Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

Diplomarbeit

Dynamisches Rebinden in kooperierenden persistenten Objektsystemen

19. Februar 1998

Kay Ramme
Buchsbaumweg 16
22299 Hamburg

Tel.: 040 / 511 59 61

Betreut von:
Prof. Dr. Joachim W. Schmidt
Dr. Martin Lehmann

Zusammenfassung

Diese Arbeit beschäftigt sich mit dynamischer Rebindung, kooperierenden Objektsystemen, sowie der kooperativen Softwareverwaltung und -entwicklung.

Dynamisches Rebinden oder Neubinden bezeichnet das Ersetzen der Bindung eines Objektes **A** an ein anderes Objekt **B** durch eine neue Bindung von **A** an ein drittes Objekt **C**.

Typische Anwendungen für dynamisches Rebinden sind Aktualisierungen in langlebigen persistenten Programmen, die Wiederherstellung von Beziehungen zwischen temporären externen und persistenten internen Objekten, sowie die Transformation immobiler Objektgraphen in mobile Objektgraphen.

Beispiele:

- Ein Programm verwendet die Version 2.1 einer Funktion **F**, d.h. das Programm ist an ein Objekt gebunden, welches die Funktion **F** in übersetzter Form beinhaltet. Die Funktion **F** wird während der Laufzeit des Programmes weiterentwickelt zur Version 2.2. Dynamisches Rebinden erlaubt es nun, die Bindung des Programmes an das Objekt mit der übersetzten Funktion **F** in der Version 2.1 durch eine Bindung an ein Objekt mit der übersetzen Funktion **F** in der Version 2.2 zu ersetzen. Zukünftige Aufrufe der Funktion **F** durch das Programm verwenden die Version 2.2 der Funktion **F**.

- Ein persistentes Programm referenziert ein temporäres Betriebssystemobjekt (z.B. ein Datei-Handle). Durch einen Abbruch des Programmes (beispielsweise in Folge eines Stromausfalls) und einer Beendigung des zugehörigen Betriebssystemprozesses wird das Betriebssystemobjekt freigegeben (die zum Datei-Handle gehörige Datei geschlossen). Vor der Wiederaufnahme des Programmlaufs, müssen die Bindungen des Programms an nun nicht mehr verfügbare Betriebssystemobjekte durch Bindungen an neu allozierte Betriebssystemobjekte ersetzt werden.

- Ein Thread mit immobilen Objekten in seiner transitiven Hülle kann, infolge der Immobilität dieser Objekte, nicht ohne Verlust dieser Objekte von einem Objektspeicher in einen anderen migrieren (es sei denn, es gibt Netzwerkreferenzen). Werden die Bindungen des Threads an immobile Objekte im Ursprungsobjektspeicher durch Bindungen an äquivalente Objekte im Zielobjektspeicher ersetzt (z.B. durch Repräsentanten), so kann der Thread dochnoch migrieren.

Danksagungen

Allen, die zum gelingen dieser Arbeit beigetragen haben, möchte ich an dieser Stelle danken. Insbesondere danke ich Gerald Schröder, für seine unermüdliche konstruktive Kritik sowie für die sehr gute Betreuung. Herrn Prof. Dr. Joachim W. Schmidt danke ich für die guten Arbeitsbedingungen und dafür, daß er mir die Möglichkeit gegeben hat, diese Arbeit zu schreiben. Herrn Dr. Martin Lehmann danke ich für wertvolle Ratschläge und Anmerkungen sowie dafür, die Möglichkeit gehabt zu haben, die Arbeit zu schreiben.

Für Claudia

Inhaltsverzeichnis

Abbildungsverzeichnis

Beispiele

Kapitel 1

Einleitung und Motivation

Wurden Computersysteme früher meist als Einzelsysteme (engl. standalone systems) verwendet, so zeigt der Trend der letzten Jahre jedoch, daß Computersysteme zunehmend vernetzt werden (vgl. [KBM+89]). Besondere Aufmerksamkeit ist in diesem Zumsammenhang dem Internet zu schenken, dessen Entwicklung, insbesondere in der letzten Zeit, enorme Fortschritte gemacht hat. Computersysteme werden vernetzt um miteinander kooperieren zu können. Typischer weise stellt ein Computersystem im Netzwerk Dienste und Daten anderen Computersystemem zur Verfügung oder nimmt Dienste und Daten anderer Computer in anspruch.

Klassische Kooperationsdienste sind verteilte Dateisysteme (z.B. NFS[1]), entfernter Prozeduraufruf, Druckdienste, Dateitransfer oder WWW-Dienste [LPJ+94]. Folgende innovative, anspruchsvolle Anwendungsszenarien sind für kooperierende Computersysteme denkbar [Mat96]:

- Geschäftsvorgänge (engl. workflows) bewegen sich im firmeneigenen Netzwerk von Computer zu Computer und werden von den jeweiligen Mitarbeitern weiter bearbeitet [Mat96, WMF].
- Softwareagenten bewegen sich im Netzwerk und sammeln dabei Daten oder erledigen vom Benutzer vorgegebene Aufgaben [Joh97].
- Die Softwarearbeitsumgebung eines Benutzers begleitet diesen durch das Netzwerk an seinen jeweiligen Arbeitsplatz.
- Da ein einzelnes Computersystem eine geringere Ausfallsicherheit als ein Netzwerk bietet, werden langlebige Prozesse dupliziert bzw. migriert, insbesondere bei Abzeichnen eines Systemversagens (siehe auch [Kur97]).

[1] Network File System

- CSCW[2] Anwendungen, wie z.B. die Entwicklung von Softwaresystemen, werden erst duch Software möglich, die die Kooperation von Computersystemen unterstützt.

- Computersysteme sind vollständig von anderen Computersystemen administrierbar.

Die ersten vier Anwendungsszenarien machen es erforderlich, daß in Ausführung befindliche Programme, meist als Prozesse oder Threads bezeichnet, von einem Computersystem in ein anderes migrieren können. Die Komplexität dieser Materie verlangt, daß schon auf Systemebene Mechanismen zur Unterstützung von Mobilität zur Verfügung stehen, die es dem Anwendungsprogrammierer erlauben Software zur Unterstützung von Kooperation zu entwickeln, möglichst ohne sich mit den Problemen migrierender Prozesse auseinander setzen zu müssen.

Als Folge einer Vernetzung von heterogenen Computersystemen wurden eine Menge von plattformunabhängigen Programmiersprachen entwickelt, deren Übersetzer auch plattformunabhängigen ausführbaren Code generieren. Basis für die Ausführung plattformunabhängigen Codes sind sogenannte virtuelle Maschinen. Eine virtuelle Maschine ist ein Interpreter, der, im Prinzip wie die Recheneinheit eines Computersystems, die Anweisungen des Codes dynamisch umsetzt. Da eine virtuelle Maschine ein virtuelles Computersystem darstellt, ist es in bezug auf Durchsatz und Auslastung sinnvoll, Basisfunktionalität für Nebenläufigkeit zu implementieren (vgl. [JV86]). Programmablaufkontexte werden infolgedessen explizit modelliert und können im Speicher der virtuellen Maschine als Objekte repräsentiert werden. Diese explizite Speicherung der Ablaufkontexte erlaubt es, Prozesse migrieren zu lassen.

Eine Folge des Wunsches hoher Verfügbarkeit von Computersystemen sind langlebige Systeme. Um die Verfügbarkeit eines Computersystems zu gewährleisten, ist es notwendig, dieses im Betrieb zu verändern, damit es z.B. an veränderte Anforderungen (Adaption) angepaßt werden kann, oder damit etwaige Fehler beseitigt werden können. Voraussetzung für eine solche Evolution eines Systems (siehe [KCMS96]) sind die dynamische Erzeugung von Programmen, dynamisches Übersetzen und dynamisches Rebinden sowie die Möglichkeit, reflektiv auf die Systemkomponenten zugreifen zu können.

Um nicht in allen Objektsystemen alle möglichen Softwarekomponenten in allen Versionen vorhalten zu müssen und um migrierende Objektgraphen reduzieren zu können sowie um Bindungen für Aktualisierungen identifizieren zu können, ist es notwendig, die Verwaltung der Softwarekomponenten in das System zu integrieren (siehe Abschnitt 4.1).

Große Softwaresysteme werden von Gruppen von Programmierern entwickelt, wo-

[2]Cooperative Computer Supported Work

bei die Arbeit der Programmierer meist durch Computernetzwerke unterstützt wird. Die meisten Softwareverwaltungswerkzeuge (z.B. Teamware, TCCS[3]) sind dateibasiert. Dies hat zur Folge, daß ein verteiltes Dateisystem (z.B. NFS) benötigt wird. Dateisystembasierte Softwareverwaltungswerkzeuge können meist nicht vollständig vom zugrundeliegenden Dateisystem abstrahieren; ein vollständig transparentes Arbeiten mit Softwarekomponenten ist infolgedessen nicht möglich.

Netzwerke verursachen hohe Kosten für die Administration. Insbesondere Daten und Programme, die nicht zentral verwaltet werden können, erfordern es, daß sich der Systemadministrator individuell mit ihnen beschäftigt. Computersysteme, deren sämtliche Komponenten zentral verwaltet werden und die es erlauben, diese Komponenten dynamisch auszutauschen (Aktualisierung, Evolution / Reflektion), reduzieren die Kosten für die Administration enorm, da z.B. Aktualisierungen automatisiert werden können.

Klassische Verfahren für die Entwicklung datenintensiver Applikationen trennen zwischen Programm und Datenverwaltung. Programme und Schemata für die zugehörigen Datenbanken werden getrennt beschrieben, Typsysteme und Datenrepräsention sind meist getrennt entwickelt und nur teilweise miteinander vereinbar (engl. impedance mismatch; siehe [STS97]).

Systeme, die Programm- und Datenverwaltung integrieren, verwenden für Programme und Daten dasselbe Typsystem und dieselbe Repräsentation. Mit dem Begriff Persistenz wird die Langlebigkeit von Daten bezeichnet. Integrierte Systeme, die für die Datenspeicherung einen Objektspeicher verwenden, werden als Objektsysteme bezeichnet. Systeme dieser Art erlauben referentielle Beziehungen zwischen den Objekten des Objektspeichers und verwenden einen Speicherrückgewinnungsmechanismus, um nicht mehr erreichbare Objekte freizugeben. Persistenz wird in diesen Systemen meist orthogonal hergestellt. Objekte, die für den Programmierer nicht mehr sichtbar bzw. erreichbar sind, werden nicht gespeichert.

Objektsysteme bestehen aus einer Vielzahl von Komponenten; als wichtigste sind folgende erkennbar (siehe Abb. 1.1):

Objektspeicher: Basis eines Objektsystems ist der Objektspeicher. Objektspeicher dienen prinzipiell zur Speicherung aller Daten eines Objektsystems. Objektspeicher können auch die persistente Speicherung ihrer Objekte unterstützen, die Basisfunktionalität dieser Eigenschaft ist auch die Basis für migrierende Prozesse (vgl. [Mat96])

Virtuelle Maschine: Die virtuelle Maschine ist die den Programmcode ausführende Einheit. Programmcode wird in Objekten im Objektspeicher ge-

[3]Trivial Configuration Control System

speichert. Die virtuelle Maschine sollte die gleichzeitige Ausführung mehrerer nebenläufiger Prozesse (engl. multitasking bzw. multithreading) unterstützen.

Spezifische Anbindungen (externer Dienste): Die Funktionalität der virtuellen Maschine sollte dynamisch um systemspezifische Funktionalität erweitert werden können, da nicht alle möglichen funktionalen Eigenschaften von Computersystemen vorhersehbar sind und gleiche Funktionalität unterschiedlicher Computersysteme unterschiedliche Anbindungen erfordert. Die Integration externer Funktionalität (z.B. Kommunikationsdienste, Dateidienste) erfolgt meist mit dynamisch bindbaren Softwarebibliotheken.

Softwarekomponenten: Die Funktionalität eines Objektsystems wird in den zugehörigen Softwarekomponenten implementiert. Manche Systeme integrieren auch den Übersetzer für Softwarekomponenten als Softwarekomponente (Bootstrapping). Ist der Übersetzer als Softwarekomponente in das Objektsystem integriert, so kann reflektiv auf seine Komponenten zugegriffen werden.

Übersetzer: Der Übersetzer erzeugt aus einer Programmbeschreibung ein, für die virtuelle Maschine, ausführbares Objekt.

Prinzipiell kann jede dieser Komponenten die Funktionalität einer der anderen Komponenten nutzen.

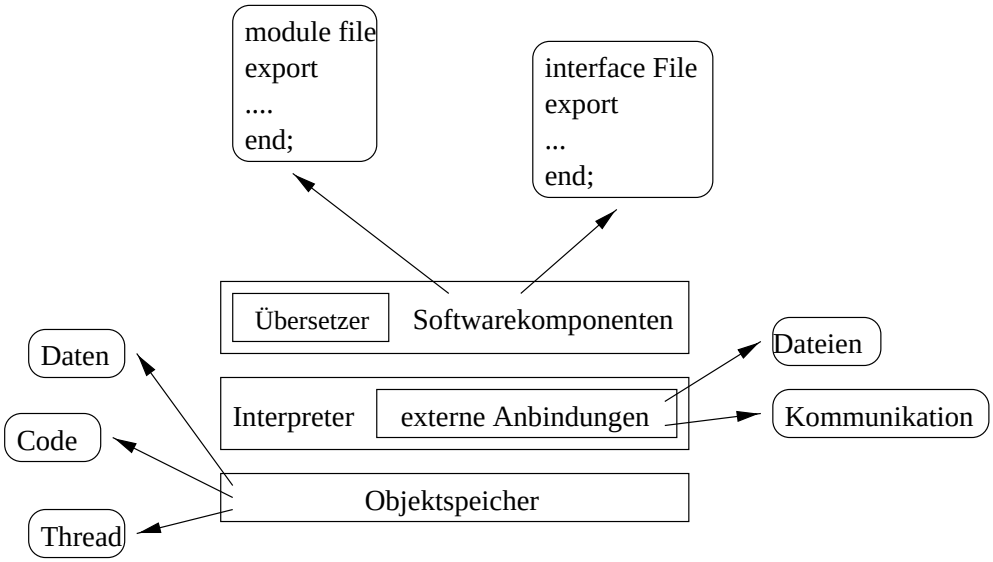

Abbildung 1.1: Komponenten eines Objektsystems

Folgende Punkte sind Anwendungsgebiete für kooperative Objektsysteme:

- Migrierende Threads erlauben es, oben genannte Aufgaben (migrierende Geschäftsvorgänge, Agenten, migrierende Arbeitsumgebungen) zu realisieren.

- Reflektive Objektsysteme können veränderten Systemanforderungen dynamisch gerecht werden (Adaption, Evolution, Administration).

- Kooperierende Objektsysteme erlauben es, die Softwarekomponentenverwaltung nahtlos zu integrieren.

Mit kooperierenden Objektsystemen sind also die anfänglich genannten Szenarien realisierbar.

Die Funktionalität kooperierender Objektsysteme geht über die von verteilten Dateisystemen angebotenen Möglichkeiten hinaus. Kooperative Werkzeuge für die Softwareverwaltung lassen sich transparent für den Benutzer implementieren. Ein Objektsystem kann beispielsweise übersetzte Module für andere Objektsysteme vorhalten. Möchte ein Benutzer nun in einem lokalen Objektsystem ein bestimmtes Modul verwenden, so wird das Modul über das Netzwerk in den lokalen Objektspeicher kopiert und in die Umgebung des Benutzers eingebunden. Z.B. lassen sich bei Kenntnis fehlerhafter Softwarekomponenten und der Verfügbarkeit einer korrigierten Version einer fehlerhaften Komponente automatisch jene Objektsysteme aktualisieren, in denen eine fehlerhafte Komponente verwendet wird.

Folgende Probleme treten bei der Migration von Objektgraphen, innerhalb kooperierender Objektsysteme, bzw. bei der persistenten Speicherung von Objektgraphen und bei langlebigen Systemen auf:

- Da migrierende Objektgraphen alle transitiv erreichbaren Objekte beinhalten, können sie sehr groß werden.

- Nicht alle Objekte eines Objektspeichers sind prinzipiell migrationsfähig. Infolgedessen sind auch Objektgraphen, mit derartigen Objekten in ihrer transitiven Hülle, nicht bzw. nur bedingt migrationsfähig.

- Ausgewählte Objekte dürfen oder sollen nicht migrieren.

- Objekte können nicht von einem Objektsystem in ein anderes verschoben, sondern nur kopiert werden.

- Objekte mit semantischen Bindungen an externe Objekte sind nicht speicher- und wieder restaurierbar (transiente Objekte).

- In evolutionären Systemen stellen ursprünglich als nicht variabel definierte Bindungen, die aktualisiert werden sollen, ein Problem dar.

Dynamische Rebindung ist die Basis für die Lösung dieser Probleme. Dynamisches Rebinden erlaubt es, Bindungen zu erneuern, die ursprünglich nicht als variabel definiert wurden. Reflektive Systeme, die einen Zugriff auf ihre Komponenten gestatten, können so zur Laufzeit verändert werden. Diese Evolution eines Objektsystems resultiert aus dynamischer Rebindung. Evolution ist die Folge adaptiver Systeme und vollständig zentral administrierbarer Systeme.

1.1 Gliederung und Aufgabenstellung

Die vorliegende Arbeit betrachtet die folgenden Bereiche:

Kooperierende Objektsysteme: Kooperierende Objektsysteme stellen Lösungen für die kooperative Arbeit von Computeranwendern zur Verfügung.

