemlösung dar. Meist existieren im Vorfeld einer Standardisierung verschiedene Lösungsmöglichkeiten diverser Firmen bzw. Institutionen. Bei der Normierung gibt jede beteiligte Institution ihre eigene zugunsten einer gemeinsamen Lösung auf. Damit gehen eventuell wichtige Vorzüge der proprietären Lösung verloren, ein verstärkter Wettbewerb ist oft Folge der Vereinheitlichung.
Um dem entgegenzuwirken, kommt es nicht selten vor, daß nur über grundlegendste Teile Einigkeit besteht und der Standard kaum in seiner beschlossenen Form zum Einsatz kommt. Man nennt diese Art der Standardisierung auch „designed by commitee“. SQL kann hierfür als Beispiel angeführt werden. Kaum eine SQL-Datenbank kommt ohne proprietäre Erweiterungen aus, die den Nutzen der Vereinheitlichung schmälern.
Einige Firmen versuchen aufgrund ihrer marktbeherrschenden Position ihre Lösung als Quasi-Standard durchzusetzen, Turbo-Pascal von Borland mag hierfür als Beispiel dienen. Versucht man Plattformunabhängigkeit zu erreichen, so muß man nicht nur verschiedene Standards, sondern auch proprietäre Ansätze beachten. In der Regel unterstützt man eine Auswahl an verbreiteten Standards und sieht eine konzeptuelle Erweiterbarkeit um andere Lösungen vor.
Die nachfolgend beschriebene Implementation basiert auf dem OpenStep-Standard.

OpenStep

OpenStep ist ein betriebssystemunabhängige, objektorientierte Programmierschnittstelle basierend auf der Objekttechnologie der Firma NeXT Computer Inc.[1]. Der Standard basiert auf den Erfahrungen, die seit 1989 mit dem NEXTSTEP-System gesammelt wurden. Er wurde in Zusammenarbeit mit SunSoft Inc. erarbeitet und 1994 in einer ersten Version veröffentlicht.
Der OpenStep-Standard besteht im wesentlichen aus drei Teilen (Abbildung 4-1):

[1] seit Februar 1997 fusioniert mit Apple Computer Inc.

Application Kit

Das Application Kit stellt die grundlegende Funktionalität zum Entwickeln interaktiver Programme dar, Anwendungen, die Fenster benutzen und auf Tastatur oder Maus Ereignisse reagieren. Das Application Kit stellt den Teil dar, der die OpenStep-Benutzeroberfläche definiert

Foundation Kit

Das Foundation Kit enthält die fundamentalen Grundbausteine, die Programme zum Umgang mit Daten und Ressourcen benötigen. Es definiert Verfahren zur Handhabung von Multibyte-Zeichensätzen, Objektpersistenz, Objektverteilung und stellt eine Schnittstelle zu gemeinsamen Betriebssystemfunktionen dar.

Display PostScript System

Das Display PostScript System stellt den OpenStep Programmen ein einheitliches, geräte-unabhängiges Grafikmodell zur Verfügung.

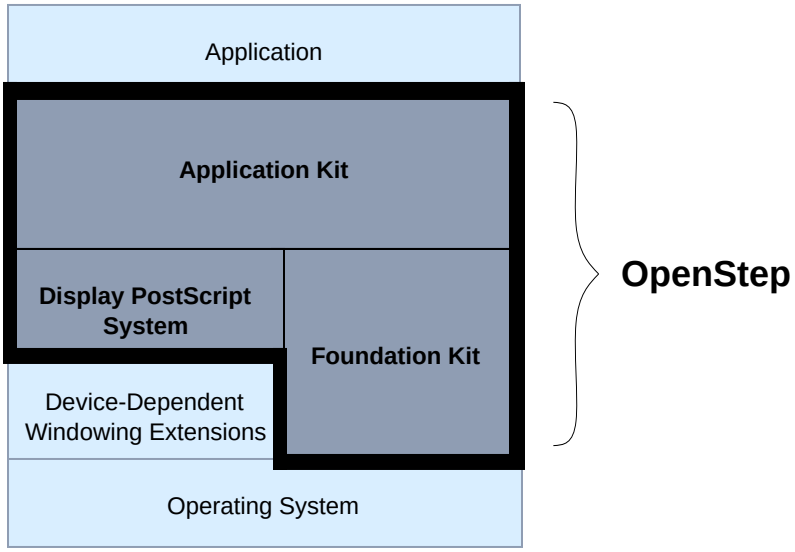


Abbildung 4-1: OpenStep-Architektur[1]

Die Basisarchitektur der Implementation konkretisiert die konzeptuelle Struktur von ProSpecT (siehe Seite 13ff) um Schnittstellen zu Datenquellen aller Art. Wie in Abbildung 4-2 gezeigt, geschieht dies sowohl unter Nutzung der Funktionalität des EOF, als auch durch proprietäre Lösungen. Ansonsten bleibt das Konzept unverändert, wobei auch die Verteilung der Aufgaben auf mehrere ProSpecT-Server (siehe Seite 14ff) realisiert wird.

[1] entnommen aus [NEX-94]

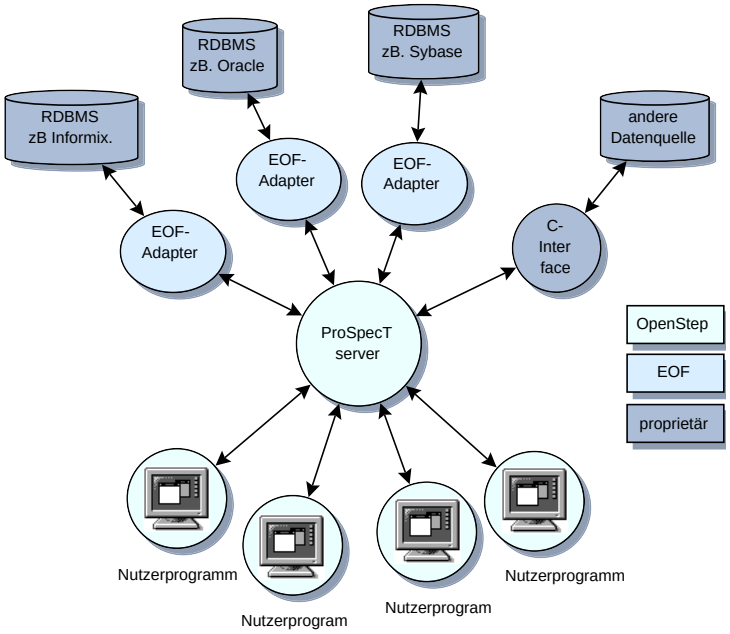


Abbildung 4-2: ProSpecT-Basisarchitektur unter Nutzung der OpenStep-Technologie

Wie angedeutet, kommt im gesamten System das Objektmodell von OpenStep zum Einsatz, d.h. Clients, Server und Datenquellen nutzen Objective C-Objekte zur Kommunikation. Das schließt eine Anwendung anderer Programmiersprachen und Objektkonzepte nicht aus, so lange diese in den OpenStep-Kontext eingebettet werden können. Damit wird es zum Beispiel ebenso möglich, Datenquellen mit Smalltalk-Schnittstelle anzubinden, als auch Clientprogramme in Java zu realisieren, wenn diese auf einem Netzwerkcomputer laufen sollen. Auch CORBA- konforme Lösungen können für ProSpecT genutzt werden, da auch für Objective C eine IDL[1] zum Objektstandard der OMG vorliegt [MV-96].

4.2 TPV - der ProSpecT-Server

Das Herzstück von ProSpecT stellt die Server-Applikation dar. Im folgenden wird sie als TPV[2] bezeichnet, während ProSpecT die Gesamtlösung charakterisiert. TPV realisiert alle in Abschnitt 3.4 (siehe Seite 7ff) beschriebenen Funktionen und stellt eine objektorientierte Schnittstelle für Erweiterungen (Spezialisierungen) bereit.
Die Aufgaben wurden entsprechend ihrer Art auf verschiedene Objekte aufgeteilt, die innerhalb des Programmes miteinander kommunizieren. Abbildung 4-3 zeigt eine schematische Darstellung des Zusammenwirkens der wichtigsten Objekte. Die Pfeile drücken dabei die Richtung der Kommunikation aus.
Das Server-Objekt stellt die zentrale Steuerung des TPV dar, nach Programmstart initialisiert es das TPVMainTemplate-Objekt, lädt alle vorhandenen Erweiterungsmodule nach und startet das verteilte, persistente Objektverwaltungssystem DiPOS. Abschließend werden alle zur Clientbehandlung notwendigen Objekte initialisiert.