Dynamische Rebindung: Dynamische Rebindung von Objekten ist die Basistechnologie für die Lösung der genannten Probleme kooperierender, sowie persistenter Objektsysteme. Mit Hilfe dynamischer Rebindung lassen sich auch verteilte Objektsysteme entwickeln[4]. Evolution bzw. Adaption basieren auf der Technik des dynamischen Rebindens.

Integrierte Softwareverwaltung: Die Integration des Softwareverwaltungsprozesses in das Objektsystem ist notwendig für die dynamische Bereitstellung von Objekten für dynamisches Rebinden und für die Identifikation von Bindungen. Außerdem ist die Softwareverwaltung eine anspruchsvolle Beispielanwendung kooperativer Objektsysteme.

Zur Aufgabenstellung dieser Arbeit gehört es, die Techniken kooperierender Objektsysteme zu erarbeiten und weiterzuentwickeln. Möglichkeiten der dynamischen Rebindung sollen untersucht und analysiert werden. Die Softwareverwaltung soll in das Basisobjektsystem integriert werden. Ein Ziel dieser Arbeit ist die Entwicklung einer Beispielanwendung genannter Konzepte und Techniken. Im Gegensatz zu [Mat96] wird ein generischer Lösungsansatz für die Probleme ubiquitärer, immobiler bzw. nicht persistenter oder zu aktualisierender Objekte (z.B. Schemaevolution) aufgezeigt. Die Integration der Softwareverwaltung wird mit orthogonaler Kombination von Versionierung, Projektverwaltung und verteilter Datenhaltung realisiert.

Die Arbeit gliedert sich in folgende Kapitel:

Dynamisches Rebinden von Objekten: Dieses Kapitel gibt eine Einführung in Objektspeicher. Varianten dynamischer Rebindung werden analysiert

[4]Ein Objektsystem, dessen Objektspeicher es erlaubt, Objekte im Netzwerk transparent zu verteilen, ist, im Gegensatz zu einem kooperierenden Objektsystem, ein verteiltes Objektsystem. Verteilung und Kooperation können auch gemischt auftreten.

und beschrieben. Es werden die variable Bindung sowie die dynamische Rebindung miteinander verglichen. Ferner wird kurz auf Typisierungsaspekte dynamischer Rebindung eingegangen. Objektmigration und Persistenz werden diskutiert sowie die Probleme dieser Techniken aufgezeigt. Die Möglichkeiten zur Identifizierung von Bindungen werden betrachtet.

Kooperierende Objektsysteme: Dieses Kapitel beschreibt, wie Objektsysteme miteinander kooperieren können. Es wird gezeigt, wie Objektgraphen von einem Objektspeicher in einen anderen migrieren und wie dynamische Rebindung hier eingesetzt wird. Es wird eine Modul zur generischen Klient-/Server-Programmierung vorgestellt und die Netzwerkprogrammierung unter TYCOON beschrieben.

Rebindungsobjekte: Objekte, die für dynamische Rebindung geeignet sind, können nach unterschiedlichen Kategorien geordnet werden. Dieses Kapitel gibt einen Überblick über mögliche Kategorien, wobei anschließend auf jede Kategorie detailliert eingegangen wird. Konzepte der Projektverwaltung und der Versionierung werden erörtert. Möglichkeiten, mit immobilen und ubiquitären Objekten umzugehen, werden aufgezeigt.

Dynamische Rebindung in kooperierenden Objektsystemen: Dieses Kapitel faßt die Kapitel zwei, drei und vier zusammen und entwickelt ein Gesamtkonzept. Es wird die Implementation einer Beispielanwendung beschrieben und anhand von Beispielen ihre Verwendung aufgezeigt. Es wird gezeigt, wie mit Hilfe von Reflektion (vgl. [KCMS96]) Programmobjekte dynamisch abgeleitet werden können.

Erfahrungen und Ausblick: Die Erfahrungen, die während dieser Arbeit gesammelt wurden, werden zusammen gefaßt. Die Arbeit wird in einen umgebenden Kontext eingeordnet. Mögliche Ergänzungen, die die Basis für Folgearbeiten sein können, werden besprochen.

Anhang: Im Anhang finden sich ausgesuchte Schnittstellen der Module des praktischen Teils dieser Arbeit sowie einige Demos, um die Benutzung der erstellten Module und der Beispielanwendung zu erklären.

1.2 Basisarchitektur

Basis dieser Arbeit ist das TYCOON[5] System, das am Arbeitsbereich DBIS des Fachbereichs Informatik der Universität Hamburg entwickelt wurde. Durchgeführt wurde diese Arbeit im Rahmen des CoPOS[6] Projektes.

[5] Typed communicating objects in open environments
[6] Cooperating Persistent Object Systems (siehe [Sch97])

TYCOON ist ein integriertes Objektsystem für die Entwicklung und Anwendung datenintensiver Applikationen. Abbildung 1.2 zeigt den schichtweisen Aufbau von TYCOON. Die Komponenten eines Objektsystems (siehe Abb. 1.1) sind zu erkennen. Auf unterster Ebene befindet sich der Objektspeicher. Im TYCOON System ist für den Objektzugriff ein, vom Objektspeicher abstrahierendes, spezielles Protokoll (TSP[7]) definiert. Auf den Objektspeicher setzt die virtuelle Maschine (TVM[8]) auf. Die virtuelle Maschine ist die Basis für den TL- bzw. TLMIN-Übersetzer, bestehend aus TML, TYCOON Sprachebene sowie TYCOON Arbeitsumgebung. Der Übersetzer ist selbst in TL[9] bzw. TLMIN[10] implementiert, was es prinzipiell erlaubt, reflektiv auf die Komponenten des Übersetzers zuzugreifen. Spezifische Anbindungen werden im TYCOON System durch externe, in der Programmiersprache C entwickelte Funktionen gekapselt. Diese Funktionen sind nahtlos in TYCOON integrierbar.

Verschiedene Objektspeicher können mittels Adapter an das TYCOON System angeschlossen werden. Es existieren Adapter für NAPIER und ObjectStore. Die Objektspeicher TyMem und TySin, die am Arbeitsbereich DBIS entwickelt wurden, erlauben die Verwendung von Hauptspeicher oder Dateien zur Objektspeicherung. Objektspeicher für das TYCOON System müssen persistent sein.

1.3 Tycoon-Terminologie

Folgende Begriffe bilden die Basisterminologie für TYCOON für die folgenden Kapitel:

Tycoon Architektur: Hierunter werden die für Tycoon Systeme charakteristische Schichtenbildung und die Verteilung verschiedener Komponenten auf diese Schichten verstanden (siehe Abb. 1.2). Die genaue Anordnung der einzelnen Komponenten, auch im Hinblick auf die Einordnung in verschiedene Schichten, kann zwischen unterschiedlichen Tycoon Systemen variieren. Die Tycoon Architektur legt weiterhin fest, welche Operationen und Repräsentationen an den Schnittstellen zwischen den Schichten Verwendung finden.

Tycoon System (Familie): Ein Tycoon System bezeichnet eine konkret existierende Implementierung der Tycoon Architektur. TYCOON Systeme unterscheiden sich im wesentlichen durch die im Übersetzer (Sprachebene) realisierte Programmiersprache, können aber auch in anderen Aspekten wie der Objektspeicherimplementierung differieren.

[7] TYCOON Store Protocol
[8] TYCOON Virtual Machine
[9] TYCOON language
[10] TYCOON language minimal

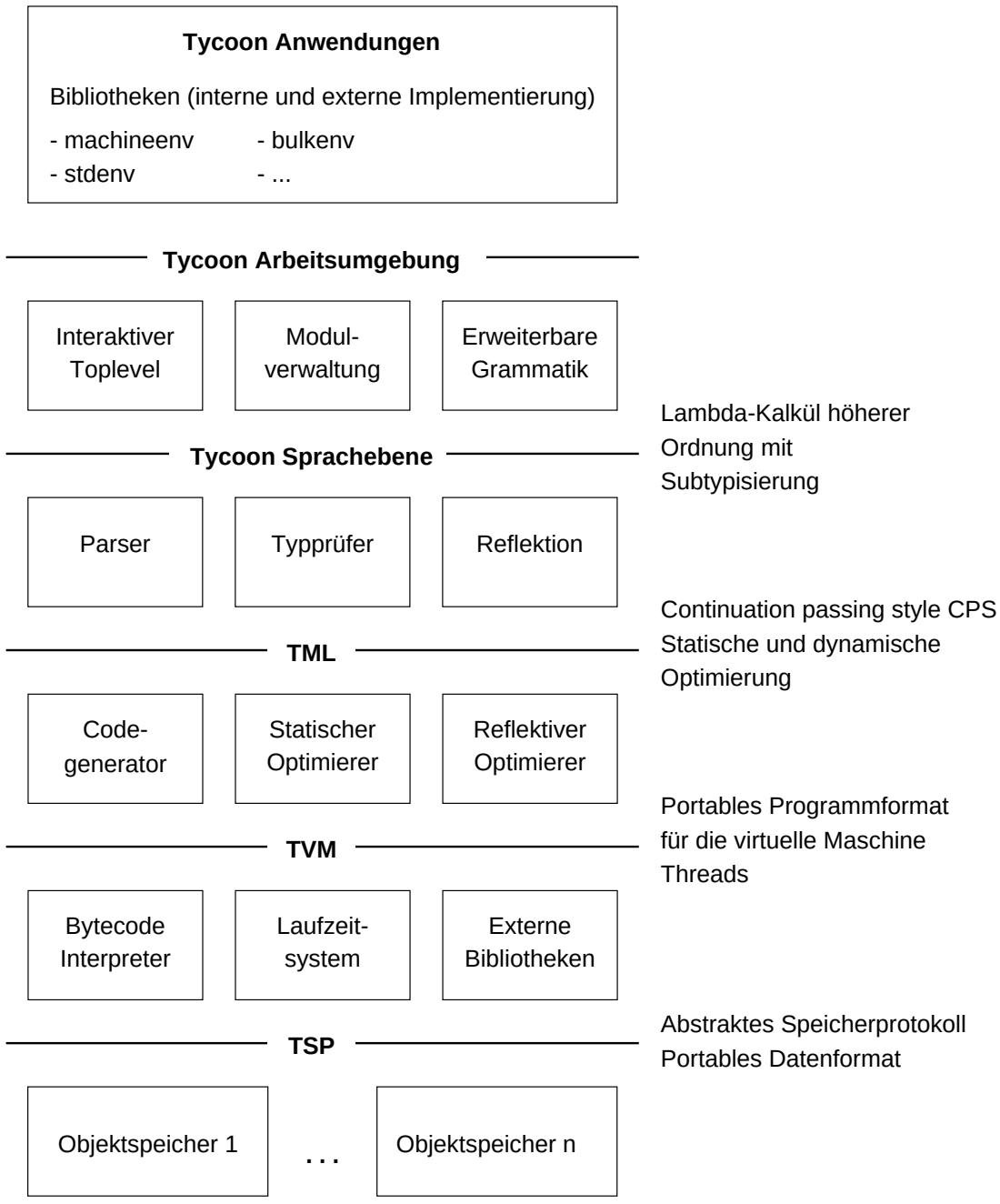

Abbildung 1.2: Die TYCOON Architektur

Die Gesamtheit aller Implementierungen der Tycoon Architektur wird Tycoon System Familie genannt. Die Tycoon System Familie ist weiter in (Unter)familien aufgeteilt, die in bezug auf die Tycoon Architektur Gemeinsamkeiten aufweisen.

Der Begriff Tycoon System (im Singular) wird in dieser Arbeit dann gebraucht, wenn ein beliebiger Vertreter aus der Familie der Tycoon Systeme gemeint ist. Abweichend zu [Mat93] ist mit Tycoon System also nicht ein bestimmtes System gemeint, welches TL als Programmiersprache besitzt. Es wird versucht, eine klare Trennung zwischen Programmiersprache einerseits und implementierendem System andererseits vorzunehmen.

Tl System: Dieser Begriff benennt ein Mitglied aus der Familie der Tycoon Systeme, genauer ein Mitglied der Unterfamilie der Systeme, die die Programmiersprache TL implementieren.

Alle Ausprägungen der Unterfamilie der TL Systeme besitzen die gleiche Sprachebene, können aber in anderen Schichten variieren. Dies verdeutlicht auch Abbildung 1.3.

TlMin System: Weitere Mitglieder in der Familie der Tycoon Systeme sind die TLMIN Systeme. Analog zur Unterfamilie der TL Systeme existiert eine Unterfamilie von TLMIN Systemen. Jedes einzelne Element dieser Unterfamilie wird seinerseits TLMIN System genannt. Alle TLMIN Systeme haben ebenfalls die Eigenschaft, sich untereinander in der Sprachebene zu gleichen.

Ein TLMIN System realisiert die Programmiersprache TLMIN. In TLMIN Systemen werden ähnliche Konzepte wie in TL Systemen verwendet. Aus diesem Grund ist es möglich, daß die unter der Sprachebene liegenden Schichten eines TL und eines TLMIN Systems gleich sind. Dies gilt z.B. für die Implementation der Objektspeicherschicht. Zusätzlich zur Sprachebene umfaßt ein TLMIN System auch eine andere Ausprägung der Schicht der Arbeitsumgebung (TLMIN Arbeitsumgebung).

Tl: Unter diesem Begriff versteht man die durch ein TL System angebotene Programmiersprache, die definiert ist durch eine Beschreibung der Syntax und der statischen und dynamischen Semantik in [Mat93]. Synonym werden die Begriffe TL Sprache oder Sprache TL angewendet.

TlMin: Analog der Definition für TL versteht man hierunter die durch ein TLMIN System angebotene Programmiersprache, deren abstrakte und konkrete Syntax sowie die statische Semantik in Form der Typregeln in [Sch94] bzw. [Bre96] angegeben ist. Die dynamische Semantik entspricht der Semantik der Sprache TL. Im Unterschied zu TL beschränkt sich TLMIN auf einen schmaleren Sprachkern.

Synonym zu TLMIN werden die Begriffe TLMIN Sprache oder Sprache TLMIN angewandt.

Der Begriff Tycoon System steht also für eine Menge von Systemen, die wiederum zu unterschiedlichen Gruppen (Unterfamilien) zusammengefaßt werden können. In Abbildung 1.3 sind die Tycoon Systeme nach der implementierten

Programmiersprache, der verwendeten Objektspeicherimplementierung und der ausführenden Tycoon Maschine partitioniert. Hervorgehoben ist die Menge der TLMIN Systeme, die eine mögliche Zusammenfassung einer Untermenge der Tycoon Systeme bildet.

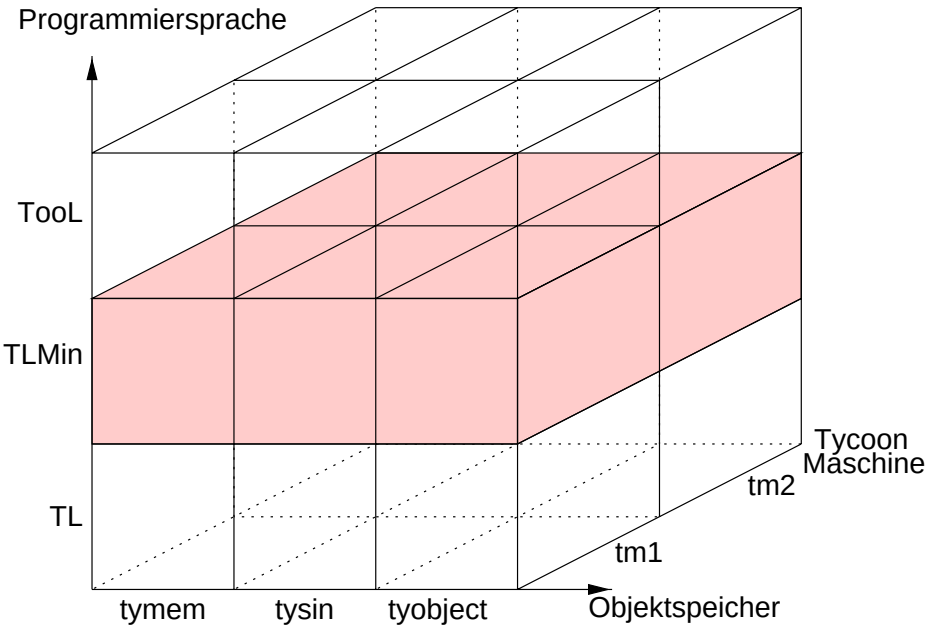

Abbildung 1.3: Mögliche Partitionierung der Tycoon Systeme.

Tycoon Systeme können nicht nur anhand ihrer unterschiedlichen Schichtenimplementierungen partitioniert werden. Auch andere Aufteilungen lassen sich finden, wie z.B. die Fähigkeit, nativen Maschinencode zu erzeugen. Im Rahmen dieser Arbeit wird diese Unterteilung herangezogen, um zwischen den Familien der TL und der TLMIN Systeme zu differenzieren und um zu betonen, daß die anderen Unterschiede für diese Arbeit nicht relevant sind.

Kapitel 2

Dynamisches Rebinden von Objekten

Daten in Computersystemen lassen sich prinzipiell als Objekte betrachten, die in Objektspeichern gespeichert sind (vgl. [Bro88, BM92]). Zu den Aufgaben eines Computerbetriebssystems gehört die Speicherverwaltung. Läßt sich Speicher dynamisch Belegen und Freigeben, so kann dieser Teil des Betriebssystem als Objektspeicher verstanden werden, wobei die belegten und freigegeben Speicherbereiche die Objekte sind. Eine relationale Datenbank, die Daten in Tabellen speichert, ist ein persistenter Objektspeicher. Die Objekte sind die Zeilen der Tabellen. Der Stapelspeicher (engl. stack) eines Prozesses kann als Objektspeicher betrachtet werden; die Objekte sind die Aufrufrahmen (engl. callframes) der bereits aktivierten Funktionen.

Ein Objekt enthält Daten und wird durch einen Bezeichner eindeutig identifiziert. Objekte in Objektspeichern stehen miteinander in Beziehung, wobei zwischen expliziten und impliziten Beziehungen unterschieden werden kann. Explizite Beziehungen lassen sich an der Form der Objekte erkennen. Sind Referenzen innerhalb der Daten eines Objektes unterscheidbar, d.h. sind Referenzen identifizierbar, so ist die referentielle Beziehung zwischen Objekten eine explizite Beziehung. Wären die Referenzen eines Objektes nicht von den anderen Daten unterscheidbar, so wäre die referentielle Beziehung eine implizite Beziehung.

Objektspeicher, die als solche benannt werden, verwenden meist ein uniformes Objektmodell, Objekte sind hierbei nur elementar getypt oder völlig ungetypt. Die Schnittstellen dieser Speicher enthalten Funktionen zur dynamischen Erzeugung von Objekten und es werden meist syntaktische referentielle Beziehungen zwischen Objekten unterstützt.

Dynamisches Rebinden bezeichnet das Verändern referentieller Beziehungen zwischen Objekten sowie das Ersetzen der Bindung eines Objektes an ein anderes

Objekt durch eine neue Bindung.

Folgende Anwendungen nutzen dynamische Rebindung:

- Anwendungen, die Daten oder Programmobjekte[1] aktualisieren, nutzen dynamisches Rebinden; z.B. kann eine Anwendung alle Referenzen eines Programmes auf eine Funktion **A** durch Referenzen auf eine andere Funktion **B** ersetzen. Die nächste Aktivierung des Programmes verwendet daraufhin die Funktion **B** (siehe Abschnitt 4.1).

- Dynamisches Rebinden kann von Anwendungen genutzt werden, die Objekte bzw. Objektgraphen migrieren (siehe Kapitel 3) lassen. Beispielsweise können ubiquitäre Objekte (siehe Abschnitt 4.3) aus einem migrierenden Objektgraphen entfernt werden. Vor der Migration des Objektgraphen werden ubiquitäre Objekte durch diese Objekte beschreibende Objekte ersetzt. Nach der Migration des Objektgraphen werden die Objekte, die nun an Beschreibungen ubiquitärer Objekte gebunden sind, an Ausprägungen der ubiquitären Objekte im Zielobjektsystem neu gebunden (vgl. [Mat96]).

- Anwendungen, die Objektgraphen migrieren (siehe Kapitel 3) lassen wollen, die auf Grund eines immobilen Objektes in ihrer transitiven Hülle insgesamt immobil sind, verwenden dynamisches Rebinden (vgl. [Mat96]). Vor der Migration des Objektgraphen werden immobile Objekte durch diese Objekte beschreibende Objekte ersetzt. Nach der Migration werden die beschreibenden Objekte durch neu erzeugte, zu denen im Ursprungsobjektsystem äquivalente Objekte ersetzt (siehe Abschnitt 4.2).

- Werden Objektgraphen von einem Objektspeicher in einen zweiten bewegt (engl. to move vs. engl. to copy), so verbleibt im Ursprungsobjektspeicher entweder eine Kopie des Objektgraphen oder die Bindungen an den Objektgraphen werden ungültig, was zu ungültigen Referenzen (engl. dangling references) führt. Dynamische Rebindung erlaubt es, im Ursprungsobjektspeicher Repräsentanten für die Objekte des Objektgraphen zu hinterlassen.

- Inhärent transiente Objekte können mit Hilfe dynamischer Rebindung, pseudo-persistent (vgl. [Mat96]) gemacht werden. Z.B. können Kommunikationsendpunkte eines Objektgraphen vor der persistenten Speicherung durch Beschreibungen ersetzt werden, um sie, nach der Reintegration in ein Objektsystem, wieder durch konkrete Kommunikationsendpunkte zu ersetzen.

- Eine weitere Anwendung dynamischer Rebindung, insbesondere in langlebigen Systemen, ist die dynamische Schemaänderung. Z.B. werden alle Vorkommnisse eines Types durch äquivalente Instanzen eines anderen (erweiterten) Typs ersetzt.

[1] Mit dem Begriff Programmobjekt werden im weiteren Quelltexte oder von Quelltexten abgeleitete Objekte, z.B. Syntaxbäume, Parsebäume oder Code bezeichnet (siehe Abschnitt 4.1).

- Andere Anwendungen, die Objektgraphen migrieren (siehe Kapitel 3) lassen, reduzieren diese, z.B. aus Sicherheitsaspekten oder um die Netzlast zu reduzieren, um Programmobjekte. Um ungültige Referenzen zu vermeiden, muß der migrierende Objektgraph im Zielobjektsystem an äquivalente Programmobjekte neu gebunden werden (siehe Abschnitt 4.1).

2.1 Einführung in Objektspeicher

Objektspeicher speichern Objekte unterschiedlicher Ausprägung. Ein Objekt ist ein einfaches oder ein zusammengesetztes Datum mit einem eindeutigen Bezeichner (engl. object identifier, kurz OID). Objekte innerhalb eines Objektspeichers können in referentieller Beziehung zueinander stehen, wobei diese Beziehungen gerichtet sind.

2.1.1 Die Referenzrelation

Eine wichtige Relation zwischen Objekten in Objektspeichern ist die Referenzrelation. Zwei Objekte stehen in referentieller Relation, wenn mindestens eines der beiden Objekte einen Verweis auf das andere Objekt hat. Die Referenzrelation ist eine transitive Relation.

Beinhaltet ein Objekt eine Referenz auf ein anderes Objekt, so ist das referenzierende Objekt an das referenzierte Objekt gebunden. Die beiden Objekte stehen in direkter referentieller Beziehung. Ist das referenzierte Objekt jedoch nur über eine Folge von Referenzen vom referenzierenden Objekt aus erreichbar (transitiv), so stehen die beiden Objekte in indirekter Beziehung. Ein Objekt kann prinzipiell mit einer beliebig großen Menge von Objekten in direkter Beziehung stehen. Ein Objekt kann also an eine Menge von anderen Objekten gebunden sein.

Mit dem Begriff der referentiellen transitiven Hülle eines Objektes wird die Menge all jener Objekte bezeichnet, die vom ursprünglichen Objekt aus direkt oder indirekt erreichbar sind.

2.1.2 Objektspeicherarten

Objektspeicher können nach unterschiedlichen Kriterien spezifiziert und kategorisiert werden:

- Objektspeicher können nach der Art der unterstützten referentiellen Beziehungen zwischen Objekten differenziert werden.

- Objektspeicher können auch nach Eigenschaften unterschieden werden. Wichtige Eigenschaften sind Persistenz (vgl. [STS97]), plattformunabhängige Darstellung der Objekte, Multithreadingfähigkeit, Offenheit.

Der Unterschied zwischen Objektspeichern, die unterschiedliche referentielle Beziehungen zwischen Objekten unterstützen, liegt im Speicherfreigabemechanismus. Objektspeicher, die nur lineare oder baumartige Beziehungen unterstützen, benötigen nur einen einfachen Speicherfreigabemechanismus, da ein Objekt nur von genau einem anderen Objekt referenziert werden kann. Wird ein Objekt gelöscht, so können alle von diesem Objekt aus erreichbaren Objekte auch gelöscht werden (also alle Objekte der referentiellen transitiven Hülle des Objektes).

Objektspeicher, die beliebige azyklische referentielle Beziehungen unterstüzten, benötigen für jedes Objekt einen Referenzenzähler. Ein Objekt wird genau dann abgebaut, wenn der Referenzenzähler auf Null heruntergezählt wird.

Objektspeicher, die beliebige referentielle Beziehungen ihrer Objekte unterstützen, benötigen einen expliziten Speicherrückgewinnungsmechanismus (engl. garbage collection). Dieser Speicherrückgewinnungsmechanismus wird meist in zeitlich konstanten Abständen aktiviert, um Objekte abzubauen, die nicht von anderen Objekten direkt oder indirekt (siehe Abschnitt 2.1.1) referenziert werden. Je nach Objektspeicher kann es ein (Wurzelobjekt) oder mehrere ausgezeichnete Objekte geben, die stabil sind, d.h. die auch dann nicht abgebaut werden, wenn sie nicht referenziert werden. Diese Objekte bilden die Entscheidungsbasis für den Speicherrückgewinnungsmechanismus, um zu ermitteln, welche Objekte abgebaut werden sollen und welche nicht. Alle Objekte, die sich nicht in der transitiven referentiellen Hülle eines stabilen Objektes befinden, werden vom Speicherrückgewinnungsmechanismus freigegeben.

Der Inhalt eines persistenten Objektspeichers bleibt auch dann erhalten, wenn die Applikation, die den Objektspeicher instantiiert hat, terminiert. Bei einem Neustart der Applikation kann der Inhalt des Objektspeichers anschließend wiederhergestellt werden.

Objektspeicher, deren Objekte plattformunabhängig dargestellt werden oder in eine plattformunabhängige Darstellung überführbar sind oder dieselbe Darstellung haben, können Objekte bzw. Objektgraphen untereinander austauschen. Diese Eigenschaft ist die Basis für die Kooperation von Objektsystemen unterschiedlicher Computerplattformen (siehe Kapitel 3).

Basis für einen Speicherrückgewinnungsmechanismus ist die Möglichkeit, einen objektunabhängigen Zugriff auf die Referenzen eines Objektes zu haben, d.h. jedes Objekt erlaubt es, die Referenzen dieses Objektes auf andere Objekte abzufragen (syntaktische referentielle Objektbeziehung). Ist auch ein verändernder Zugriff auf die Referenzen eines Objektes möglich, so können Objekte innerhalb des Objektspeichers dynamisch neu gebunden werden (siehe Abb. 2.1). Ein Ob-

jekt **A** referenziert ein Objekt **B**, Objekt **C** ist eine verbesserte Version von **B**. Ziel ist es, daß die Referenz des Objektes **A** zukünftig auf **C** verweist. Ist es möglich, Referenzen eines Objektes zu verändern, so wird dieses Ziel dadurch erreicht, daß die Referenz des Objektes **A** auf das Objekt **B** durch eine Referenz auf das Objekt **C** ersetzt wird.

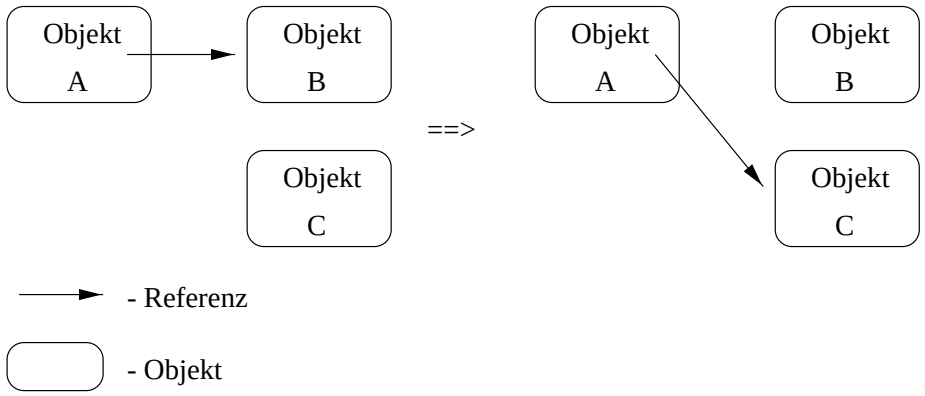

Abbildung 2.1: Rebinden eines Objektes

Ein Beispiel für einen Objektspeicher ist ein Dateisystem (siehe Abb. 2.2). Innerhalb eines Dateisystems eines UNIX-Betriebssystems gibt es Verzeichnisobjekte und Datenobjekte. Verzeichnisobjekte enthalten Verweise auf weitere Objekte, welche wiederum Verzeichnisobjekte oder Datenobjekte sind. Datenobjekte und Verzeichnisobjekte können von einer beliebigen Anzahl von Verzeichnisobjekten referenziert werden. Ein ausgezeichnetes Verzeichnisobjekt, die Wurzel des Objektgraphen, ist die stabile Basis des Objektspeichers. Es werden nur jene Objekte abgebaut, die transitiv von der Basis aus nicht erreichbar sind. Zyklische Strukturen sind innerhalb der Objektgraphen nicht erlaubt. Daraus folgt, daß es genügt, für jedes Objekt die Verweise zu zählen; gibt es keine Verweise mehr auf ein Objekt, so kann der belegte Speicher freigegeben werden. Sind z.B. drei Verzeichnisobjekte **A**, **B** und **C** vom Wurzelverzeichnisobjekt aus erreichbar, und referenziert das Verzeichnisobjekt **A** das Datenobjekt **A1**, sowie das Verzeichnisobjekt **B** die Datenobjekte **A1** und **B1**, und wird die Referenz auf das Verzeichnisobjekt **B** aus dem Wurzelobjekt entfernt, so sind das Verzeichnisobjekt **B** und das Datenobjekt **B1** vom Wurzelobjekt aus nicht mehr erreichbar und werden abgebaut.

Ein Dateisystem eines UNIX-Betriebsystems ist ein persistenter Objektspeicher mit möglichen nebenläufigen Zugriffen und der Unterstützung von gerichteten azyklischen Objektgraphen.

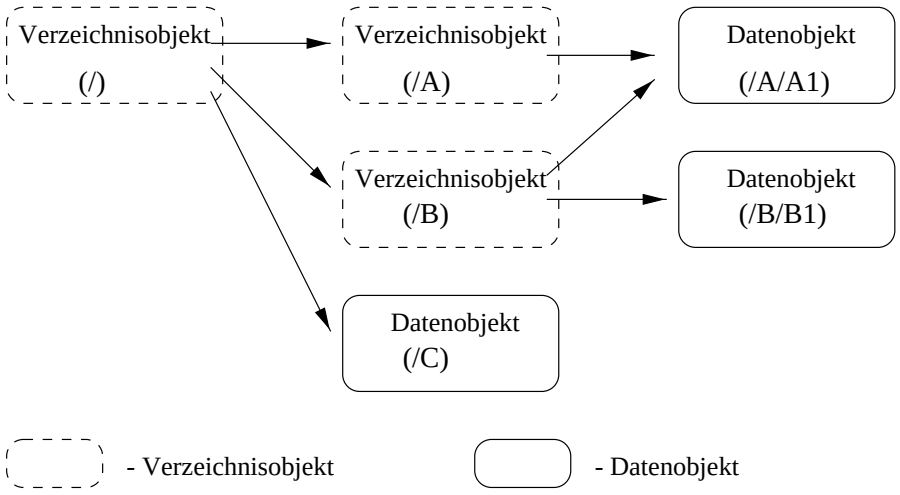

Abbildung 2.2: UNIX-Dateisystem

2.1.3 Manipulation von Objekten

Verschiedene Kontroll- und Manipulationsmöglichkeiten auf Objekten werden benötigt, um mit Objekten in Objektspeichern zu arbeiten. Nachfolgend sind Kontroll- und Manipulationsmöglichkeiten auf Objekten aufgelistet, die für dynamische Rebindung relevant sind:

- Objektdaten können gelesen und geschrieben werden.
- Referenzen können von anderen Objektdaten unterschieden werden (syntaktische referentielle Objektbeziehung).
- Die Größe des Datenbereiches eines Objektes ist veränderbar (bzw. der Datenbereich eines Objektes kann ausgetauscht werden).
- Bei getypten Objektspeichern können die einzelnen Objekttypen ineinander überführt werden.
- Für, vom Objektspeicher zur Verfügung gestellte, Operationen auf Objekten sind Rückruffunktionen (engl. callbacks) anmeldbar.

Abhängig davon, welche Manipulationsmöglichkeiten auf Objekten zur Verfügung stehen, können unterschiedliche Arten von dynamischer Rebindung entwickelt und implementiert werden (siehe Abschnitt 2.2).

2.1.4 Das Objektspeichermodel von Tycoon

Als Basis für die Beispielimplementation der dynamischen Rebindung und der Beispielanwendung für dynamisches Rebinden wird das TYCOON System verwendet. Das TYCOON System nutzt zur persistenten Speicherung von Objekten einen Objektspeicher, der beliebige referentielle Beziehungen zwischen Objekten unterstützt und einen Speicherrückgewinnungsmechanismus zur Freigabe nicht erreichbarer Objekte verwendet. Um die Unabhängigkeit des TYCOON Systems von einer speziellen Objektspeicherimplementation zu wahren, ist das TSP definiert. Das TSP stellt eine abstrakte Schnittstelle für Objektspeicher dar. Verschiedene Implementationen dieser Schnittstelle erlauben es, z.B. NAPIER oder ObjectStore als Objektspeicher für TYCOON zu verwenden. Am Arbeitsbereich sind auch eigene Objektspeicher implementiert worden, wie z.B. TyMem oder TySin, welche hauptspeicher- bzw. dateibasiert sind. Eine auf Basis des TSP bzw. TYCOONs entwickelte Implementation dynamischer Rebindung kann auch prinzipiell TYCOON-Objekte in den genannten Objektspeichern neu binden.