[1] Interface Definition Language

[2] in Anlehnung an den s.g. „Total Perspective Vortex“, beschrieben in [ADA-80]

Soll TPV beendet werden, kommt dem Server Objekt die Aufgabe zu, das TPVMainTemplate-Objekt zu sichern, das DiPOS zu beenden und alle Clientverbindungen abzubauen. Die übrigen Objekte erfüllen folgende Aufgaben:

TPVTypeDict	- Verwaltung der Erweiterungstypen und ladbaren Objekte,
TPVMainTemplate	- Klassenobjekt, dient zum Erzeugen neuer Unterklassenobjekte,
ext. Bundle	- stellt Funktionalität der erweiterten Klassenobjekte bereit,
TPVDiPOS	- realisiert das verteilte, persistente Objektverwaltungssystem,
TPVDispatcher	- weist neuen Clients eine exklusive Verbindung zu,
TPVConnectionDict	- enthält Informationen über alle aktuellen Clientverbindungen,
TPVAmbassador	- nimmt Clientanfragen entgegen und vermittelt sie weiter.

Das dargestellte TPV-Objekt ist nicht Bestandteil des Servers, vielmehr kennzeichnet es ein beliebiges, im DiPOS gespeichertes Objekt, das von einem Clientprogramm über den Ambassador angesprochen wird. Es stellt die Einheit des ProSpecT-System dar, die zur Aufbewahrung der nutzerrelevanten Daten bzw. Datenverweise benutzt wird.

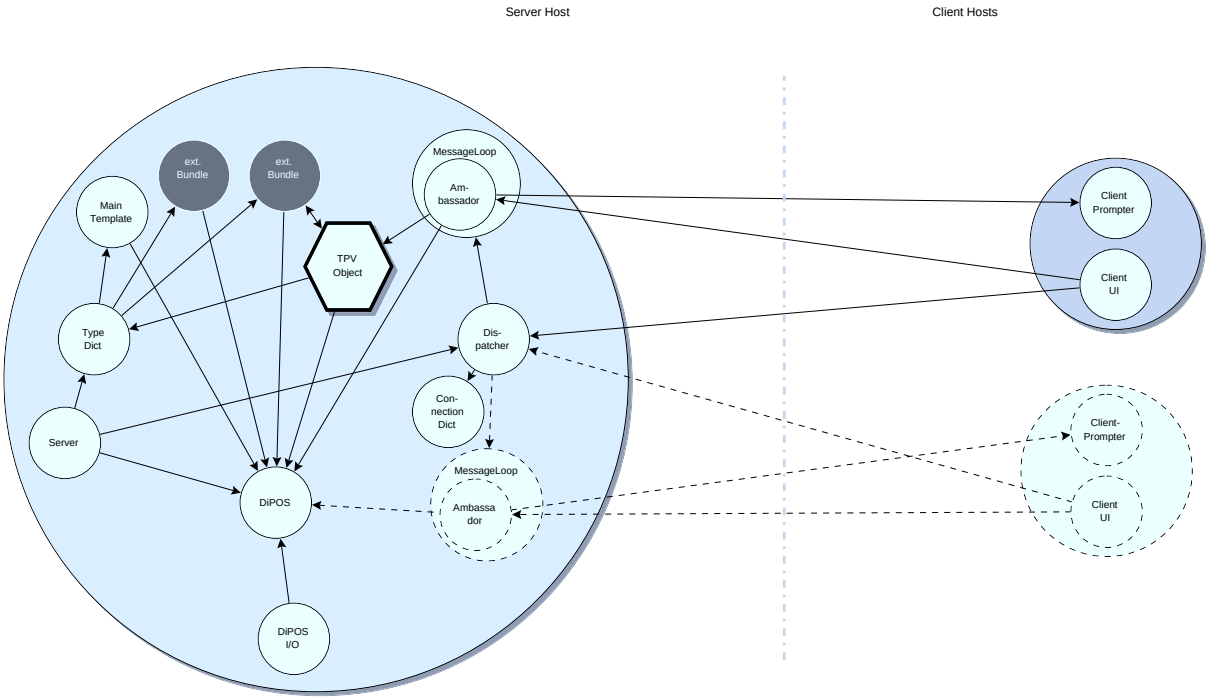


Abbildung 4-3: Objektaufbau des ProSpecT-Servers

Um eine möglichst ungehinderte Kommunikation mit mehreren Clientprogrammen zu ermöglichen, wurde in TPV eine leistungsfähige Clientbehandlung realisiert.

4.2.1 Clientbehandlung

TPV stellt die Umsetzug des Multi-Threaded-Servers-Konzeptes dar. Um allen Clientprogrammen gleichberechtigten Zugang zu den gespeicherten Daten zu ermöglichen, bekommt jeder Client eine eigene Verbindung zum Server bereitgestellt. Dazu wird in einem neuen Thread ein exklusives TPVAmbassador-Objekt erzeugt, das die Kommunikation auf Serverseite realisiert. Somit können alle Clients quasi gleichzeitig mit dem TPV kommunizieren, es tritt keine systembedingte Blokkierung von Clientanfragen auf.
Wie bereits in Abschnitt 3.5 (siehe Seite 12ff) beschrieben, exisitieren zwischen Client und Server zwei unabhängige Verbindungen. Für Anfragen an das TPV-System nutzt die ClientUI genannte Komponente eine Verbindung zum TPVAmbassador-Objekt. Diese Komponente des Clients stellt dabei die interaktive Schnittstelle zum Benutzer dar und kann sowohl grafisch als auch alphanumerisch realisiert werden.
Andererseits benötigt das TPVAmbassador-Objekt des Servers für Ausnahmefälle eine Verbindung zum Client. Diese wird durch das TPVClientPrompter-Objekt der Client-Applikation bereitgestellt. Genau betrachtet, treten TPV und Clientprogramm sowohl als Diensterbringer als auch als Dienstnehmer auf. Um beide Verbindungen aufzubauen, ohne daß dabei Verklemmungen auftreten, ist ein Protokoll zwischen Server und Client notwendig. Außerdem muß sichergestellt werden, das kein zweiter Client eine bestehende Verbindung nutzt.
Jede Kommunikation wird durch die OpenStep-Technologie der verteilten Objekte (distributed objects) realisiert. Ohne hier näher darauf einzugehen [NEX-94] stellt sie einen sehr einfachen und trotzdem leistungsfähigen Ansatz zur Verteilung von Objekten im Netz dar. Verbindungen zwischen verteilten OpenStep-Objekten werden durch Namen charakterisiert. Stellt ein Server ein Objekt zur netzwerkweiten Nutzung zur Verfügung, kann es von jedem Client genutzt werden, der den Namen des Dienstes kennt. Um eine Kommunikation zwischen genau zwei Objekten zu gewährleisten, müssen diese vorher einen einzigartigen Namen für die gewünschte Verbindung aushandeln. Damit ist ausgeschlossen, daß ein drittes Objekt dieselbe Verbindung nutzt.
Diese Aufgabe erfüllt der Dispatcher in TPV. Er stellt eine allen Clients bekannte Verbindung bereit, über die man mit TPV den Namen einer anderen, exklusiven Verbindung erhält. Das gesamte Protokoll zum Aufbau beider bilateralen Verbindungen stellt Abbildung 4-4 dar.
Das Clientprogramm startet als erstes nach seinem Aufruf einen ClientPrompter-Service in einem separaten Thread. Der Name seiner Verbindung wird durch eine netzwerkweit einmalige Zeichenkette gebildet. Ist der ClientPrompter initialisiert, nimmt die ClientUI Komponente mit dem TPVDispatcher über die ihm bekannte Verbindung auf. Dieser gestrichelt dargestellte Verbindungsaufbau erfolgt implizit durch das OpenStep-Laufzeitsystem. Anschließend sendet die ClientUI-Komponente eine ConnectionRequestFor-Nachricht, mit der sie ihren Wunsch nach einer exklusiven Verbindung ausdrückt. Dabei übergibt es dem Dispatcher ein Objekt, das alle clientrelevanten Daten enthält, so auch den Namen der ClientPrompter Verbindung.
Der TPVDispatcher generiert ein neues TPVConnection-Objekt, das alle wichtigen Informationen über die Clientverbindungen enthält und fügt es in sein Verzeichnis bestehender Verbindungen ein, das durch das TPVConnectionDict-Objekt realisiert ist. Anschließend startet er ein TPVMessageLoop-Objekt in einem separaten Thread, übergibt diesem das TPVConnection-Objekt und blockiert. Daraufhin nimmt das TPVMessageLoop -Objekt Verbindung mit dem ClientPrompter des Clients auf (gestrichelt dargestellt).