2.1.4.1 Objekte im TSP

Objekte, die in einem TSP konformen Objektspeicher gespeichert werden, bestehen aus einem Objektkopf und einem Objektrumpf (siehe Abb. 2.3). Der Kopf eines Objektes enthält Verwaltungsdaten, wie z.B. die Anzahl der Speicherplätze (engl. slots) im Objektrumpf, sowie das Format der Speicherplätze. Im Rumpf eines Objektes werden die vom TSP-Klienten mit diesem Objekt assoziierten Daten des TSP-Klienten gespeichert. Abhängig von der Anzahl der Speicherplätze kann ein Objekt unterschiedlich viele Daten aufnehmen. Die Speicherplätze eines Objektes sind uniform, d.h. in allen Speicherplätzen eines Objektes werden Daten gleichen Typs gespeichert, es kann also der Typ der Speicherplätze eines Objektes mit dem Objekt selbst assoziiert werden. Objekte gleichen Typs können sich dann nur noch durch die Anzahl ihrer Speicherplätze und durch den Inhalt der Speicherplätze unterscheiden. Die Reihenfolge der Speicherplätze eines Objektes ist eindeutig, d.h. durch OID und Index des Speicherplatzes ist jedes Datum innerhalb des Objektspeichers eindeutig identifiziert.

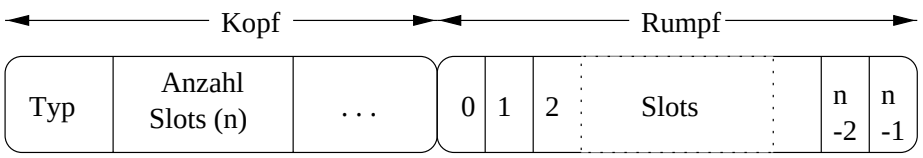

Abbildung 2.3: Aufbau eines TSP-Objektes

Zwischen folgenden generellen Datenformaten von TSP-Objekten wird unterschieden:

direkte Werte (engl. immediate values): Objekte, deren Format vom Typ **tsp_Format_IMMEDIATE**, **tsp_Format_BYTE** oder **tsp_Format_REAL** ist, enthalten nur Werte (Ganzzahlen, Bytes oder Gleitkommazahlen).

markierte Werte (engl. tagged values): Objekte, deren Format vom Typ **tsp_Format_TAGGED** ist, enthalten entweder Ganzzahlenwerte oder Verweise auf andere Objekte. Verweise und Werte werden anhand des am weitesten rechts liegenden Bits (Bit 0) differenziert (siehe Abb. 2.4).

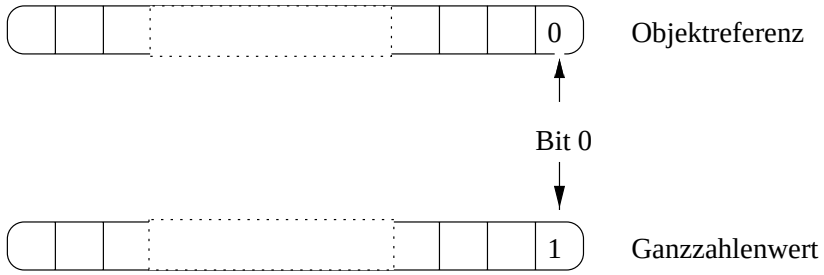

Abbildung 2.4: Markierter Wert

Durch Verwendung von Objekten mit markierten Werten können beliebige referentielle Beziehungen zwischen Objekten im Objektspeicher aufgebaut werden.

Durch diese einfache Typisierung der Objekte des Objektspeichers wird die Linearisierung, d.h. die Überführung in eine sequentielle Folge von Daten, von Objektgraphen unterstützt, so daß Objektgraphen zwischen Objektsystemen ausgetauscht werden können, indem der linearisierte Objektgraph im Zielobjektsystem wieder in einen konkreten Objektgraphen überführt wird. Um Objektgraphen zwischen Objektsystemen verschiedener Plattformen austauschen zu können, werden die Objektdaten bei der Linearisierung in eine objektspeicherunabhängige Darstellung transformiert (engl. marshalling).

Das Tsp definiert für Objekte folgende Funktionalität:

Erzeugung: Es können Objekte verschiedener Formate und unterschiedlicher Größe erzeugt werden.

Manipulation: Die Speicherplätze eines Objektes können ausgelesen und beschrieben werden (vorausgesetzt, der Objektzustand ist nicht eingefroren).

Einfrieren: Der Status eines Objektes kann auf unveränderbar (engl. immutable) gesetzt werden. Objekte, die unveränderbar sind, können nur ausgelesen, aber nicht mehr beschrieben werden.

Verwaltung: Es können Metainformationen über Objekte erfragt werden, wie z.B. das Objektformat, die Anzahl der Speicherplätze und ob ein Objekt unveränderbar ist.

Referenzen: Mit Hilfe einer Aufzählungsfunktion (engl. enumerator function) können Objektreferenzen (OIDs) extern, d.h. außerhalb des Objektspeichers, gehalten werden. Diese Funktion muß vom Klienten des Objektspeichers zur Verfügung gestellt werden.

Im-/Export: Objekte können mitsamt ihrer transitiven referentiellen Hülle externalisiert, d.h. in eine plattformunabhängige Darstellungsform überführt werden. Plattformunabhängige Objektgraphen können internalisiert werden, diese Objektgraphen werden dabei in den Objektspeicher integriert.

2.1.4.2 Laufzeitrepräsentation von Tycoon-Objekten im Objektspeicher

Alle Objekte, die zur Laufzeit vom TYCOON System erzeugt werden, werden im Objektspeicher abgelegt. Das Übersetzen von Programmen erfolgt komplett innerhalb des Laufzeitsystems (im Gegensatz z.B. zu Java; vgl. [Fla97]), da der TYCOON Übersetzer selbst eine TYCOON Anwendung ist. Alle Zwischenrepräsentationen von Programmcode (z.B. Parsebäume, Syntaxbäume) werden als Objekte in den Objektspeicher übertragen. Auch sämtliche zur Laufzeit erzeugten Strukturen (Tupel, Funktionen, Arrays, ...) werden in Form von Objekten gespeichert.

Z.B. wird folgendes Tupel, wie in Abbildung 2.5 gezeigt, im Objektspeicher abgelegt:

Let Rec *T <:* ***Ok = Tuple*** *:T :String :Array(Int)* ***end;***

let rec *t <: T =* ***tuple*** *t "Zeichenkette"* ***array*** *1 2 3* ***end end;***

Die Eigenschaft, alle Daten im System als Objekte im Objektspeicher abzulegen, ist die Basis für Reflektion (siehe Abschnitt 5.3) und für Anwendungen dynamischer Rebindung (vgl. [KCMS96]).

2.2 Dynamisches Rebinden

Dynamisches Rebinden bezeichnet das Ersetzen einer Bindung eines Objektes an ein anderes Objekt durch eine neue Bindung. Referenziert ein Objekt **A** ein Objekt **B**, so ist das Objekt **A** an **B** gebunden. Wird Objekt **A** an Objekt **C** neugebunden, so wird dies als dynamische Rebindung von **A** an **C** bezeichnet, wobei die ursprüngliche Bindung an **B** verlorengeht.

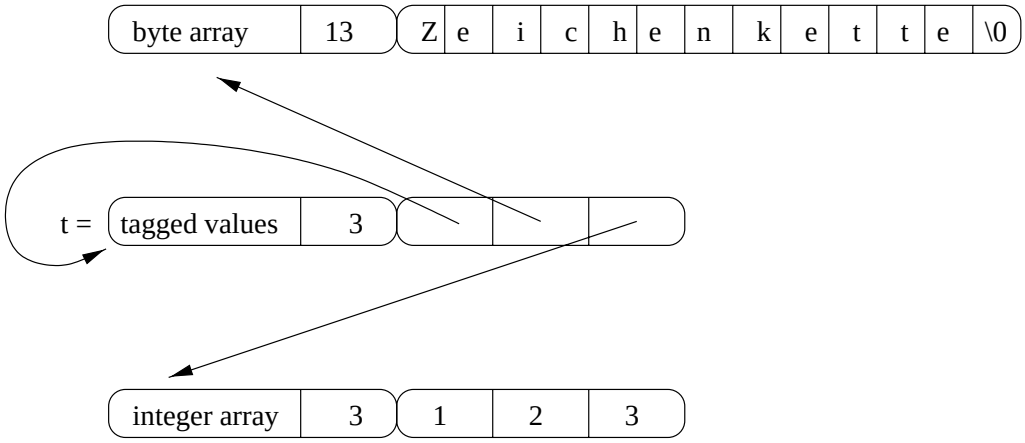

Abbildung 2.5: TYCOON-Laufzeitobjekte

Es gibt verschiedene Möglichkeiten, dynamisches Rebinden zu implementieren. Die Auswahl einer dieser Möglichkeiten hängt von der Art des zu verwendenden Objektspeichers sowie von der gewünschten Art der Rebindung ab.

Zwei Arten von Rebindung können unterschieden werden:

- Alle Bindungen von Objekten an ein bestimmtes Objekt sollen ersetzt werden (totale Rebindung).
- Es soll nur eine Auswahl von Bindungen an ein bestimmtes Objekt ersetzt werden (partielle Rebindung).

Sollen Objekte neu gebunden werden, so sind Objektdaten zu verändern, wobei einerseits Daten gebundener Objekte und andererseits Daten bindender Objekte verändert werden können. Da die partielle Rebindung auch die totale Rebindung ermöglicht, ist ein Objektspeicher rebindungsvollständig (partielle und totale Rebindung), wenn partielle Rebindung realisiert werden kann.

Folgende Techniken der Objektmanipulation für Rebindung sind unterscheidbar:

- Das gebundene Objekt wird verändert. Die Referenz des gebundenen Objektes an das bindende Objekt wird durch eine Referenz an das neubindende Objekt ersetzt.
- Der Inhalt des bindenden Objektes wird durch den Inhalt des neubindenden Objektes ersetzt; eventuell vorhandene Bindungen an das neubindende Objekt gehen dabei verloren bzw. Bindungen an das neubindende Objekt werden inkonsistent (siehe Abb. 2.6).

- Das bindende Objekt wird in einen Repräsentanten[2] (engl. proxy) für das neubindende Objekt umgewandelt (siehe Abb. 2.8). Die Referenz des gebundenen Objektes verweist anschließend indirekt auf das neubindende Objekt.

Diese Arten der Objektmanipulation für Rebindung können auch kombiniert oder fallabhängig verwendet werden. Welche Techniken konkret für Rebindung verwendet werden können, hängt von den Möglichkeiten des Objektspeichers ab.

Tabelle 2.1 gibt einen Überblick über Objektmanipulationen, über die Voraussetzungen der Objektmanipulationen und über Rebindungsarten. Durch Änderung der Verweise gebundener Objekte ist partielle Rebindung implementierbar. Kann der Inhalt des bindenden Objektes durch den Inhalt eines anderen Objektes ausgetauscht werden, so ist totale Rebindung möglich. Erlaubt es der verwendete Objektspeicher für einzelne Objekte Rückruffunktionen zu registrieren, so kann zumindest totale Rebindung realisiert werden. Sind die Rückruffunktionen mit dem Objekt parametrisiert, das die Operation anstößt, so ist auch partielle dynamische Rebindung möglich.

Voraussetzung	Manipulation	Rebindung
Referenzänderung (syntaktisch)	Referenzmanipulation	partiell
Objektdatenveränderung	Objektinhaltersetzung	total
Rückruffunktionen	Indirektion	total
Rückruffunktionen mit Referenzparameter	Indirektion	partiell

Tabelle 2.1: Rebindungsarten und Voraussetzungen

Vorteil der Rebindung durch Manipulation des bindenden Objektes ist es, daß sofort alle gebundenen Objekte die Änderungen sehen, die Rebindung also atomar, in einem Schritt, zu vollziehen ist.

Können die Daten des bindenden Objektes durch die Daten des neubindenden Objektes ersetzt werden, so läßt sich hiermit Rebindung realisieren (siehe Abb. 2.6). Um die Daten des neubindenden Objektes in das bindende Objekt zu übertragen, kann es notwendig sein, den Typ oder die Größe des Datenbereiches des bindenden Objektes zu verändern, dies ist abhängig von den Möglichkeiten des Objektspeichers. Neubindung durch Ersetzung des Inhaltes des bindenden Objektes ist insofern problematisch, als daß ein Vergleich der Bindungen zweier Objekte A und B an ein bindendes Objekt C, von denen eines mittels Rebindung gebunden wurde, ein Ungleich zum Ergebnis hat (siehe Abb. 2.7). Dies kann insbesondere bei partieller Rebindung, abhängig von der Intension der Rebindung, Inkonsistenzen zur Folge haben.

[2]Ein Repräsentant steht stellvertretend für ein bestimmtes Objekt. Alle Operationen auf einem Objektrepräsentaten werden auf das zugehörige Objekt umgeleitet.

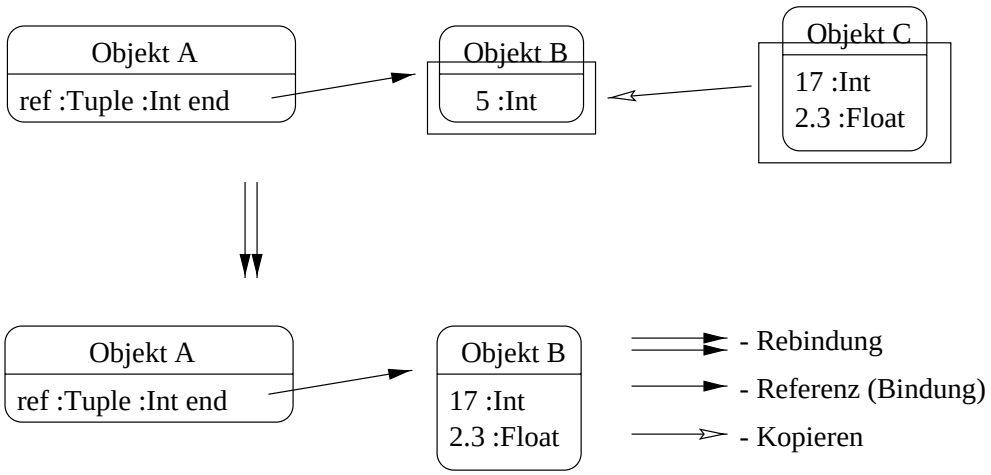

Abbildung 2.6: Rebinden durch Ersetzung

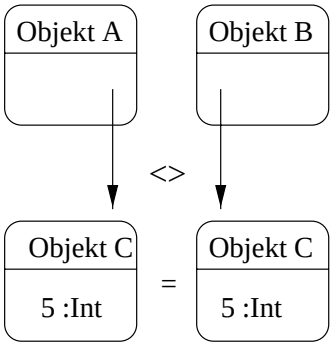

Abbildung 2.7: Inkonsistenz bei partieller Rebindung durch Ersetzung

Erlaubt es der verwendete Objektspeicher, Rückruffunktionen für die auf Objekten möglichen Operationen anzumelden (siehe Abb. 2.8), so läßt sich Rebindung durch die Überführung des ursprünglich bindenden Objektes in einen Repräsentanten für das neubindende Objekt umsetzen. Objekt A ist an Objekt B gebunden, diese Bindung soll durch eine Bindung des Objektes A an das Objekt C ersetzt werden. Objekt B wird in einen Repräsentanten mit Verweis auf Objekt C umgewandelt. Objekt A referenziert anschließend Objekt C transparent indirekt über Objekt B. Diese Technik vermeidet das Verlieren der schon vorhandenen Bindungen, bei partieller Rebindung, an das neubindende Objekt und mögliche Inkonsistenzen. Die Parametrisierung der Rückruffunktionen mit dem referenzierenden Objekt erlaubt es, auch teilweise Neubindung durch Repräsentanten zu implementieren. In diesem Fall ist mit Repräsentanten Rebindungsvollständigkeit zu erreichen, mit dem Vorteil der Atomarität.

Das Objektspeicherprotokoll TSP des dieser Arbeit zugrundeliegenden TYCOON

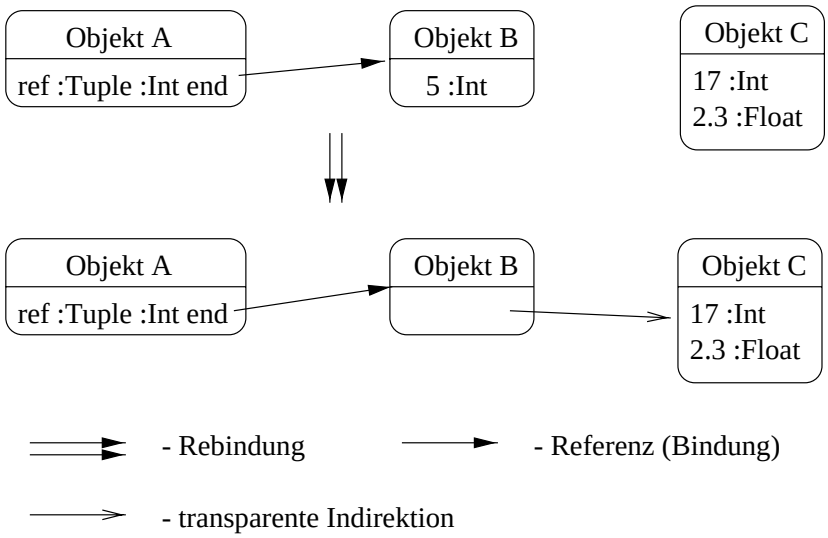

Abbildung 2.8: Rebinden durch einen Repräsentanten

Systems unterstützt syntaktische referentielle Beziehungen zwischen Objekten sowie das Auslesen und Beschreiben von Objektdaten. Die Größe des Datenbereiches eines erzeugten Objektes ist nicht veränderbar. Das TSP bietet keine Funktionalität, um Rückruffunktionen für Objektoperationen anzumelden. Aufgrund dieser Einschränkungen wird in dieser Arbeit nur dynamisches Rebinden durch Ersetzen von Referenzen betrachtet.