Gelingt dies, wird ein neues TPVAmbassador-Objekt initialisiert und eine neue Verbindung zur Verfügung gestellt. Anschließend wird der Name dieser Verbindung an den TPVDispatcher übergeben, dieser setzt seine Tätigkeit fort und schickt der ClientUI-Komponente den Namen ihrer exklusiven Verbindung, die das TPVAmbassador-Objekt bereitstellt.
Danach nimmt diese direkt Kontakt zum Ambassador Objekt auf, der seinerseits bereits eine Verbindung zur ClientPrompter Komponente besitzt. Alle notwendigen Bestandteile der Kommunikation zwischen TPV und Client sind initialisiert, die Verbindung zum TPVDispatcher wird abgebaut (Dies ist nicht dargestellt, da es implizit geschieht).

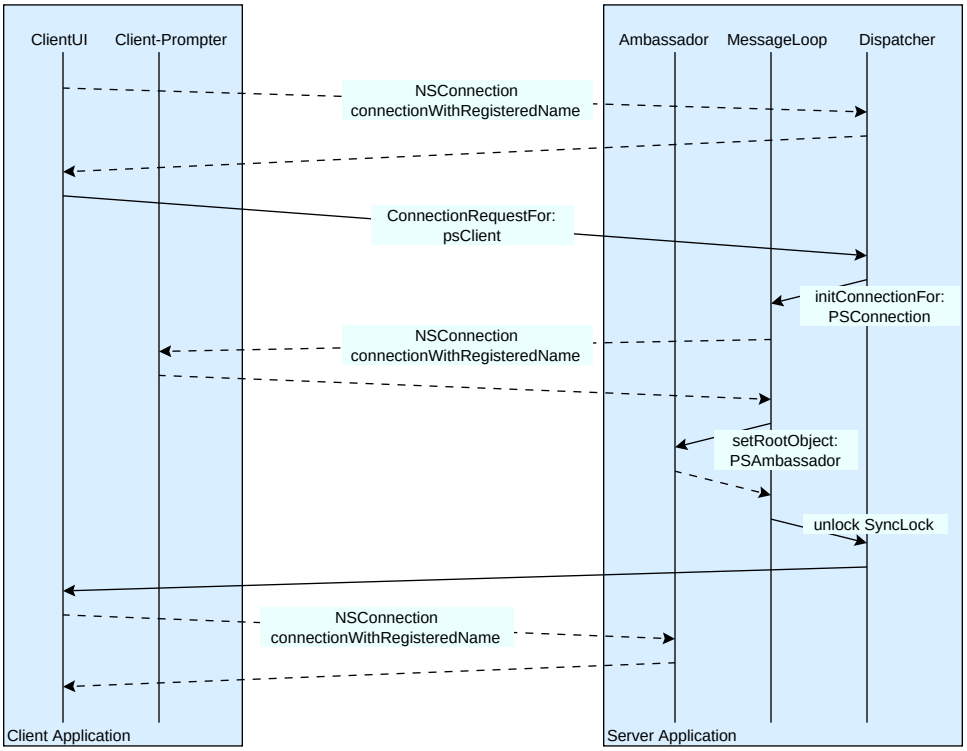


Abbildung 4-4: Protokoll des Verbindungsaufbaus zwischen Clientprogramm und TPV-Server

Anschließend können Client und TPV die eigentliche Nutzkommunikation durchführen, deren Protokoll deutlich einfacher ist (Abbildung 4-5). Für den Client steht nur eine mögliche Nachricht zur Verfügung, performObject:With:. Diese adressiert ein TPV-Objekt, sendet diesem eine Nachricht und übergibt gegebenenfalls Parameter. Kann diese Anfrage durch TPV befriedigt werden, erhält die ClientUI-Komponente die Ergebnisse ihrer Nachricht.
Tritt bei der Bearbeitung der Clientanfrage eine Ausnahmesituation auf, kann das TPVAmbassador-Objekt verschiedene Nachrichten an den ClientPrompter senden, um die Anfrage eventuell erfolgreich abschließen zu können.

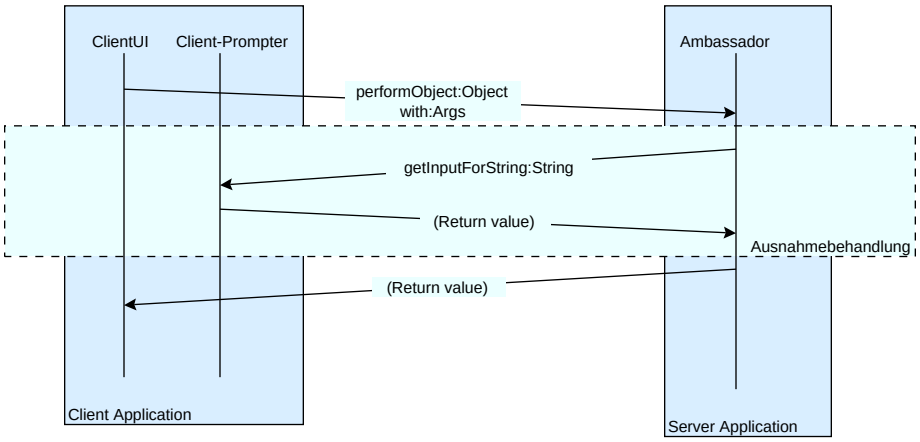


Abbildung 4-5: Protokoll der Nutzkommunikation zwischen Clientprogramm und TPV-Server

Diese Nachrichten sind:

- alert

die Anfrage kann nicht erfolgreich beantwortet werden, Ausgabe einer Fehlermeldung, das erwartete Ergebnis ist ungültig

- getInputForString

die Eingabe eines weiteren Arguments ist notwendig, das ClientPrompter-Objekt fordert eine Benutzereingabe und sendet das Ergebnis zurück.

- fillOutForm

es werden mehrere Argumente benötigt, das ClientPrompter-Objekt fordert den Benutzter zur Eingabe der Werte auf und sendet alle Eingaben zurück an das TPVAmbassador-Objekt.

Damit ist eine leistungsfähige Möglichkeit zum Datenaustausch zwischen Clientanwendung und TPV-Server gegeben. Die bestehenden Verbindung werden so lange aufrecht erhalten, wie das Clientprogramm läuft. Da die Kommunikation zwischen verteilten Objekten in OpenStep nicht über aktives Warten realisiert wird, führt dies nicht zu einer unnötigen Belastung von TPV-Server und Netzwerk und stellt kein Performance-Problem dar.
Das Kommunikationsende erfolgt implizit bei Beenden des Clientprogramms. Der Abbruch der bestehenden Verbindungen wird dem TPV-Server automatisch durch das OpenStep-Laufzeitsystem mitgeteilt. Dieser baut danach seinerseits bestehende Verbindungen ab, zerstört das TPVAmbassador-Objekt und kann den laufenden Thread gegebenenfalls einem neuen Clientprogramm zuteilen.

4.2.2 Das abstrakte TPV-Objekt

Wie bereits angedeutet, werden durch TPV-Objekte alle nutzerrelevante Daten und Informationen repräsentiert. Es stellt somit eine Art Container dar, dessen Inhalt entweder die gewünschten Daten selbst sind, oder einen Verweis auf die Datenquelle, die diese Daten bereits enthält. Zugang bekommt der Nutzer zu diesen Attributen des TPV-Objekts nur über die Methoden bzw. Anfragen, die es erfüllen kann. Somit ist für den Nutzer nicht sichtbar, ob die gewünschte Information lokal im TPV-Objekt vorliegt, oder von einer anderen Datenquelle bereitgestellt wird. So wird das Prinzip der Transparenz der Datenquellen umgesetzt, das in Abschnitt 3.4.5 (siehe Seite 11ff) beschrieben ist.
TPV-Objekte werden durch das TPV-System verwaltet, alle hier beschriebenen Teile des Systems dienen einzig diesem Zweck. Die Funktionalität des Gesamtsystems hängt in wesentlichen Maße von der Funktionalität der TPV-Objekte ab. Wie erläutert, können ihre unterschiedliche Methoden ebenso wie ihre Attribute gekapselt werden. Für die Verwaltung innerhalb des TPV sind alle TPV-Objekte gleich. Es reicht daher aus, ein abstraktes TPV-Objekt zu definieren, das im Sinne des Benutzers keine nennenswerte Funktionalität besitzt, andererseits aber alle Bedingungen für eine Verwaltung im TPV-System erfüllt. Es muß durch zusätzliche Komponenten, die nicht Teil von TPV sind, die gewünschte Funktionalität erhalten können. Genau dies wird durch das TPVMainTemplate-Objekt erreicht. Es stellt eine leere Hülle für Attribute, Methoden und Modelle dar, die von Erweiterungskomponenten, den externen Bundle-Objekten gefüllt werden können.
Die Funktionalität des TPVMainTemplate-Objekts läßt sich am besten durch eine Betrachtungsweise erklären, wie sie für objektorientierte Programmiersprachen verwendet wird. So gesehen stellt das TPVMainTemplate-Objekt das Klassenobjekt einer Metaklasse dar, die durch die externen Bundle-Objekte abgeleitet wird. Dabei wird zuerst eine genaue Kopie des TPVMainTemplate-Objekt erzeugt. Anschließend wird dessen Methods Dictionary genanntes Verzeichnis der über dem Objekt möglichen Anfragen um zusätzliche Methoden erweitert. Dabei wird gegebenenfalls das Attribute Dictionary, der Container für Werte der Methoden gefüllt. Aus dem abstrakten TPVMainTemplate-Objekt ist eine funktional erweiterte Unterklasse entstanden, genauer gesagt deren Klassenobjekt (Abbildung 4-6).

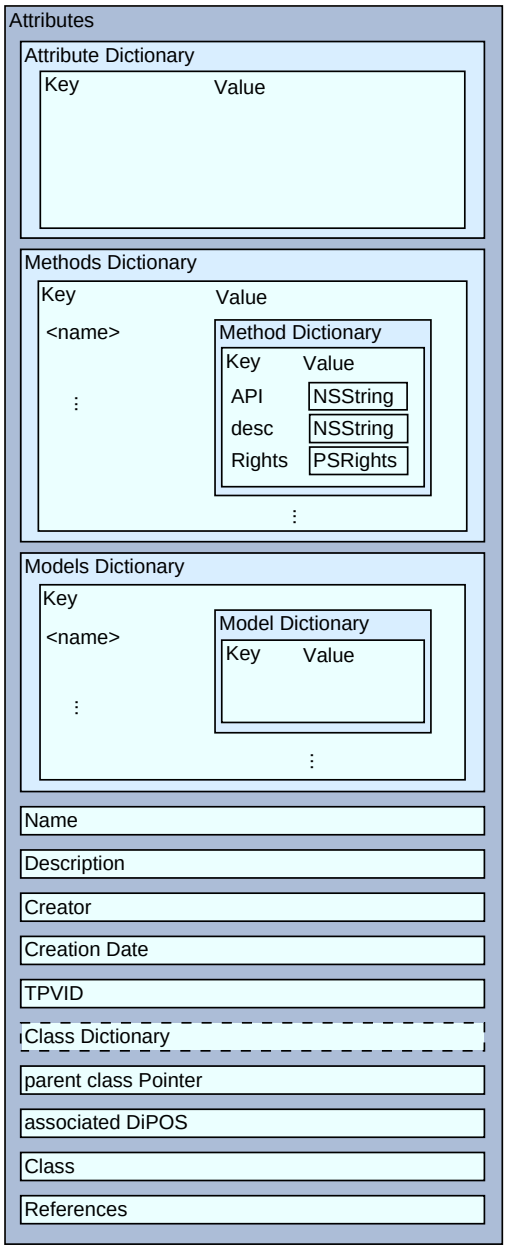


Abbildung 4-6: Bestandteile des TPV-Objekts

Wird dieses nun instanziiert, entstehen TPV-Objekte, die gemeinsame Funktionalität besitzen und deren Attribute privat sind - Instanzenobjekte. Diese nehmen die nutzerrelevanten Informationen auf, Klassenobjekte sind für den Benutzer als Schablonen sichtbar. Sie werden zum Erzeugen neuer „Informationsobjekte“ benötigt.

addAttribute: nsAttDict	fügt Attribute zum Attributsverzeichnis hinzu
getAttributeForKey:nsAttName	liefert Attribut passend zu einem Schlüssel
deleteAttribute:nsAttributeKey	löscht Attribut passend zu einem Schlüssel
addMethod:nsMethDict	fügt neues MethodDict Verzeichnis zu Methodenverzeichnis hinzu
deleteMethod:nsMethodKey	löscht MethodDict Verzeichnis aus dem Methodenverzeichnis
addModel:nsModelDict	vermerkt neues Modell über Objekt im Modellverzeichnis
deleteModel:nsModelKey	löscht Modellverweis aus dem Modellraum
getObjId	liefert den Objektidentifikator
setAssocDiPOS:psAssocStore	weist Objekt zugehöriges DiPOS-Objekt zu
getModels	liefert ein Verzeichnis von Modellen, in denen Objekt referenziert ist
getMethods	liefert Verzeichnis aller implementierten Methoden (als MethodDict)
getMethodDescription	liefert eine verbale Beschreibung einer oder aller Methoden
getName	liefert den Objektnamen
setName:nsNewName	setzt den Objektnamen
getType	liefert die Klasse des TPV-Objekts
setRights:tpvNewRights ForMethod:nsMethodName	setzt Benutzerrechte für gegebene Methode
getRightsForMethod:nsMethodName	liefert Benutzerrechte für gegebene Methode
setName:nsNewName	setzt den Namen des Objekts
subclassParent:tpvTheParent asType:nsTyp	setzt Verweis auf zugehöriges Klassenobjekt
runMethod:nsMethod withArgs:nsArgs	führt die gewünschte Methode mit ihren Argumenten aus
replicate	klont das Klassenobjekt zur Spezialisierung durch externe Bundle-Objekte
instance	erzeugt ein Instanzobjekt
initWithRef:iReferenceCount	initialisiert ein neues Objekt und setzt den Referenzzähler
addSubclassRef:nsNewClassDict	fügt Verweis auf neue Unterklasse ein (nur bei abstraktem Objekt)
getClassDict	liefert das Verzeichnis der Unterklassen (nur bei abstraktem Objekt)
addReference	incrementiert den Referenzzähler
subReference	dekrementiert den Referenzzähler
lock	schützt alle Attribute vor nicht exklusivem Zugriff
unlock	hebt exklusivität der Attribute auf
setObjectDescription:nsNewDesc	fügt dem Objekt eine verbale Dokumentation hinzu
getObjectDescription	liefert die Objektbeschreibung
createdAt	gibt Ensthungsdatum an
createdBy	liefert Urheber des Objekts

Tafel4-1: Methoden eines TPV-Objektes

Damit entstehen aus dem TPVMainTemplate ein Vielzahl von TPV-Objekten verschiedener, spezialisierter Klassen. Die Zugehörigkeit eines Instanzobjekts zu einer Klasse wird durch seinen Class-Bestandteil ausgedrückt. Das Klassenobjekt dieser Klasse ist durch den parent- Class-Zeiger erreichbar. Der gestrichelt dargestellte Class Dictionary Bestandteil dient dem TPVMainTemplate-Objekt als Verzeichnis der von ihm abgeleiteten Klassenobjekte und wird ausschließlich in diesem Objekt benötigt. Der Grund dafür ist die Objektverwaltung, für die alle TPV-Objekte gleich sind, sie kann weder Klassen- noch Instanzobjekte identifizieren. Außerdem sind für eine Verwaltung sinnvolle Attribute, wie Erzeugungsdatum, Urheber, Objektidentifikator, Aufbewahrungsort, Verweiszähler und der Objektname Bestandteile jedes TPV-Objekts. Dem Anspruch des Konzepts,

die Dokumentation stärker an das betreffende Objekt zu binden, trägt die Implementation durch ein Objektbeschreibungsbestandteil jedes TPV-Objekts Rechnung. So ist zu jeder Zeit eine Dokumentation von Objekteigenschaften möglich, die leicht auffindbar ist. All diese Bestandteile sind nur über Methodenaufrufe zugänglich. Diese sind in Tafel 4-1 kurz zusammengefaßt.
Ähnlich wie bei objektorientierten Programmiersprachen enthalten Instanzobjekte lediglich Attribute, das heißt ein TPV-Instanzenobjekt besitzt kein Methoden- und Modellverzeichnis. Deshalb werden alle Methoden, die darauf zugreifen, an das jeweilige Klassenobjekt weitergeleitet.