2.2.1 Objektaktualisierungen

Persistente Objektsysteme erlauben langlebige Prozesse. Die Lebenserwartung dieser Prozesse ist so groß, daß die von ihnen verwendeten Softwarekomponenten während ihrer Laufzeit weiterentwickelt werden. Es wird also Prozesse geben, die alte Versionen von Softwarekomponenten verwenden. Mit Hilfe dynamischer Rebindung ist es möglich, die aktiven Prozesse an neue Softwarekomponenten neu zu binden.

Abbildung 2.9 zeigt einen aktiven Thread, der eine Funktion **F1** verwendet. Durch dynamische Rebindung wird der Thread an die Funktion **F2** neu gebunden.

Bei der nächsten Aktivierung der Funktion **E** wird die Funktion **F2** aufgerufen.

2.2.2 Typsicherheit

Für dynamische Rebindung wird Objektspeicherfunktionalität verwendet. Objektspeicher liegen auf einer niedrigeren Abstraktionsebene als die den Objekt-

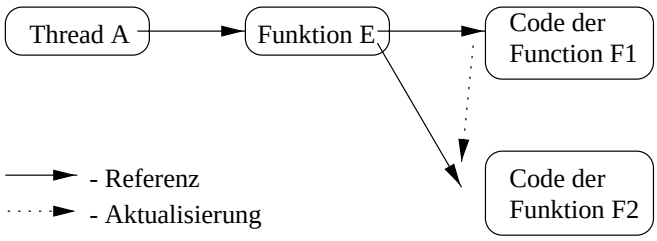

Abbildung 2.9: Aktualisierung eines Programmobjektes

speicher verwendende Programmiersprache. Werden die Objekte des Objektspeichers nicht explizit mit Typinformationen versehen, so kann ein auf Objektspeicherfunktionalität basierender Mechanismus prinzipiell nicht typsicher sein. Wird die Bindung zweier Objekte durch eine neue Bindung ersetzt und haben die beteiligten Objekte keine Typinformationen, so kann der Anwender der Rebindung keinen Typtest vornehmen, ob die neue Bindung typsicher ist.

Z.B. ist Objekt A an Objekt B gebunden. Diese Bindung soll durch eine Bindung an das Objekt C ersetzt werden. Objekt B ist vom Typ Tupel und Objekt C vom Typ Zeichenkette. Werden diese Typinformationen nicht an den Objekten gespeichert, so kann bei Rebindung nicht festgestellt werden, das Objekt C nicht zu Objekt B kompatibel ist.

Dynamisches Rebinden ist typunsicher. Eine Anwendung dynamischer Rebindung ist die typunsichere Erneuerung von Bindungen, z.B. im Zusammenhang mit Ko- und Kontravarianz (siehe Abschnitt 2.2.4). Die semantische Typsicherheit dynamischer Rebindung muß in der Anwendung sichergestellt werden.

2.2.3 Dynamische Schemaänderung

Wird die Struktur eines Typs zur Laufzeit verändert, bzw. wird ein Typ zur Laufzeit durch einen neuen ersetzt (beispielsweise durch dynamische Rebindung dynamischer Typen), so sind alle Instanzen des Typs an die neue Struktur anzupassen. Diese Aufgabe läßt sich mit Hilfe dynamischer Rebindung lösen.

Bespiel 2.1 zeigt eine dynamische Schemaänderung. Die Tupel eines Feldes werden um jeweils ein Attribut ergänzt, das den Wert des Tupels als Zeichenkette beschreibt. Nach der Schemaänderung wird ein neuer Bezeichner vereinbart, der auf dieses Feld verweist. Der Typ des neuen Bezeichners entspricht dem neuen Schema. Die Identitäten der Elemente und des Feldes bleiben erhalten.

```
Let T = Tuple value :Tuple number :Int end end;

let values :Array(T) = arrayOp.new(2 tuple tuple 0 end end);
values[0] := tuple tuple 123 end end; (* Das Feld wird mit zwei Elementen gefüllt *)
values[1] := tuple tuple 456 end end;

(* Es werden die Funktionen für die dynamische Schemaänderung definiert *)
Let New_T = Tuple value :Tuple number :Int  str :String end end;

let update(oid :tsp.OID) :tsp.OID = begin
  if oid == values[0].value \/ oid == values[1].value then
    let t = unsafe.typeCast(oid :Tuple number :Int end)
    print.string("v: " <> fmt.int(t.number) <> "\n")
    let r :tsp.OID = tuple t.number fmt.int(t.number) end
  else
    oid
  end
end;

(* Hilfsfunktion, die hier eigentlich nicht benötigt wird. *)
let post(oid :tsp.OID) :tsp.OID = oid;

rebind.update(values update post); (* Jedes Tupel des Feldes erhält ein neues attribut *)

(* Das geänderte Feld wird unter dem neuen Typ bekannt gemacht *)
let new_values = unsafe.typeCast(values :arrayOp.T(New_T));
```

Beispiel 2.1: Dynamische Schemaänderung

2.2.4 Dynamisches Rebinden versus variable Bindung

Bindungen zwischen Objekten, die zur Laufzeit veränderbar sein sollen, werden als variabel deklariert. Im Gegensatz dazu sind Bindungen, die als konstant bzw. nicht als variabel deklariert werden, nicht veränderbar. Variable Bindungen erlauben es, zur Laufzeit Bindungen zwischen Objekten, abhängig von der Typisierung, zu ersetzen.

Konstante und variable Bindung sind Programmiersprachenkonstrukte und sind somit auf einer semantisch höheren Ebene angesiedelt als dynamische Rebindung, welche nur auf Objektspeicherfunktionalität beruht. Dynamische Rebindung ergänzt die Möglichkeiten variabler und konstanter Bindungen.

Folgende Punkte zeigen die unterschiedliche Anwendbarkeit variabler Bindungen und dynamischen Rebindens:

- Da variable Bindungen ein Programmiersprachenkonzept sind, unterliegen sie, im Gegensatz zu dynamischer Rebindung, den Typzwängen der Pro-

grammiersprache.

- Variable Bindungen müssen schon im vorhinein als solche deklariert werden. Dies ist nicht immer möglich und wünschenswert (siehe Beispiele 2.2 und 2.3).
- Dynamische Rebindung über Repräsentanten stellt ein Konzept dar, das direkt nicht mit variablen Bindungen zu realisieren ist (z.B. Repräsentanten für im Netzwerk verteilte Objekte; Netzwerktransparenz).

Gewisse Funktionssignaturen und Typbeziehungen können durch ein Typsystem ausgeschlossen sein (siehe Beispiel 2.2). Es ist z.B. nicht ohne Aufgabe der Typsicherheit möglich, ein als variabel deklariertes Tupel als Parameter an eine Funktion zu übergeben, deren Signatur diesen Parameter als Supertyp des aktuell übergebenen Parameters definiert (Ko- und Kontravarianz; siehe [Mey90]).

```
Let Person = Tuple name :String end;
Let Student = Tuple name :String  matrikel :String end;

let f(var a :Person) :Ok = begin
  a := tuple "Georg"  "1234567" end
end;

let var b :Student = tuple "Harald"  "8901234" end;

f(b); (* dieser Funktionsaufruf ist semantisch korrekt, wird aber vom Typsystem abgewiesen *)
```

Beispiel 2.2: Ko- und Kontravarianz

Beispiel 2.3 zeigt eine nicht mögliche Erneuerung einer variablen Bindung. Variable **x** kann nicht variabel gebunden werden, da es ansonsten zu Typkonflikten kommen kann. Mit Hilfe dynamischer Rebindung ist es jedoch möglich, **x** neu zu binden.

In verteilten Objektsystemen kann es vorkommen, daß ein Objekt aus einem Objektspeicher in einen anderen Objektspeicher migriert. Bei dieser Migration kann unterschieden werden zwischen einer Bewegung (engl. move) und einem Kopieren (engl. copy) des betreffenden Objektes. Wird ein Objekt von einem Objektspeicher in einen anderen Objektspeicher bewegt, so muß ein Repräsentant (engl. proxy) für das bewegte Objekt im Quellobjektspeicher verbleiben, da ansonsten Referenzen auf das bewegte Objekt ungültig werden würden (es sei denn, es gibt Netzwerkreferenzen). Dieses Ersetzen des Objektes durch einen Repräsentanten ist eine Form dynamischen Rebindens. Können für Objektspeicheroperationen Rückruffunktionen registriert werden, so ist es möglich, Repräsentanten zu realisieren (siehe Abschnitt 2.2). Sollen Repräsentanten durch variable Bindung implementiert werden, so ist der Indirektionsmechanismus explizit auszuprogrammieren und die Verwendung des Repräsentanten wäre nicht mehr transparent.

```
Let ADT = Tuple T <: Ok  new() :T  inc(:T) :T  end;

let var x :ADT = tuple (* diese variable Bindung ist nicht erlaubt *)
  Let T = Int
  let new() :T = 1
  let inc(t :T) :T = t + 1
end;

let adt :x.T = x.new();

x := tuple     (* diese Zuweisung ist semantisch korrekt, jedoch nicht typsicher *)
  Let T = Int
  new() :T = 1
  inc(t :T) :T = t + 2
end;

x.inc(adt);
```

Beispiel 2.3: Nicht erlaubte Erneuerung einer variablen Bindung

Die Technik des dynamischen Rebindens ist, verglichen mit konstanten und variablen Bindungen, ein ergänzendes Verfahren, das je nach Art der im Programm definierten Bindungen verwendet werden kann. Tabelle 2.2 zeigt dynamische Rebindung in Bezug zu Bindungsarten (zu optimierenden Bindungen vgl. Abschnitt 2.2.7).

Bindungsart	Deklaration	typsicher	dynamisch rebindbar
optimierend	statisch	ja	nein
konstant	statisch	ja	ja
variable	statisch	ja	ja

Tabelle 2.2: Dynamische Rebindung, konstante und variable Bindung

2.2.5 Finden von Bindungen

Sollen Bindungen dynamisch ersetzt werden, so müssen die an einer Bindung beteiligten Objekte identifiziert werden. Folgende Möglichkeiten zur Identifizierung können unterschieden werden:

Navigation: Von einem bekannten Objekt aus werden das gebundene und das bindende Objekte jeweils durch Spezifikation eines Pfades definiert.

Identifikation des gebunden Objektes: Das gebundene Objekt und der Index der Bindung sind bekannt.

Identifikation des bindenden Objektes: Das bindende Objekt ist bekannt. Sind auch gebundene Objekte bekannt, so kann die Rebindung entweder partiell oder total erfolgen.

Bei Objektspeichern mit kleinen bis mittleren Objektmengen kann z.B. eine Tabelle mit bindenden Objekten gehalten werden. Sollen Objekte aktualisiert werden, so können die Bindungen mit Hilfe der Tabelleneinträge und des Wurzelobjektes identifiziert werden.

2.2.6 Implementation von Rebindung für Tycoon

Im Rahmen dieser Arbeit wurde für das TYCOON System ein Modul für Rebindung implementiert. Beispiel 2.4 zeigt einen Ausschnitt der Schnittstelle dieses Moduls.

```
mark(oid :tsp.OID  bit :Int) :Ok
(* mark a store object *)

unmark(oid :tsp.OID  bit :Int) :Ok
(* unmark a store object *)

marked(oid :tsp.OID  bit :Int) :Bool
(* test if store object is marked *)

deepCopy(O <: Ok  obj :O  pre(:tsp.OID) :tsp.OID
  mark(:tsp.OID) :Ok  post(:tsp.OID) :tsp.OID) :O

update(O <: Ok  obj :O  pre(:tsp.OID) :tsp.OID  post(:tsp.OID) :tsp.OID) :O
```

Beispiel 2.4: Funktionen des Rebindungsmoduls

Objekte können mit den Funktionen *mark* und *unmark* markiert bzw. demarkiert werden. Konkret wird das Bit, mit der beim Funktionsaufruf angegebenen Nummer, im Typfeld des Objektkopfes gesetzt oder zurückgesetzt. *marked* erlaubt es, die aktuelle Markierung zu ermitteln, indem festgestellt wird, ob das angegebene Bit gesetzt oder nicht gesetzt ist. Die Funktion *deepCopy* kopiert einen Objektgraphen. Es können Funktionen als Parameter angegeben werden, die einzelne Objekte während des Kopierens austauschen (siehe Bespiel 2.5). Die Funktion *pre* wird vor dem Kopieren eines Astes des Objektgraphen für jedes Objekt des Objektgraphen aufgerufen. *mark* wird direkt nach der Erstellung einer Kopie eines Objektes aktiviert. Die Funktion *post* wird aufgerufen, nachdem der aktuelle Ast kopiert ist.

Das Beispiel 2.6 zeigt die Verwendung der Funktion *update*. Die Parameter *pre* und *post* haben den gleichen Zweck wie bei der Funktion *deepCopy*. Der Parameter

```
let t = tuple 123 end;
let r = tuple "ersatz tupel" end;

let pre(oid :tsp.OID) :tsp.OID = if oid == t then
  let h :tsp.OID = r
else
  oid
end;

let mark(oid :tsp.OID) :Ok = begin end;
let post(oid :tsp.OID) :tsp.OID = oid;

let o = tuple t end;
let c = rebind.deepCopy(o pre mark post);
```

Beispiel 2.5: Tiefes Kopieren mit Rebindung

```
let t = tuple 567 end;

let pre(oid :tsp.OID) :tsp.OID = if rebind.marked(oid) then
  t
else
  oid
end;

let post(oid :tsp.OID) :tsp.OID = oid;

let c = rebind.update(r pre post);
```

Beispiel 2.6: Tiefes Aktualisieren mit Rebindung

mark entfällt, da nicht kopiert, sondern nur aktualisiert wird.

2.2.7 Wechselwirkungen mit Optimierern

Bei der Anwendung dynamischer Rebindung gibt es Probleme, die im Zusammenhang mit optimierenden Übersetzern auftreten. Ist z.B. eine Bindung zwischen zwei Objekten als konstant bzw. nicht variabel deklariert, so kann ein optimierender Übersetzer indirekte Zugriffe auf das bindende Objekt durch direkte Zugriffe ersetzen, bzw. er kann indirekte Bindungen durch direkte Bindungen ersetzen. Die optimierten Zugriffe und Bindungen bleiben erhalten und beziehen sich, auch nach Erneuerung der konstanten Bindung mit dynamischer Rebindung, auf das ursprüngliche bindende Objekt. Dieses Problem läßt sich z.B. durch Einführen einer Optimierungsanweisung für den Übersetzer in die Programmiersprache vermeiden. Substitutionsoptimierungen würden nur noch dort angebracht werden,

wo sie explizit erlaubt wären.

Kapitel 3

Kooperierende Objektsysteme

Objektsysteme können mittels unterschiedlicher Medien miteinander kooperieren, eine einfache Art der Kooperation stellt der Datenaustausch z.B. mittels Disketten dar. Anspruchsvollere Szenarien erfordern es, daß Daten automatisch ausgetauscht werden können, d.h. z.B. über Netzwerkverbindungen oder durch gemeinsame Nutzung eines Speichers (Hauptspeicher, Festplatte, ...). Da Netzwerke als Kooperationsmedium entwickelt werden und da Abbildungen von Kooperation mit Netzwerken auf andere Kooperationsmedien gefunden werden können, wird die Kooperation von Objektsystemen anhand von Netzwerken erläutert.

Die Kommunikationen bzw. Kooperation zwischen Programmen in Computernetzwerken ist zweigeteilt. Ein Programm erbringt einen Dienst (Server) und ein zweites Programm erbittet die Erbringung eines Dienstes (Klient). Möchte ein Klient von einem Server einen Dienst erbracht bekommen, so muß der Klient zunächst eine Netzwerkverbindung zum Server aufbauen. Das Aufbauen einer Verbindung geschieht durch das Anfordern eines Kommunikationsendpunktes vom Betriebssystem sowie der Äußerung eines Verbindungswunsches dieses Kommunikationsendpunktes mit einem (über Rechnernamen und lokale Kennung beschriebenen) anderen Kommunikationsendpunkt an das Betriebssystem.

3.1 Computernetzwerke

Heutige Computernetzwerke setzen sich aus Computern heterogener Computerplattformen zusammen (siehe Abb. 3.1). Verschiedene Netzwerkprotokolle stellen Verbindungen zwischen den einzelnen Rechnern her (TCP/IP[1], IPX/SPX[2], NETBIOS[3]).

[1] Transmission Control Protocol / Internet Protocol
[2] Internetwork Package eXchange / Sequence Packet eXchange
[3] Network Basic Input Output System

Netzwerkprotokolle erlauben es, durch Abstraktion von der Netzwerktopologie, einen beliebigen Computer im Netzwerk von einem beliebigen anderen Computer aus direkt zu erreichen, d.h. es können Daten direkt zwischen zwei beliebigen Computern des Netzwerkes ausgetauscht werden.[4]

Jeder Computer des Netzwerkes bekommt einen eindeutigen Namen (dieser wird in der Regel als Zahl kodiert). Computer, die gleichzeitig in mehrere Netzwerke eingebunden sind, sind unter sovielen verschiedenen Namen erreichbar, wie sie Netzwerkschnittstellen haben, d.h. es werden eigentlich nicht die Computer selbst benannt, sondern ihre Netzwerkschnittstellen.