4.2.3 Das verteilte, persistente Objektspeichersystem DiPOS

Alle TPV-Objekte haben nach ihrer Erzeugung eine unbegrenzte Lebensdauer, erst ein Fehlen von Referenzen kann zu ihrer Vernichtung führen. Daraus leitet sich die Forderung ab, daß unabhängig vom Zustand des TPV-Systems die Existenz aller Objekte zu sichern ist. Diese Eigenschaft der TPV-Objekte wird Persistenz genannt. Durch das System ist sicherzustellen, daß alle persistenten Objekte auch nach Programmende weiter existieren und bei Neustart wieder zur Verfügung stehen. Dieses kann erreicht werden, indem zu jeder Zeit ein Objekt sowohl im flüchtigen als auch im nichtflüchtigen Speicher gehalten wird. Dabei kann die Behandlung des Objekts auf zweierlei Art realisiert werden. Zum einen ist es möglich, den Übergang vom aktiven Zustand, wo sich das Objekt im nichtflüchtigen Speicher befindet, zum inaktiven, ausgelagerten Zustand dem Programmierer zu übertragen. TPV nutzt diesen Ansatz. Dabei muß im Programm stets zwischen beiden Zuständen unterschieden werden, schließlich kann ein Objekt nur dann auf Nachrichten reagieren, wenn es sich im Arbeitsspeicher befindet. Wird es nicht mehr benötigt, lagert man es aus. Dazu nutzt man eine persistente Objektspeicherverwaltung, auch persistent object store (POS) genannt.
Ein anderer Ansatz bietet die gleiche Funktionalität, wobei aber hier der Objektzustand auch für den Programmierer transparent ist. Dies wird auch orthogonale Persistenz [DV-96] genannt. Ein Objekt wird als persistent gekennzeichnet und das Laufzeitsystem stellt seine Persistenz sicher, ohne daß es für die Programmierung inaktiv scheint. OpenStep bietet bisher keine orthogonale Persistenz für Objekte, für Java ist diese Funktionalität in Vorbereitung.
Ausgehend von möglichen Einsatzfeldern für TPV stellt sich eine Speicherung aller persistenten Objekte auf nur einem Server als problematisch heraus. Dabei spielt sowohl der Aspekt der Sicherheit und Verfügbarkeit eine Rolle, als auch die Auslastung von Rechentechnik und Netzwerk. Die Erfahrungen mit Mainframe-Computern zeigen, daß eine zentrale Datenhaltung zwar Vorteile hinsichtlich Konsistenz, Wartung und Zugriffskontrolle hat, andererseits aber nicht tolerierbare Nachteile wie fehlende Fehlerredundanz, Performance-Probleme und hohe Kosten hat. Es war daher ein Ziel von TPV, die Speicherung von persistenten Objekten im Netzwerk zu verteilen. So ist es möglich, mehrere TPV-Server im Netzwerk parallel zu betreiben, die eine gemeinsame Basis an persistenten Objekten nutzen. Die persistente Objektspeicherverwaltung mußte verteilt realisiert werden, es entstand das DiPOS (Distributed Persistent Object Store).
Außerdem bestand die Aufgabe, eine Versionsverwaltung über die persistenten Objekte zu realisieren, wie in Abschnitt 3.4.3 (siehe Seite 9ff) beschrieben. Ein Objektverwaltungssystem dieser Art stellt ein komplexes Stück Software dar, insbesondere die Leistungsfähigkeit bei sehr großen Datenmengen stellt eine echte Herausforderung dar. Von daher wurde die Nutzung eines bereits verfügbaren POS-Systems ins Auge gefaßt. Die Funktionalität des DiPOS sollte in einem Objekt gekapselt werden das über eine minimale Schnittstelle angesprochen wird.

Bei der Literaturrecherche zu diesem Thema stellte sich allerdings heraus, daß es nur wenige einsatzfähige POS-Systeme weltweit gibt [Hain-95]. Betrachtet wurden hierbei:

Texas	POS-System für C++Objekte
Distributed Texas	experimentelles, verteiltes POS-System für C++Objekte
Exodus	DBMS für Objekte der Sprache E (abwärtskompatibel zu C++
Ontos	kommerzielles POS für C++ und ObjectSQL-Objekte
Shore	sprachunabhängiges POS mit ODL[1]

Darunter findet sich kein System, das die zwei Hauptanforderungen an DiPOS erfüllen kann: Persistenz für ungetypte Objective C-Objekte sowie die Möglichkeit zur Versionsverwaltung über Objekten. Aus diesem Grund wurde eine eigenständige Lösung realisiert, die zwar den gewünschten Grundforderungen nachkommt, aber nur einen bescheidenen Leistungsumfang bietet. Insbesondere die Handhabung großer Mengen an Objekten gelingt nur sehr ineffizient.
Wie schon erwähnt, handelt es sich bei DiPOS um ein nicht orthogonales Persistenzmodell. Der Zustand des Objekts ändert sich für den Programmierer sichtbar. Diese Zustandsübergänge eines persistenten TPV-Objekts stellt Abbildung 4-7 dar.

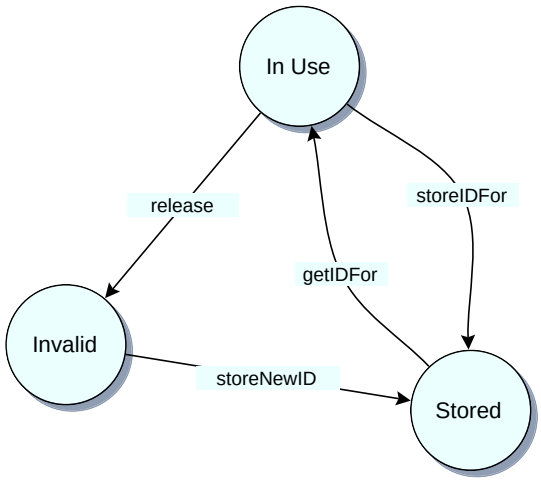


Abbildung 4-7: Zustandsübergänge eines persistenten TPV-Objekts

Ein neues TPV-Objekt wird durch Instanziieren eines Klassenobjekts erzeugt. Dabei besitzt es den Zustand Invalid, es muß noch seinen Objektidentifikator vom DiPOS zugewiesen bekommen. Nachdem es mittels storeNewID-Aufruf erstmals im DiPOS aufgenommen wurde, befindet es sich im Zustand Stored. Von dort kann es durch Aufruf der Methode getIDFor zur Erfüllung von Clientanfragen in den aktiven Zustand (In Use) kommen. Nach einer Veränderung wird ein TPV-Objekt mit der StoreIDFor-Methode des DiPOS wieder abgespeichert und befindet sich wieder im Zustand Stored. In den Zustand Invalid kann es nur noch gelangen, wenn es nicht mehr referenziert wird. Dann sendet es dem DiPOS eine release-Nachricht, wird als Invalid gekennzeichnet und anschließend gelöscht.

[1] Object Definition Language der ODMG

Zur Verwaltung der TPV-Objekte nutzt DiPOS drei unterschiedliche Verzeichnisse. Dabei enthält das nsObjectsDict alle notwendigen Informationen zum Auffinden des gewünschten Objekts, das nsObjectStore beinhaltet die Objekte selbst und das nsVerDict stellt ein Verzeichnis der verschiedenen Versionen eines Objekts dar. Das nsObjectsDict hat die im Abbildung 4-8 gezeigte Struktur.
Jedes TPV-Objekt besitzt seinen eigenen Objektidentifikator. Dieser ist als TPVid-Klasse implementiert und beinhaltet neben einer fortlaufenden Nummer die Versionsnummer und den Identifikator des Hosts, auf dem es erzeugt wurde. Die Kombination aller Teile ergibt einen netzwerkweit eindeutigen Identifikator. Um eine möglichst große Unabhängigkeit von Objekt und Objektverwaltung zu erreichen, ist es DiPOS nicht möglich, diesen Objektidentifikator direkt auszulesen. Es benutzt dazu ausschließlich Zugriffsmethoden der id-Klasse, die den Identifikator als Zeichenkette bereitstellen. Diese Zeichenkette dient bei der Client-Server-Kommunikation als Objektidentifikator und wird auch zur Objektverwaltung benutzt. Im nsObjectsDict stellen sie die Schlüssel zu den genauen Verwaltungsdaten eines Objekts dar. Diese bestehen aus dem eigentlichen Objektidentifikator, dem Rechnername, wo das Objekt gespeichert ist, dem augenblicklichen Zustand des Objekts und einem Zähler, der angibt, wie viele Clients gerade dieses Objekt nutzen.