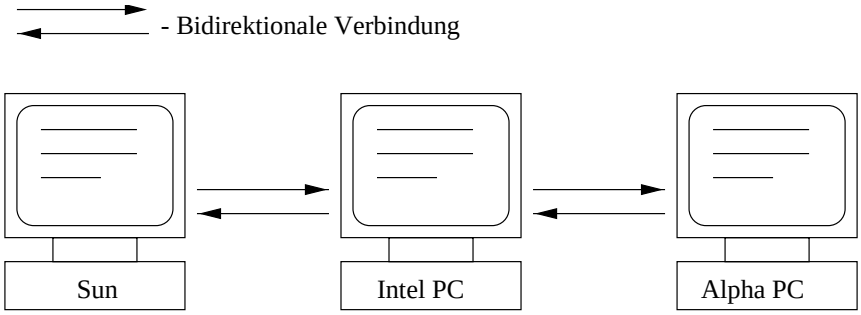

Abbildung 3.1: Computernetzwerk

Daten in Computernetzwerken werden meist als Pakete übertragen. Die meisten Netzwerkprotokolle bieten unterschiedliche Arten des Datenaustausches.

Zwischen folgenden Arten der Datenübertragung kann unterschieden werden:

verbindungslos: Datenpakete werden ungesichert übertragen, d.h. das Netzwerkprotokoll übernimmt den Transport der Datenpakete, garantiert jedoch nicht, daß die Datenpakete in der gleichen Reihenfolge, in der sie abgeschickt wurden, oder daß sie überhaupt beim Empfänger ankommen.

verbindungsorientiert: Die zweite meist angebotene Art der Datenübertragung ist die verbindungsorientierte Übertragung. Bei dieser wird zwischen den kommunizierenden Prozessen eine virtuelle Verbindung aufgebaut. Das zugrundeliegende Übertragungsprotokoll garantiert, daß die abgeschickten Daten in der gleichen Reihenfolge beim Empfänger ankommen, in der sie vom Sender abgeschickt wurden. Virtuelle Verbindungen zwischen Computern, die von Netzwerkprotokollen aufgebaut werden, sind fast immer bidirektional, d.h. Daten können in beiden Richtungen ausgetauscht werden.

Die meisten Netzwerkprotokolle und Programmierschnittstellen für Netzwerkprotokolle unterstützen mehrere virtuelle Verbindungen gleichzeitig (siehe Abb. 3.2).

[4] Firewalls und andere Sicherheitsmechanismen werden nicht betrachtet.

Es können also prinzipiell beliebig viele paarweise Verbindungen zwischen Programmen auf mehreren Computern aktiv sein.

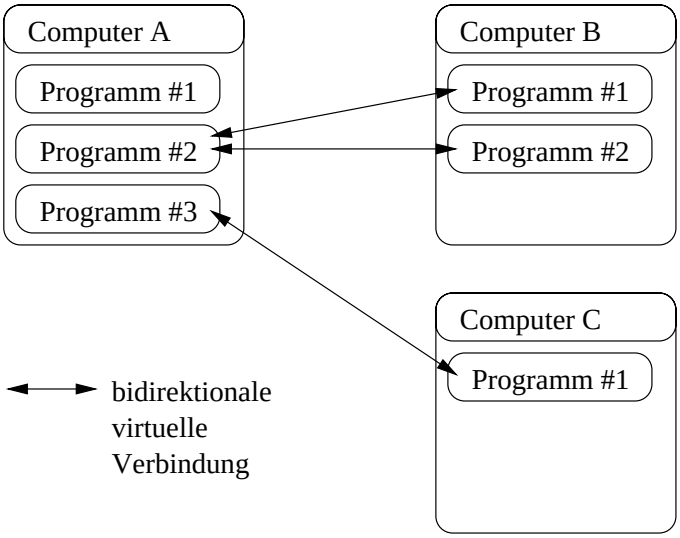

Abbildung 3.2: Virtuelle Verbindungen zwischen Computerprogrammen

Programme, die Dienste für andere Programme innerhalb des Netzwerkes erbringen möchten (Server), müssen sich im Netzwerk eindeutig bekannt machen. Basis für die Netzwerkkommunikation sind Kommunikationsendpunkte (engl. sockets). Virtuelle Verbindungen werden jeweils zwischen zwei Kommunikationsendpunkten aufgebaut. Um einen Kommunikationsendpunkt im Netzwerk bekannt zu machen, fordert das zugehörige Programm vom Betriebssystem für diesen Kommunikationsendpunkt eine computerlokale Kennung (engl. port; siehe Abb. 3.3) an. Ist eine computerlokale Kennung vom Betriebssystem vergeben, so kann diese Kennung so lange kein zweites Mal vergeben werden (Eindeutigkeit), wie der mit dieser Kennung assoziierte Kommunikationsendpunkt besteht. Klienten können zu Servern eine Verbindung aufbauen, wenn ihnen der Computername und die computerlokale Kennung eines Servers bekannt sind.

Da Netzwerkprotokolle nicht zwischen lokalen und entfernten Verbindungen unterscheiden, können sich Klient und Server auch auf demselben Computer befinden. Die Kommunikation zwischen Klient und Server ist somit verteilungstransparent.

3.2 Migrierende Objektgraphen

Objektsysteme können miteinander kooperieren. Die Kooperation von Objektsystemen besteht im Austausch von Objektgraphen (Migration). Ausgezeichnete

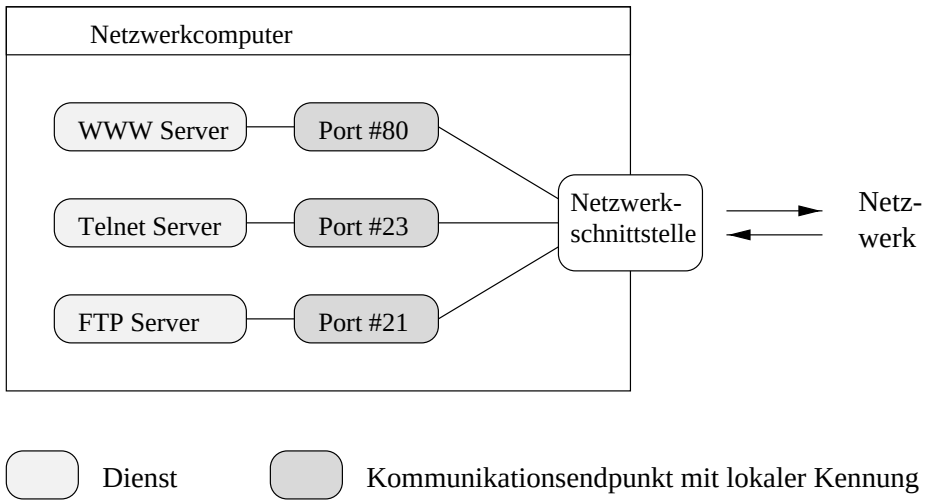

Abbildung 3.3: Kommunikationsendpunkte und Dienste

Objektsysteme können Objekte für andere Objektsysteme vorhalten, woraus eine Klient/Server Struktur zwischen Objektsystemen entsteht. Der Klient überträgt zum Server einen Objektgraphen, der den zu erbringenden Dienst beschreibt. Der Server überträgt dem Klienten anschließend einen, der Anfrage des Klienten entsprechenden, Objektgraphen.

Aktive Anwendungen (z.B. Textverarbeitung, Tabellenkalkulation) in Computersystemen bestehen aus Ausführungskontexten (Position im Programmcode, offene Dateien, ...), Daten (z.B. der in Bearbeitung befindliche Text) und Programmcode (siehe Abb. 3.4). Damit eine Anwendung migrieren kann, müssen alle drei Komponenten der Anwendung im Objektspeicher abgelegt sein, um dem Objektsystem zur Verfügung zu stehen.

Abbildung 3.5 zeigt eine vom Objektspeicher A in den Objektspeicher B migrierende Anwendung; die Anwendung ist als Objektgraph dargestellt.

Allgemein bestehen Klient und Server aus Objekten und tauschen Objekte bzw. Objektgraphen aus (siehe Abb. 3.6). Der Klient setzt sich aus verschiedenen Objekten zusammen, aus einem Ausführungskontext, Daten, einem Kommunikationsendpunkt sowie aus Funktionen für das Senden und Empfangen von Daten. Der Server besteht aus den gleichen Objekten wie der Klient, wobei der Kommunikationsendpunkt zusätzlich an eine rechnerlokale Kennung gebunden ist. Das Protokoll zwischen Klient und Server kann auch mehrere Kommunikationsphasen umfassen, z.B. wenn der Server den angeforderten Dienst nicht erbringen kann.

Die Kooperation von Objektsystemen ist nicht auf zwei Objektsysteme beschränkt. Ein Server kann z.B. wiederum Klient eines weiteren Servers werden, wenn er die Anfrage des Klienten nicht selbst beantworten kann. Abbildung 3.7

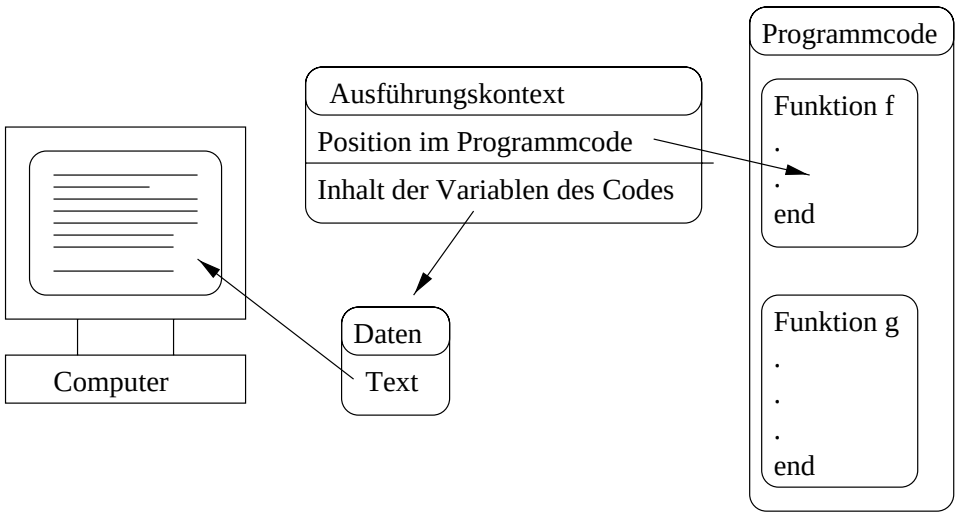

Abbildung 3.4: Textverarbeitung (Kontext, Daten, Code)

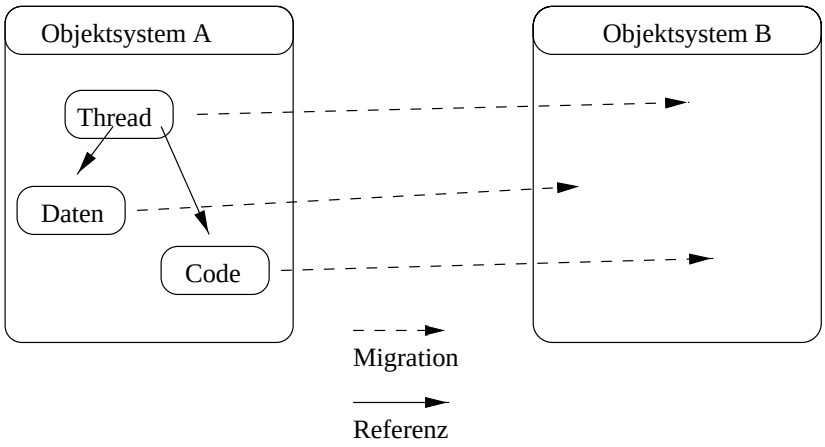

Abbildung 3.5: Migrierende Anwendung (als Objektgraph)

zeigt drei Objektsysteme die miteinander kooperieren. Das Objektsystem **A** (der Klient) stellt eine Anfrage an das Objektsystem **B** (den Server). Um das Ergebnis zu ermitteln, stellt **B** wiederum eine Anfrage an den Server **C**. Mit Hilfe des Ergebnisses der Anfrage an **C** kann **B** die Anfrage von **A** befriedigen.

3.2.1 Immobile Objekte

Objekte in Objektsystemen können semantisch (siehe Kapitel 2) an externe, also nicht im Objektspeicher abgelegte, Objekte gebunden sein (z.B. Kommunikationsendpunkt). Derartige Objekte werden als immobile bzw. potentiell immobile

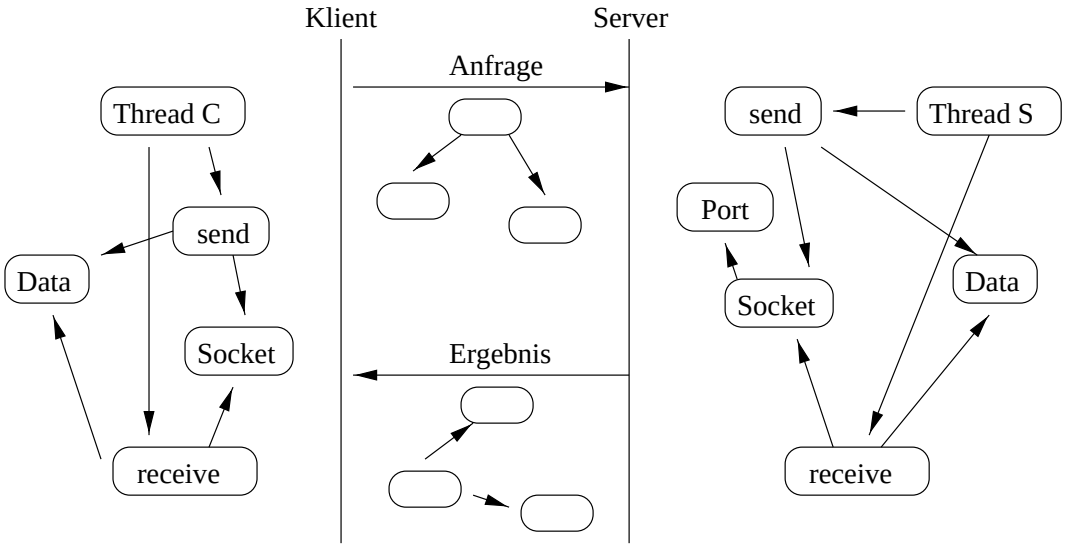

Abbildung 3.6: Kommunikation zwischen Klient und Server

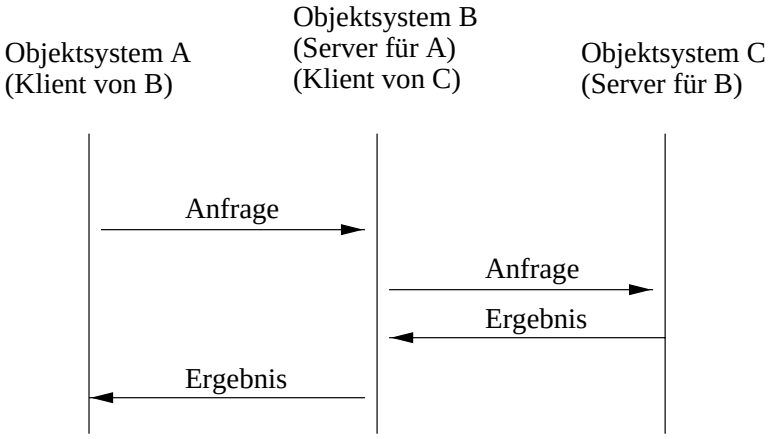

Abbildung 3.7: Rückfrage bei Kommunikation zwischen Klient und Server

Objekte bezeichnet, da bei einer Migration eines solchen Objektes die semantische Bindung, in Abhängigkeit vom Zielobjektsystem, verloren geht bzw. verloren gehen kann. Soll ein Objektgraph, welcher ein immobiles bzw. potentiell immobiles Objekt in seiner referentiellen transitiven Hülle aufweist, von einem Objektsystem in ein anderes Objektsystem migrieren (siehe Abb. 3.8), so muß sichergestellt werden, daß im Zielobjektsystem die semantische Bindung erneuert wird, bzw. daß das immobile Objekt nicht mit migriert. Verbleibt das immobile Objekt im Quellobjektsystem, so können Verweise des migrierenden Objektgraphen auf das immobile Objekt entweder in Netzwerkreferenzen umgewandelt oder invalidiert werden.

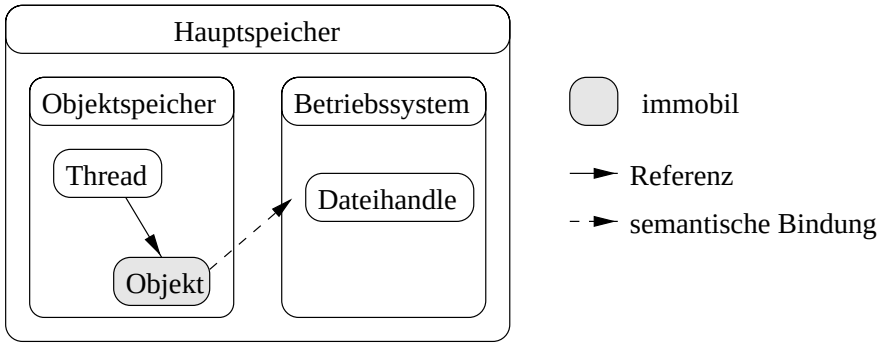

Abbildung 3.8: Immobiles Objekt

Migriert ein Objektgraph von einem Quellobjektsystem in ein Zielobjektsystem, und sollen semantische Bindungen immobiler bzw. potentiell immobiler migrierender Objekte im Zielobjektsystem wiederhergestellt werden, so ist für jedes immobile bzw. potentiell immobile Objekt vor der Migration eine Beschreibung der semantischen Bindungen zu erstellen. Nach der Migration wird mit Hilfe dieser Beschreibung im Zielobjektsystem eine zur ursprünglichen semantischen Bindung äquivalente semantische Bindung erzeugt.

Z.B. beinhaltet ein Objektgraph in seiner referentiellen transitiven Hülle einen Ausführungskontext und einen Dateiidentifikator (siehe Abb. 3.9). Der Dateiidentifikator ist ein immobiles Objekt, da er semantisch an ein Betriebssystemobjekt gebunden ist. Soll der Objektgraph aus dem Objektsystem A in das Objektsystem B migrieren, so muß sichergestellt werden, daß der Dateiidentifikator entweder nicht mit migriert, oder daß im Zielobjektsystem eine, zur ursprünglichen Bindung äquivalente, neue semantische Bindung etabliert wird. D.h. daß im Zielobjektsystem die gleiche Datei wie im Quellobjektsystem geöffnet und in denselben Zustand gebracht wird. Um im Zielobjektsystem eine neue Bindung erzeugen zu können, wird das immobile Objekt vor der Migration durch eine Beschreibung der semantischen Bindung ersetzt, bestehend aus Dateinamen, aktueller Dateiposition und Öffnungsmodus. Im Zielobjektsystem wird anschließend ein neues, semantisch gebundenes Objekt erzeugt, indem eine Datei desselben Namens im selben Modus geöffnet und der Dateipositionszeiger an der richtigen Stelle positioniert wird.

3.2.2 Transiente Objekte

Die Menge der transienten Objekte eines Objektgraphen, also jener Objekte, die nicht persistent gespeichert werden können, bilden eine Teilmenge der immobilen Objekte des Objektgraphen. Prinzipiell läßt sich die persistente Speicherung eines Objektgraphen als Migration betrachten. Quell- und Zielobjektsystem befinden

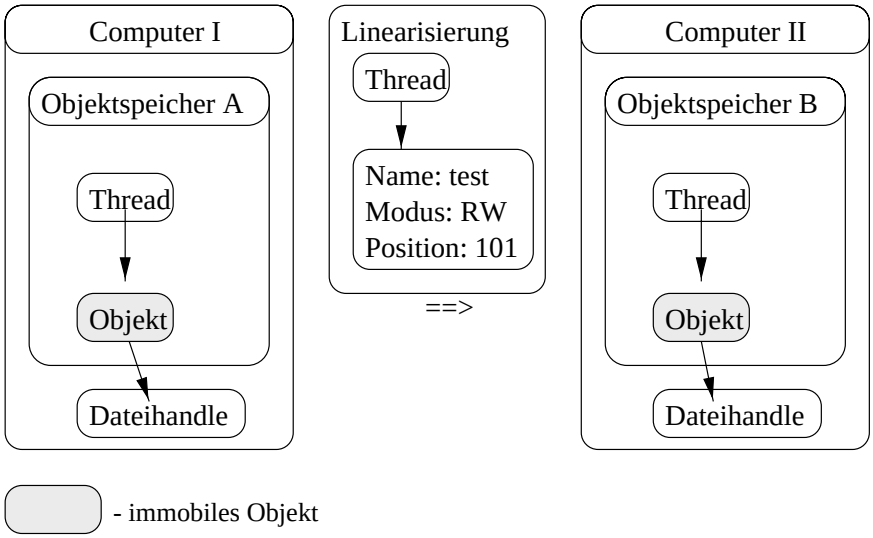

Abbildung 3.9: Migration eines ursprünglich immobilen Objektgraphen

sich dabei im selben Computersystem. Die Differenz der Menge der immobilen Objekte und der Menge der transienten Objekte setzt sich aus den potentiell immobilen Objekten zusammen, deren Immobilität vom Quell- und Zielcomputersystem abhängen.

Ein Objekt, welches einen Dateipfad enthält, ist beispielsweise ein potentiell immobiles Objekt. Gibt es im Zielcomputersystem keine Datei mit entsprechendem Pfad und ist dieses Objekt mit der Intension erzeugt worden, daß die beschriebene Datei immer vorhanden ist, so geht diese semantische Bindung zwischen Objekt und Datei bei einer Migration verloren. Migriert das Objekt allerdings zwischen Objektsystemen eines Computersystems (persistente Speicherung), so bleibt die Bindung erhalten.

Da die Behandlung immobiler Objekte die Behandlung transienter Objekte mit einschließt, brauchen transiente Objekte im weiteren nicht mehr betrachtet zu werden.

3.3 Rebindung bei Migration

Dynamische Rebindung erlaubt es, die Erzeugung notwendiger Beschreibungen für die Herstellung bzw. Wiederherstellung semantischer Bindungen immobiler bzw. potentiell immobiler Objekte migrierender Objektgraphen zu automatisieren. Dies geschieht, indem alle erzeugten immobilen Objekte registriert werden, anschließend ist es möglich, alle immobilen Objekte eines migrierenden Objektgraphen vor der Migration des Objektgraphen aus dem Graphen zu entfernen und

durch Beschreibungen der zugehörigen semantischen Bindungen zu ersetzen. Der Objektgraph wird um immobile Objekte reduziert. Im Zielobjektsystem werden diese Beschreibungen, wieder mittels dynamischer Rebindung, durch zu denen im Quellobjektsystem äquivalente Objekte ersetzt, der Objektgraph wird expandiert.

Durch Reduktion des Objektgraphen vor der Migration bzw. Expansion des Objektgraphen nach der Migration, ist es prinzipiell möglich, für jedes Objekt die Art der Migration zu beeinflussen. Beispielsweise kann verhindert werden, das ubiquitäre Objekte oder Objekte mit Programmcode migrieren. Verschiedene Arten von Objekten, deren Migration beeinflußt werden soll, wie z.B. ubiquitäre Objekte, Programmobjekte, Dateiidentifikatoren oder Kommunikationsendpunkte können identifiziert werden (siehe Kapitel 4).

3.4 Die Netzwerkprogrammierung unter Tycoon

Externe Dienste werden in das TYCOON System mittels dynamisch gebundener Bibliotheken integriert. Die auf niedriger Ebene angesiedelten Kommunikationsmechanismen des das TYCOON System umgebenden Betriebssystems stehen auf diese Weise dem TYCOON-Programmierer zur Verfügung.

3.4.1 Kommunikationsendpunkte

Folgende Basisfunktionalität für die Manipulation von Kommunikationsendpunkten ist für die Programmierung verfügbar:

Erzeugung: Kommunikationsendpunkte können dynamisch erzeugt werden, wobei die Art der Datenübertragung (verbindungslos / verbindungsorientiert) und der Protokolltyp anzugeben (TCP/IP, IPX/SPX, ...) sind.

Freigabe: Kommunikationsendpunkte müssen explizit freigegeben werden, da sonst die belegten Ressourcen (computerlokale Kennung, ...) belegt bleiben. Insbesondere kann es vorkommen, daß ein Kommunikationsendpunkt von keiner Anwendung mehr erreicht werden kann, die Ressourcen aber nicht freigegeben werden.

Verbindung: Ein Kommunikationsendpunkt kann mit einem zweiten, benannten Kommunikationsendpunkt verbunden werden. Anschließend können Daten ausgetauscht werden.

Anforderung: Ein Server, der einen benannten Kommunikationsendpunkt erzeugt hat, kann, durch Anforderung einer anonymen Verbindung, auf den Verbindungsaufbau eines Klienten warten.

Bekanntmachung: Kommunikationsendpunkte können im Netzwerk bekannt gemacht werden.

Senden und empfangen: Es können Daten vom Kommunikationspartner empfangen bzw. zum Kommunikationspartner übertragen werden.

Das TYCOON System stellt eine Reihe von Mechanismen zur Verfügung, die es erlauben, Objektgraphen zwischen Objektsystemen heterogener Plattformen auszutauschen. Folgende Punkte müssen dabei beachtet werden:

- Es wird eine Schnittstelle zu den Betriebssystemfunktionen für die Netzwerkprogrammierung bzw. für die Ein-/Ausgabeprogrammierung benötigt.
- Objekte müssen in eine plattformunabhängige Darstellung transferiert werden können (Marshalling).
- Objektgraphen müssen linearisiert und wieder delinearsiert werden können.

Da prinzipiell keine Typsicherheit beim Austausch von Objekten zwischen Objektsystemen garantiert werden kann, wurde für Objekte der TYCOON Systeme eine dynamische Typisierung eingeführt. Das Modul *Dynamic* der TYCOON Standardbibliothek erlaubt die Erzeugung dynamisch getypter Objekte. Ein derart dynamisch getyptes Objekt kann typsicher zwischen Objektsystemen ausgetauscht werden. Das Modul *Dynamic* bietet die Funktionalität, das dynamisch getypte Objekt im Zielobjektspeicher wieder in ein statisch getyptes Objekt zu konvertieren.

Das Objektspeicherinterface (TSP) des TYCOON Systems stellt Funktionen für die Linearisierung und das Marshalling von Objektgraphen zur Verfügung. Das Modul *Dynamic* der TYCOON Standardbibliothek kapselt diese Funktionalität in den Funktionen *extern* und *intern*. Die Funktion *extern* erlaubt es, dynamisch getypte Objektgraphen linearisiert und in standardisierter Darstellung auf Ausgabegeräten auszugeben. Mit der Funktion *intern* werden liniarisierte und gemarshallte Objektgraphen aus Eingabegeräten eingelesen.

Im Rahmen der Dissertation [Mat96] wurde unter anderem ein Modul für die Netzwerkprogrammierung implementiert. Das Modul *Socket* enthält Funktionen für die Belegung und Verwendung von Kommunikationsendpunkten. Das Modul stellt auch Funktionen für die Auflösung symbolischer Rechnernamen zur Verfügung. Kommunikationsendpunkte können in Ein- und Ausgabgeräte der Standardbibliothek umgewandelt werden. Zusammen mit den Funktionen des Moduls *Dynamic* ist es daher möglich, Objektgraphen zwischen heterogenen Objektsystemen auszutauschen.

3.4.2 Der Portmapper

Wie in Abschnitt 3.1 beschrieben, kann ein Computer mehrere Netzwerkdienste gleichzeitig anbieten. Ein Objektsystem, welches aktiv auf einem Computer ist und welches Nebenläufigkeit unterstützt, kann ebenfalls mehrere Netzwerkdienste gleichzeitig anbieten.

Jede rechnerlokale Kennung eines Computersystemes ist eindeutig, d.h. ist ein Kommunikationsendpunkt mit einer Kennung assoziiert, so wird ein Klient, der eine Verbindung zu diesem Kommunikationsendpunkt aufbaut, den zum Kommunikationsendpunkt zugehörigen Dienst aktivieren. Soll nun eine Vielzahl von Diensten von einem Computer angeboten werden, so muß jedem dieser Dienste eine rechnerlokale Kennung zugewiesen werden. Ist jederzeit bekannt, welche Dienste erbracht werden sollen, so kann diesen Diensten statisch eine Kennung zugewiesen werden, d.h. die zugewiesene Kennung ist auch dann reserviert, wenn der Dienst nicht erbracht wird (Beispiele für eine derartige statische Zuweisung sind die Internetdienste FTP, Telnet und WWW). Werden Dienste dynamisch angeboten, werden also diensterbringende Programme beliebig gestartet und gestoppt, so ist es nicht sinnvoll, diesen Diensten statisch Kennungen zuzuweisen, da es z.B. mehr verschiedene Dienste als Kennungen geben kann. Dies ist insbesondere bei Diensten möglich, die mehrfach angeboten werden können. Sollten diese Dienste statisch Kennungen zugewiesen bekommen, so würde im vorhinein die maximale Anzahl von Diensten eines Diensttyps, die erbracht werden sollen, festgelegt werden, was eine unnötige Einschränkung darstellt.

Ein Portmapper löst dieses Problem, indem jedem Dienst ein beliebiger, kennungsunabhängiger, eindeutiger Name zugeordnet werden kann, d.h. die resultierende Kennung eines Dienstes (sein Name), wird nicht mehr vom Betriebssystem vergeben, sondern beliebig erzeugt, wodurch insbesondere die Auswahl des Namensraumes vom Betriebssystem unabhängig wird.

Aufgabe des Portmappers ist es, jedem eindeutig benannten Dienst, der angemeldet wird, eine rechnerlokale Kennung zuzuweisen. Um diesen Dienst ansprechen zu können, muß es der Portmapper auch erlauben, die zu einem Dienstnamen gehörige rechnerlokale Kennung abzufragen. Um diese Aufgaben zu erfüllen, führt der Portmapper eine Tabelle mit Dienstnamen und rechnerlokalen Kennungen. Der Portmapper wiederum ist selbst ein Dienst, dessen Kennung statisch vergeben wird.

Will ein Klient zu einem Server eine Verbindung aufbauen, so stellt er zunächst eine Verbindung zu dem auf dem Zielrechner laufenden Portmapper her. Anschließend stellt der Klient beim Portmapper eine Anfrage nach der rechnerlokalen Kennung des Servers. Mit Hilfe dieser Kennung baut der Klient daraufhin eine Verbindung zum gewünschten Server auf.

Sollen Dienste eines Objektsystems Klient-Programmen zur Verfügung gestellt

werden, so ist der Name des Dienstes beim Portmapper anzumelden. Als Ergebnis der Anmeldung gibt der Portmapper eine rechnerlokale Kennung zurück. Der zum Dienst zugehörige Kommunikationsendpunkt wird anschließend an diese rechnerlokale Kennung gebunden, wodurch der Dienst im Netzwerk sichtbar wird.

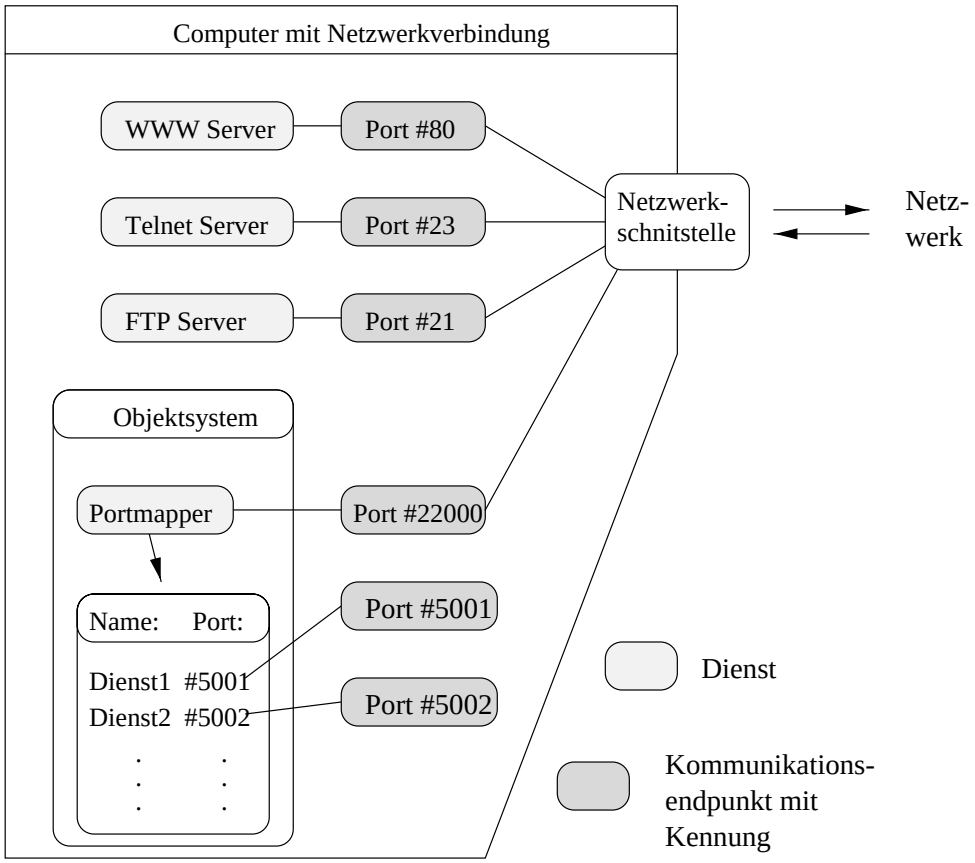

Abbildung 3.10: Portmapper im Objektsystem im Computersystem

Die Schnittstelle des Portmappermoduls gliedert sich in zwei Kategorien (siehe Anhang A). Die Funktionen der ersten Kategorie dienen der Administration eines Portmappers und die Funktionen der zweiten Kategorie erlauben einen transparenten Zugriff auf einen im System aktiven Portmapper.

Der Portmapper unterstützt folgende Kommandos:

GET: Bei der Übertragung dieses Kommandos sowie eines Dienstnamens gibt der Portmapper die zum Dienst gehörige rechnerlokale Kennung zurück.

SET: Das Kommando hat als Parameter einen anzumeldenden Dienstnamen. Das Ergebnis der Anmeldung ist eine rechnerlokale Kennung. Ist der Dienstname bereits in der Tabelle des Portmappers gespeichert, so wird eine neue

rechnerlokale Kennung erzeugt, der Eintrag in der Tabelle aktualisiert, und diese Kennung zurückgegeben. Ist kein Portmapper aktiv, so wird zunächst ein Portmapper im aktuellen Objektsystem gestartet.