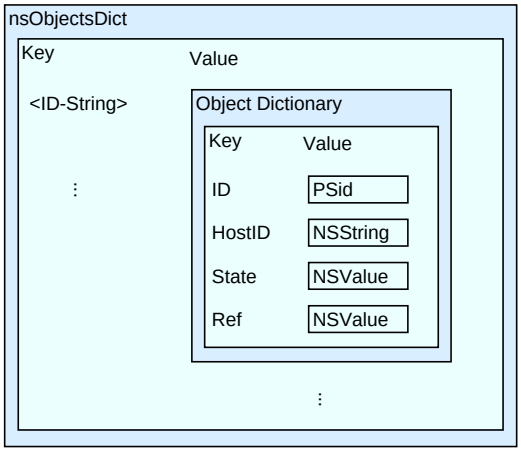


Abbildung 4-8: Objektverzeichnis des DiPOS

Ist der Objektidentifikator eines TPV-Objektes bekannt und befindet sich dieses auf dem lokalen Rechner, so kann man es direkt aus dem nsObjectStore genannten Verzeichnis erhalten. Abbildung 4-9 zeigt dessen Aufbau.
So ist es möglich, schnell ein beliebiges TPV-Objekt zu lokalisieren. Eine Ordnung der Objekte nach ihren Versionen ist nicht realisiert. Dies hat seine Ursache in der geringen Geschwindigkeit dieses Ansatzes. So wird man nur in sehr seltenen Fällen zuerst Zugriff auf alle Versionen eines Objekts benötigen, um anschließend eines daraus auszuwählen. Die vermutlich häufigste Zugriffsart ist der direkte Zugriff auf ein Objekt, dessen Referenz zum Beispiel in einem Modell vorkommt. Es mußte also in erster Linie ein möglichst schneller Zugriff auf einzelne Objekte realisiert werden.

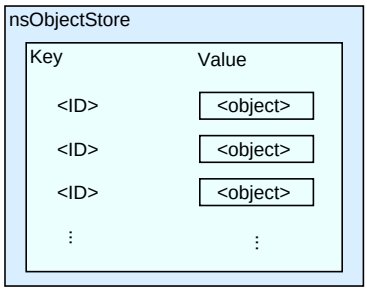


Abbildung 4-9: Struktur des Objektlagers

Um dennoch eine Ordnung der Objekte nach ihren Versionen zu ermöglichen, wurde das Versionsverzeichnis implementiert. Dabei wurde der Fakt ausgenutzt, daß alle Versionen des gleichen Objekts dieselbe fortlaufende Nummer besitzen, die das DiPOS der ersten Version zugewiesen hat. Somit enthalten die Objektidentifikatoren aller Versionen die gleiche Teilzeichenkette, in Abbildung 4-10 FamID genannt. Will man nun von einem Objekt alle anderen Versionen erfahren, so extrahiert DiPOS diese Zeichenkette aus dem Objektidentifikator und benutzt sie als Schlüssel für das Versionsverzeichnis. Die so erhaltene Liste beinhaltet die Objektidentifikatoren aller Versionen des Objekts, die nun direkt angesprochen werden können. Wird eine neue Version des Objekts gespeichert bzw. eine existierende Version freigegeben, so muß neben dem Objektverzeichnis auch das Versionsverzeichnis entsprechend aktualisiert werden.

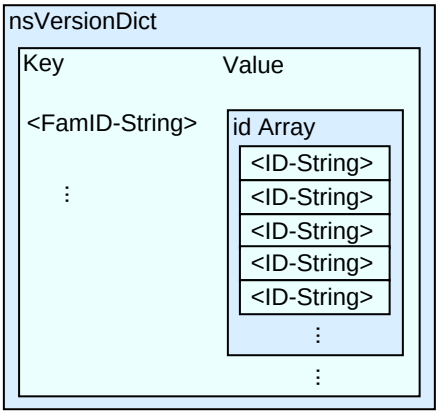


Abbildung 4-10: Aufbau des Versionsverzeichnisses

Da alle TPV-Server eine gemeinsame, verteilte Objektbasis nutzen, kann es vorkommen, daß das gewünschte Objekt nicht lokal gespeichert ist und von einem anderen TPV-Server bereitgestellt werden muß. Das Feld HostID im Objektverzeichnis enthält den Namen dieses Rechners.
Es muß zwischen den DiPOS-Komponenten jedes TPV-Servers eine Kommunikation stattfinden, die den Austausch von Objekten regelt. Dies wird durch die DiPOS-I/O- Komponente realisiert. Nach der Initialisierung der Objektverwaltung wird in einem separaten Thread ein TPVDiPOSio-Objekt erzeugt, das netzwerkweit Zugang zum den von DiPOS verwalteten Objekten bietet. Es stellt dabei eine im Vergleich zu DiPOS vereinfachte Schnittstelle bereit (Tafel 4-2).

getCopyFor: <TPV-Objekt>	liefert eine Kopie des gewünsschten TPV-Objekts
storeObject:<TPV-Objekt> with:<Objekt-ID>	speichert TPV-Objekt in entferntes DiPOS
noNeedFor:<Objekt-ID>	markiert TPV-Objekt als unbenutzt
releaseObject:<TPV-Objekt>	veranlaßt Löschen des TPV-Objekts

Tafel4-2: Schnittstellen des DiPOSIO-Objekts

Wird ein TPV-Objekt von einem entfernten Host benötigt, so baut das DiPOS eine Verbindung zum TPVDiPOSio-Objekt des entfernten TPV-Servers auf und sendet eine getCopyFor:-Nachricht. Daraufhin fordert das TPVDiPOSio-Objekt das gewünschte TPV-Objekt vom lokalen DiPOS an und sendet eine Kopie dessen zurück. Abbildung 4-11 stellt diese Kommunikation dar. Es kann eine Kopie des Objekts verwendet werden, da es innerhalb des DiPOS zu keinen Inkonsistenzen beim mehrfachen Zugriff auf das gleiche Objekt kommen kann. Wird ein TPV-Objekt modifiziert, so muß es mit einer höheren Versionsnummer neu eingespeichert werden, ein im System verwaltetes Objekt ist nicht veränderlich und kann lediglich freigegeben werden, wenn es nicht mehr referenziert wird. Wird ein TPV-Objekt von mehreren Clients gleichzeitig modifiziert, ist durch DiPOS lediglich sicherzustellen, daß beim Einspeichern unterschiedliche Versionsnummern vergeben werden.
Es ist ebenfalls möglich, Objekte auf entfernten TPV-Servern zu speichern. Damit ist die Möglichkeit zur Migration von Objekten gegeben, das heißt die verwalteten TPV-Objekte können für den Benutzer unsichtbar im Netzwerk verteilt werden. Dies ermöglicht zum einen eine Redundanz im Fehlerfall, kann aber ebenso zur Lastverteilung oder Systemwartung genutzt werden. Die hierbei erwünschten Effekte werden durch eine Objektverteilungsstrategie erreicht, die Regeln für die Migration von Objekten festlegt. In der Praxis ist diese stark vom Aufgabengebiet des Systems abhängig, eine universelle Strategie gibt es nicht. Aus diesem Grund wurde die Objektmigration in der vorliegenden Implementation nicht verwirklicht. Dies bleibt einer praxistauglicheren Realisierung der Objektverwaltung überlassen, die an eine konkrete Umgebung angepaßt werden muß

4.2.4 Erweiterbarkeit durch externe Bundle-Objekte

Wie bereits geschildert, schöpft ProSpecT seine Funktionalität zum größten Teil aus den Erweiterungen des TPV-Kerns. Dabei dient das MainTemplate-Objekt als Ausgangspunkt für spezialisierte Unterklassen.
Realisiert wurde diese Erweiterbarkeit durch zur Laufzeit zuladbare Objective C-Objekte. Die OpenStep-Klasse NSBundle stellt diese Funktionalität der Programmiersprache bereit. Im TPV-System übernimmt das Server-Objekt die Aufgabe der Erweiterbarkeit. Wird das TPV-Programm gestartet, so sucht es in einem festgelegten Verzeichnis nach NSBundle- Objekten. Diese werden daraufhin zum laufenden Programm hinzugeladen. Die darin enthaltenen Klassenobjekte können durch das Laufzeitsystem identifiziert werden, sind so dem TPV-Programm bekannt. Sie können fortan instanziiert und die damit bereitgestellte Funktionalität genutzt werden.

Die so erhaltenen Objekte besitzen noch keine Beziehung zu TPV-Objekten, obgleich sie deren erweiterte Funktionalität bereitstellen. Die Verknüpfung von nachgeladenem externen Bundle-Objekt und TPV-Objekt wird durch das nsTypeDict genannte Verzeichnis realisiert. Es wird nach Start des TPV-Kerns initialisiert und besitzt folgende Struktur:

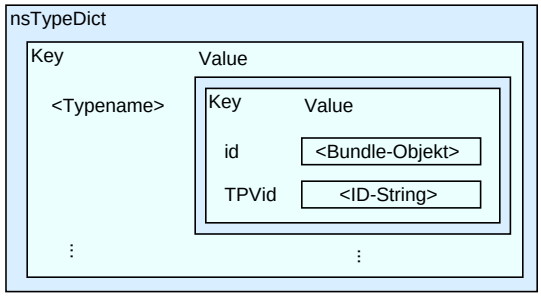


Abbildung 4-12: Aufbau des Typenverzeichnisses

Im Ausgangszustand (beim allerersten Start, es existiert nur das abstrakte TPVMainTemplate -Objekt) werden zunächst Unterklassenobjekte des TPVMainTemplates gebildet. Dazu wird es geklont und das Duplikat dem externen Bundle-Objekt übergeben. Dieses füllt entsprechend seiner Funktionalität den Methodenraum und gegebenenfalls den Attributraum. Der Methodenraum wird dabei um nsMethodDict-Objekte (siehe Abbildung 4-6) erweitert, die eine Beschreibung der neuen Methoden enthalten. Diese enthalten dafür die API genannte Schnittstellenbeschreibung zum Clientprogramm, eine ausführlichere, verbale Beschreibung der Methode sowie die zu ihrer Ausführung notwendigen Benutzerrechte. Anschließend vergibt das externe Bundle-Objekt einen neuen Typnamen für das neue TPV-Klassenobjekt. Das Klassenobjekt wird im DiPOS abgelegt und die neue Unterklasse im TPVMainTemplate-Objekt vermerkt.Abschließend wird im Typenverzeichnis ein neuer Eintrag eingefügt, der Verweise auf das zum neuen Typ zugehörige externe Bundle sowie das TPV-Klassenobjekt enthält.
Wird nun ein TPV-Instanzenobjekt durch den Benutzer erzeugt, ist über dessen Typname sowohl das TPV-Klassenobjekt, als auch das die Funktionalität bereitstellende externe Bundle-Objekt erreichbar.Bei jedem anschließenden Neustart des TPV-Systems sind alle Klassenobjekte bereits im DiPOS vorhanden, ihre Typen sind im ClassDict des MainTemplate-Objekts enthalten und können so in das neu zu aufzubauende Typverzeichnis aufgenommen werden. Die zugehörigen externen Bundle-Objekte werden durch das Server-Objekt zum System hinzugeladen und vervollständigen das Typenverzeichnis. Das TPV-System ist arbeitsfähig.

4.2.5 Objektnachrichtensystem

Aufgabe des Objektnachrichtensystems ist die Vermittlung zwischen den Clientanfragen an das System und den tatsächlich vorhandenen Objekten. Dem Benutzer steht dazu eine funktionale Schnittstelle zur Verfügung, die eine Abstraktion der implementierten Objective C-Methoden aller TPV-Bestandteile darstellt. Diese besteht aus drei Teilen:

dem Zielobjektidentifikator,
dem Funktions-/Methodenname,
dem zu übergebenden Argumentvektor.

Als Ergebnis dieser funktionalen Anfrage gibt das TPV-System einen Ergebnisvektor zurück, der die entstandnen Resultate enthält. Um eine solch abstrakte Schnittstelle realisieren zu können, muß der TPV-Server alle erhaltenen Werte interpretieren können. Auf Clientseite ist eine Aufbereitung der Resultate notwendig. Im TPV-System übernimmt das TPVAmbassador-Objekt diese Vermittlungsrolle. Diese besteht im einzelnen aus:

- Entgegennahme der Clientanfrage,
- Bereitstellung des Zielobjekts der Clientanfrage (Funktion),
- Übertragen des formalen Funktionsnamens auf den zugehörigen Objektmethodenaufruf,
- Prüfen der Zugriffsrechte des Clients,
- Aufbereitung der Argumente,
- Ausführen der Funktion, wenn möglich,
- Aufbereiten der Ergebnisse,
- Übergabe des Ergebnisvektors an den Client.

Der Nachrichtenaustausch zwischen TPVAmbassador-Objekt und Clientprogramm wird durch verteilte OpenStep-Objekte realisiert. Es kommt dabei das in Abschnitt 4.2.1 (siehe Seite 19ff) beschriebene Protokoll zur Anwendung.
Einen Teil der Clientanfrage stellt eine Zeichenkette dar, die das Objekt identifiziert. Dies ist die gleiche Zeichenkette, die auch vom DiPOS zur Objektverwaltung verwendet wird. Um das Zielobjekt einer Clientanfrage zu ermitteln, kann es einfach vom DiPOS angefordert werden. Dabei gibt es zwei mögliche Ausnahmen. Zum einen kann neben einem TPV-Objekt auch das TPV-System selbst vom Client angesprochen werden. Tafel 4-3 zeigt alle Systemanfragen. Dazu wird anstelle einer Objektidentifikation die Zeichenkette „TPV“ als Adressat der Anfrage benutzt.

addref	teilt Objekt neuen Verweis mit
api	gibt API der Methoden aus
created	liefert das Erzeugungsdatum
creator	liefert den Urheber des Objekts
describe	verbale Beschreibung der API
help	liefert Objektbeschreibung
instance	erzeugt Instanz von Klassenobjekt
migrate	migriert Objekt zu anderem Host
models	zeigt alle Modelle, die die Objektklasse
name	liefert den Objektname
objects	zeigt alle im DiPOS enthaltenen Objekte
shutdown	beendet TPV-Server
subref	teilt Objekt gelöschten Verweis mit
rights	zeigt notwendige Benutzerrechte an
type	liefert Klasse des Objekts
versions	zeigt alle Versionen des Objekts

Tafel 4-3: Clientanfragen an das TPV-System

Andererseits kann der Typ des adressierten Objekts eingeschränkt werden, das heißt man beschränkt die sichtbare Funktionalität des Objekts auf die einer seiner Oberklassen. In der Terminologie der Softwaretechnik spricht man auch von „type casting“. Dazu ist der gewünschte Typ am Ende der Identifikatorzeichenkette, durch ein „@“ getrennt, anzugeben. Es muß zur Erfüllung einer solchen Anfrage also nicht das Klassenobjekt des adressierten Instanzenobjekts angesprochen werden, sondern das zum angegeben Typ gehörende Klassenobjekt. Ist dies in der Vererbungshierarchie nicht eine Oberklasse des eigentlichen Objekttyps, tritt ein Typfehler auf, die Anfrage kann nicht bearbeitet werden.
Ist das Zielobjekt der Anfrage gefunden, muß die gewünschte Funktion durch einen Methodenaufruf ersetzt werden. Auch hierbei ist zwischen zwei möglichen Fällen zu unterscheiden. Jedes TPV-Objekt erbt durch das TPVMainTemplate die Funktionalität mehrerer einfacher Methoden. Diese können vom Objekt selbst erfüllt werden, alle weiteren Methoden werden durch externe Bundle-Objekte bereitgestellt. Handelt es sich bei der Clientanfrage um eine „einfache“ Methode, kann das TPVAmbassador-Objekt das Zielobjekt direkt ansprechen. Ansonsten muß es mit Hilfe des Typenverzeichnisses das zugehörige externe Bundle-Objekt ermitteln und die Anfrage an dieses weiterleiten.
Bevor jedoch ein Methodenaufruf stattfindet, prüft das TPVAmbassador-Objekt die Berechtigung des Clients, diese auch ausführen zu dürfen. Durch die Identifikation des Clients bei Verbindungsaufnahme sind die Benutzerrechte des Clients bekannt, sie müssen den notwendigen Rechten zur Ausführung der gewünschten Methode entsprechen. Die Zugriffsrechte auf Methoden für erweiterte Methoden sind im MethodDict-Verzeichnis des Methodenverzeichnisses (Abbildung 4-6) enthalten, Zugriff auf die Basisfunktionalität der TPV-Objekte regelt das TPVAmbassador-Objekt.
Besitzt der Client die notwendigen Rechte, wird der Methodenaufruf durchgeführt. Bei erweiterten Methoden wird dazu ein NSInvocation-Objekt des OpenStep-Standards genutzt. Es enthält sowohl den Selektor der gewünschten Methode, als auch einen Vektor für Argumente und Rückgabewerte. Vor Aufruf der Methode fügt das TPVAmbassador-Objekt das TPV-Objekt als ersten Parameter in das NSInvocation-Objekt ein, um dem externen Bundle-Objekt Zugriff auf dessen privaten Attribute zu geben. Daran schließen sich alle anderen vom Client erhaltenen Argumente an.
Das externe Bundle-Objekt erfüllt anschließend die im NSInvocation-Objekt verpackte Methode. Dabei hat es die Prüfung der Argumente auf Korrektheit selbst zu übernehmen, da alle erweiterten Methoden für das Ambassador-Objekt anonym sind. Dieser überprüft auch nicht die zurückerhaltenen Ergebnisse, sondern schickt sie, gegebenenfalls neu geordnet, zurück an den Client.

4.3 Client

Das realisierte Clientprogramm stellt eine einfache, alphanumerische Schnittstelle zum TPV-Server dar. Dabei kann es prinzipiell sowohl im interaktiven Modus als auch im Batch-Modus[1] arbeiten.
Als erstes Argument beim Programmaufruf wird der Hostname des TPV-Servers erwartet. Wird kein entsprechender Schalter (-i) angegeben, interpretiert das Clientprogramm alle folgenden Parameter als Dateinamen, versucht diese einzuladen und den Inhalt als Anfragen zum TPV-Server zu senden.
Wie in Abschnitt 4.2.1 (siehe Seite 19ff) bereits ausgeführt, besteht das Clienprogramm aus zwei Komponenten, welche die Kommunikation zum TPV-Server realisieren. Der ClientPrompter wird durch das gleichnamige Objekt dargestellt, die ClientUI-Komponente ist durch das PSInterpreter-Objekt realisiert.

[1] Stapelverabeitungsmodus

Nach Start des Clientprogramms wird das PSClientPrompter-Objekt in einem separaten Thread gestartet. Ist dies erfolgreich geschehen, nimmt das Clientprogramm mit dem TPV-Server des angegebenen Hosts Kontakt auf. Dabei wird das in Abschnitt 4.2.1 (siehe Seite 19ff) beschriebene Verbindungsprotokoll verwendet. Kommt eine Verbindung zustande, wird ein PSInterpreter-Objekt erzeugt und diesem die Kommunikation mit dem TPV-Server übertragen.
Im interaktiven Modus erwartet dieser Benutzereingaben über die Tastatur. Wurde eine Textzeile eingegeben, zerlegt das PSInterpreter-Objekt die Zeichenkette und setzt aus ihren Bestandteilen eine TPV-Anfrage zusammen. Dabei erfolgt keinerlei Überprüfung der Eingaben auf Syntax und Semantik. Anschließend setzt das PSInterpreter-Objekt die Anfrage an den TPV-Server ab, nimmt den Ergebnisvektor entgegen und gibt ihn aus. Tritt bei der Anfrage eine Ausnahme auf, so gibt das PSClientPromter-Objekt an gleicher Stelle eine Mitteilung aus und wartet gegebenenfalls auf Benutzereingaben.
Der Batch-Modus besitzt nur eine untergeordnete Bedeutung. Das Konzept von ProSpecT geht von einer Interaktion mit dem Benutzer aus, eine automatische Abarbeitung von Skripten ist nur eingeschränkt und zu Debugging-Zwecken gedacht. Deshalb ist eine Reaktion auf eine Ausnahme im Batch-Modus nicht möglich, das Programm blockiert.
Davon abgesehen erfolgt die Verarbeitung der Skripte ähnlich wie im interaktiven Modus. Eine Zeile des Skripts wird dabei als Anweisung interpretiert, an den TPV-Server geschickt und das Ergebnis ausgegeben.
Damit steht für Testzwecke ein einfaches Clientprogramm zur Verfügung, das alle Möglichkeiten des TPV-Servers ausschöpfen kann und prototypischen Anforderungen genügt.

4.4 Ergebnisse und Probleme

Die hier beschriebene Implementation des vorgestellten ProSpecT-Konzepts stellt im gegenwärtigen Status lediglich die prototypische Basis für weitere Untersuchungen dar. Wie angedeutet sind Grundbestandteile wie das DiPOS-System nicht praxistauglich. Deshalb können an dieser Stelle auch noch keine abschließenden Ergebnisse über die Funktionalität, Stabilität und Leistungsfähigkeit des ProSpecT-Systems vorgestellt werden. Da die geforderte Funktionalität von ProSpecT durch dessen Erweiterungskomponenten erreicht wird, ist es nur sehr schwer, die Funktionalität des TPV-Basissystems zu überprüfen und beurteilen.
Durch die realisierten Komponenten konnte bis jetzt lediglich die Funktionsfähigkeit der Clientverwaltung und des Clientprogramms selbst getestet werden. Dabei wurde aus Gründen der Programmstabilität eine kompliziertere Verwaltung der TPVMessageLoop-Objekte notwendig. Die Ursache dafür stellte eine unzuverlässige Funktionalität des NSRunloop-Objekts in der Solaris-Implementation des OpenStep-Standards dar.

5 Zusammenfassung

In der vorliegenden Arbeit wurde ein bislang nicht benutzter integrativer Ansatz verwendet, um ein Informations- und Ausführungssystem für den durchgängigen Halbleiterprozeß zu entwerfen. Dabei konnte gezeigt werden, daß der gewählte Ansatz bisherigen Lösungen konzeptuell überlegen sein kann.

Weiterhin wurde deutlich, daß das Konzept des Modellierungssystems ProSpecT aufgrund seiner minimalistischen Struktur ohne Probleme auch auf Aufgabengebiete jenseits der Mikroelektronik angewendet werden kann und auch dort seine konzeptuellen Vorteile voll zum Tragen kommen können. Mit ProSpecT steht damit ein Konzept zur Verfügung, das für viele Bereiche einen integrativen Lösungsansatz realisierbar erscheinen läßt.
Mit TPV wurde versucht, das erarbeitete Konzept auf bestehender Rechentechnik zu implementieren. Obwohl aus Zeitmangel die Funktionsfähigkeit der Implementation nicht abschließend demonstriert werden kann, ist dennoch ein kompaktes und leistungsfähiges Programmpaket entstanden, das als Basis für weitergehende Untersuchungen genutzt werden kann. Die dazu nötige Grundfunktionalität wird bereits durch das bis jetzt realisierte TPV-System bereitgestellt.

6 Literaturverzeichnis

[ADA-80] Adams, Douglas: *The Restaurant at the End of the Universe*
Ballantine Books New York 1980
ISBN 0-345-39181-0

[DV-96] Alan Dearle, Francis Vaughan:
Grasshopper: An orthogonally persistent operating system
Department of Computer Science University of Adelaide
WWW: http://www.gh.cs.su.oz.au/Grasshopper/Papers/GH10/gh10.html

[FRA-91] Frank, Martin: *Modellierung und Simulation, 1.Lehrbrief*
TU-Dresden, Faklutät Informatik 1991

[GPV$^+$-90] Gyssens, Marc, Paredaens, Jan, Van den Bussche, Jan, Van Gucht, Dirk:
A Graph-Oriented Object Database Model
9th ACM Symposium on Principles of Database Systems 1990

[GS-96] Gratz, Achim, Spallek, Rainer G. :
Benutzeroberfläche und Datenverwaltung für Systeme gekoppelter Simulatoren
ASIM ’96 - 10.Symposium Simulationstechnik 16.9.-19.9.96
TU-Dresden

[HAI-95] Haines, Jason: *Persistent Object Store Applications*
Department of Computer Science, Australian National University
WWW: http://demos.anu.edu.au/jason/thesis/thesis/build/thesis.ps

[JON-96] Jones,Richard:
Garbage Collection-Algorithms for Automatic Dynamic Memory Management,
John Wiley & Sons, Ltd, ISBN 0-471-94148-4

[LKK-93] Lockemann, Peter C.,Krüger, - Gerhard, und Krumm, Heiko:
Telekommunikation und Datenhaltung,
Carl Hanser Verlag, Studienbücher der Informatik, München
Wien 1993

[MV-96] Cheri McKeown, Umesh Vaishampayan:
Mapping OMG IDL to Objective C
30 October 1996 - Preliminary
WWW: http://www.next.com:80/OPENSTEP/WhitePapers/
Mapping_OMG_IDL_To_Objective-C.htmld/

[NEX-94] NeXT Computer Inc, SunSoft Inc.:
OpenStep Specifikation 1.0
Redwood 1994
FTP: ftp.next.com/pub/OpenStepSpec/OpenStepSpec.ps.Z

[OMG-95] ObjectManagement Group:
Corba 2.0 Request for Proposal - Portability Enhancement
FTP: ftp.omg.org/pub/docs/1995/95-06-26.ps

[TAN-92] Tanenbaum, Andrew S.: *Modern Operating Systems*
Prentice-Hall International, Inc ISBN 0-13-595752-4

Index