SUB: Dieses Kommando erwartet als Parameter einen Dienstnamen. Der Dienstname sowie die mit dem Dienstnamen assoziierte lokale Kennung werden aus der Tabelle des Portmappers entfernt. Ist die Tabelle anschließend leer, so beendet der Portmapper seine Ausführung.

DIR: Dieses Kommando hat als Ergebnis eine Liste aller lokal verfügbaren Dienste.

QUIT: Der Portmapper beendet seine Ausführung.

Dadurch, daß ein Klient die Menge der auf einem Rechner zur Verfügung stehenden Dienste einsehen kann, kann der Klient sich einen der angebotenen Dienste aussuchen. Dies ist z.B. dann von Vorteil, wenn Dienste gleicher Funktionalität, aber unterschiedlicher Qualität, angeboten werden.

3.4.3 Generische Klient/Server Programmierung

Da die Kommunikation zwischen Klient und Server meist im wechselseitigen Austausch von Daten besteht, ist es sinnvoll, von den Elementen der Netzwerkkommunikation (Kommunikationsendpunkt, lokale Kennung) zu abstrahieren, indem das Augenmerk nur auf die auszutauschenden Daten gelegt wird. Jeder Server und jeder Klient stellen dabei eine Kommunikationsfunktion zur Verfügung, welche Anfragen sendet bzw. empfängt und Antworten empfängt bzw. sendet.

Bei Servern kann unterschieden werden zwischen solchen, die nur einen Dienst für einen Klienten zur Zeit erbringen können, und solchen, die prinzipiell beliebig viele Anfragen gleichzeitig bearbeiten können. Letztere reagieren auf einen Verbindungswunsch mit der Erzeugung eines neuen Ausführungskontextes (engl. thread), welcher entweder die Anfrage des Klienten entgegennimmt oder aber auf weitere Verbindungswünsche reagiert.

Da bei den meisten Dienstanforderungen durch Klienten diese Anforderungen unabhängig voneinander sind und weil der Durchsatz des Servers dadurch gesteigert werden kann, ist es sinnvoll, Server generell so zu entwickeln, daß sie mehrere Anfragen gleichzeitig bearbeiten können.

Nach Verbindungsaufbau sendet der Klient eine Anfrage an den Server, welche die Parameter für den zu erbringenden Dienst enthält. Der Server bearbeitet die Anfrage und sendet das Ergebnis an den Klient zurück. Sollte der Klient mit dem Ergebnis der Anfrage nicht zufrieden sein, oder sollte der Klient weitere Anfragen stellen wollen, so kann sich dieser Zyklus beliebig oft wiederholen. Anschließend

geben Server und Klient ihre Kommunikationsendpunkte wieder frei und die Verbindung wird abgebaut.

Werden Dienste unter Verwendung eines Portmappers im System angemeldet, so geht dem Verbindungsaufbau von Klient zu Server noch ein Verbindungsaufbau zum Portmapper voraus (siehe Abschnitt 3.4.2).

Prinzipiell wird folgende Funktionalität für die Implementation von Klient und Server benötigt:

Erzeugung eines Servers: Ein Server kann mit Hilfe einer Kommunikationsfunktion und eines Dienstnamens erzeugt werden.

Beendigung eines Servers: Ein Server kann terminiert werden.

Kontaktieren eines Servers: Ein Klient kann zu einem benannten Server eine Verbindung aufbauen.

Kommunikation zwischen Klient und Server: Eine einfache Kommunikationsfunktion kann den Datenaustausch Klient und Server koordinieren. Diese Funktion stellt eine benutzerdefinierte Anfrage und empfängt ein benutzerdefiniertes Ergebnis.

Da während dieser Arbeit mehrere Paare von Klienten und Servern implementiert wurden, wurde ein abstrahierendes Modul für die Kommunikation zwischen Klient und Server entwickelt. Die Ermittlung der rechnerlokalen Kennung eines Servers erfolgt über den Umweg eines Portmappers, Dienstnamen werden als Zeichenketten angegeben. Die Schnittstelle des Moduls ist im Anhang angegeben (siehe Anhang A).

Kapitel 4

Rebindungsobjekte

Dynamische Rebindung erlaubt es, Objekte in Objektgraphen auszutauschen. Wie bereits in Kapitel 3 beschrieben, ist es möglich, mit Hilfe dynamischer Rebindung das Problem der immobilen bzw. transienten Objekte kooperativer persistenter Objektsysteme zu lösen. Auch Aktualisierungsprobleme langlebiger Prozesse können gelöst werden. Mit Hilfe eines Rebindungsmanagers (siehe Kapitel 5) kann individuell für jedes Objekt eines Objektgraphen für die Migrationsreduktion und für Migrationsexpansion eine Transformationsfunktion angegeben werden. Es werden verschiedene Kategorien von Objekten betrachtet, für die es sinnvoll bzw. notwendig ist, Transformationsfunktionen anzugeben.

Prinzipiell kann unterschieden werden zwischen folgenden Objektkategorien:

Programmobjekte: Mit dem Begriff Programmobjekt werden all jene Objekte bezeichnet, die von einem Übersetzer auf dem Weg vom Quelltext zum ausführbaren Code erzeugt werden. Objekte, die Programmtext bzw. Syntaxbäume oder ausführbaren Code enthalten, sind Programmobjekte.

Immobile bzw. transiente Objekte: Objekte mit einer semantischen Bindung an externe Objekte (z.B. einem Dateiidentifikator oder einer Datei mittels des Dateipfades und des Dateinamens) sind prinzipiell nicht oder nur eingeschränkt persistent bzw. mobil (siehe Kapitel 3).

Ubiquitäre Objekte: Objekte, die in jedem Objektsystem vorhanden sind, werden als ubiquitär bezeichnet. Da dies eine semantische Eigenschaft ist, müssen derartige Objekte explizit als ubiquitär markiert werden. Ubiquitäre Objekte brauchen prinzipiell nicht zu migrieren bzw. gespeichert zu werden, da sie immer vorhanden sind.

4.1 Programmobjekte

Übersetzer erzeugen aus Quelltexten ausführbare Programme. Das Übersetzen geschieht meist nicht in einem Schritt, sondern in mehreren Phasen. Abbildung 4.1 zeigt die Phasen des Übersetzens nach [ASU88]. Alle Objekte, die in ausführbaren Code übersetzt werden können, sind Programmobjekte. Ein Objekt, daß einen Syntaxbaum und die zugehörige Symboltabelle zumsammenfaßt, ist z.B. ein Programmobjekt. Derartige Objekte werden nur, nach dem sie hauptsächlich auszeichnenden Attribut benannt, in diesem Fall ist das entsprechende Objekt ein Syntaxbaum-Programmobjekt.

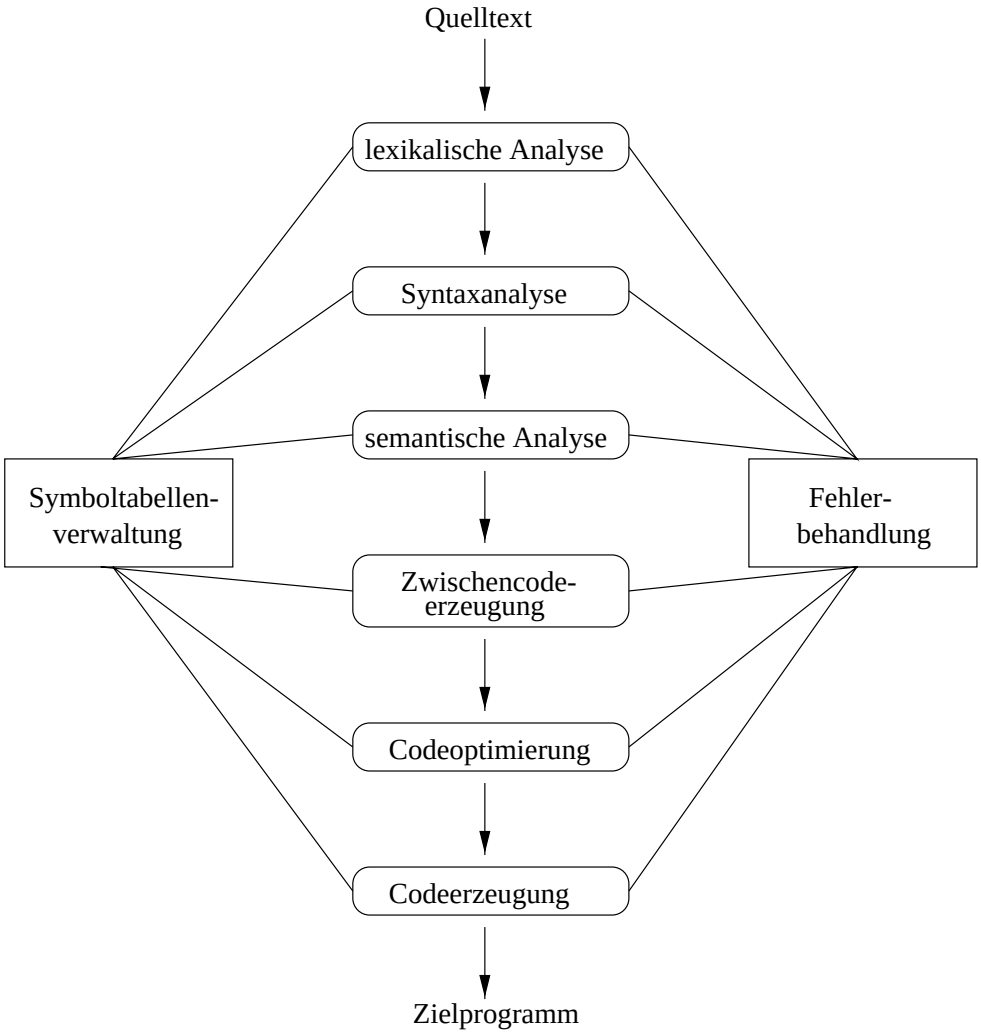

Abbildung 4.1: Phasen des Übersetzens

Programmobjekte können in untschiedlichen Projekten und in verschiedenen Ver-

sionen vorkommen. Alle Versionen, beispielsweise eines Textes, werden so in Archiven zusammengefaßt, daß die Ableitungsbeziehungen zwischen einzelnen Versionen erhalten bleiben. Ein Projekt verwaltet eine Menge von Objekten, die zu einem Produkt zusammengebaut werden. Projekte in der Softwareentwicklung enthalten z.B. alle notwendigen Archive der einzelnen Softwaremodule. Bei der Veröffentlichung werden die Objektdateien der einzelnen Module dann zu einem Programm zusammengebunden.

Abbildung 4.2 zeigt die Orthogonalität von Versionierung, Projektverwaltung und Übersetzung.

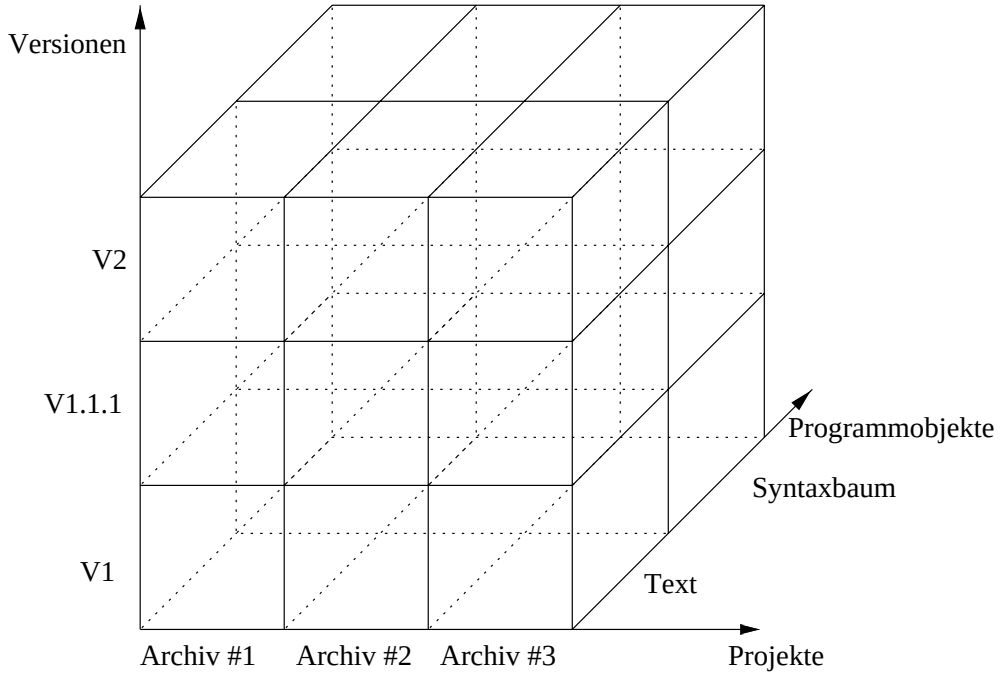

Abbildung 4.2: Orthogonalität von Projektverwaltung, Archivierung und Programmobjekten

4.1.1 Reduktion migrierender Objektgraphen um Programmobjekte

Migrierende Objektgraphen werden aus folgenden Gründen um Programmobjekte reduziert, wobei Referenzen auf Programmobjekte durch Platzhalter ersetzt werden.

- Programmobjekte werden aus Sicherheitsgründen nicht übertragen. Gewisse Instruktionen ausführbaren Codes erlauben z.B. das Ausführen nicht

erlaubter Operationen auf der Zielmaschine. Werden migrierende Objektgraphen um ausführbaren Code reduziert, so können solche Operationen im Zielobjektsystem nicht zur Ausführung gebracht werden.

- Programmobjekte werden aus Performanzgründen nicht übertragen. Viele Bibliotheken mit Programmcode werden von verschiedenen Anwendungen verwendet. Befindet sich im Zielobjektsystem z.B. Programmcode mit gleicher Funktionalität wie im migrierenden Objektgraphen, so braucht nicht der gesamte Objektgraph zu migrieren, sondern nur jene Teile des Objektgraphen, die neue Funktionalität zur Verfügung stellen. Es werden insgesamt also weniger Daten übertragen, was die Netzlast sowie die Übertragungszeit reduziert.
- Programmobjekte werden nicht übertragen, weil sie nicht portabel sind, also nativen Code enthalten. Der Übersetzer im Quellobjektsystem erzeugt ausführbaren Code, der nur vom Prozessor des Quellobjektsystems ausgeführt werden kann. Die Übertragung dieses Codes ist nicht sinnvoll.

Migriert ein Objektgraph von einem Objektsystem in ein anderes, so kann er um Programmobjekte reduziert werden. Die Programmobjekte werden durch Platzhalter ersetzt, die diese Objekte beschreiben. Eine Beschreibung eines Programmobjektes enthält z.B. einen eindeutigen Bezeichner für den Quelltext, die Version sowie eventuell weitere quelltextabhängige Informationen, wie z.B. den Namen des Moduls, in dem das Programmobjekt definiert ist. Abbildung 4.3 zeigt einen Objektgraphen, in dem ein Programmobjekt durch einen Platzhalter ersetzt wird.

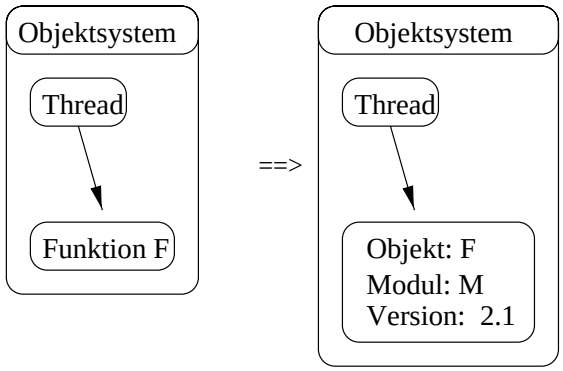

Abbildung 4.3: Ersetzen eines Programmobjektes durch einen Platzhalter

Im Zielobjektsystem werden derartige symbolische Referenzen durch Referenzen auf lokal vorhandene äquivalente Programmobjekte ersetzt. Um nun nicht in jedem Objektsystem alle Programmobjekte in allen Versionen und Zuständen vorhalten zu müssen, ist es notwendig, die Verwaltung der Programmobjekte in das Objektspeichersystem zu integrieren. Programmobjekte können dann bei Bedarf repliziert und abgeleitete Programmobjekte auch erzeugt werden.

4.1.2 Die Projektverwaltung

Große Softwaresysteme werden nicht von einzelnen Programmierern, sondern von Gruppen von Programmierern entwickelt. Mehrere Programmierer bearbeiten, je nach Aufgabenbereich, das zu entwickelnde Softwaresystem gleichzeitig. Die Arbeit der Programmierer muß koordiniert werden, welches die Aufgabe einer Projektverwaltung ist. Eine Projektverwaltung erlaubt es einer Anzahl von Entwicklern, gleichzeitig koordiniert an einem Projekt zu arbeiten.

Die Basis einer Projektverwaltung ist der Arbeitsbereich. Innerhalb eines Arbeitsbereichs kann ein Softwareentwickler Programmobjekte verändern, ableiten oder neu entwickeln. Änderungen, die an Programmobjekten innerhalb eines Arbeitsbereiches gemacht werden, sind außerhalb des Arbeitsbereiches nicht sichtbar. Arbeitsbereiche sind meist baumartig, hierarchisch organisiert, daraus folgt, daß alle Arbeitsbereiche, bis auf den Wurzelarbeitsbereich, genau einen Elternarbeitsbereich haben. Jeder Arbeitsbereich kann beliebig viele Kindarbeitsbereiche haben (siehe Abb. 4.4).

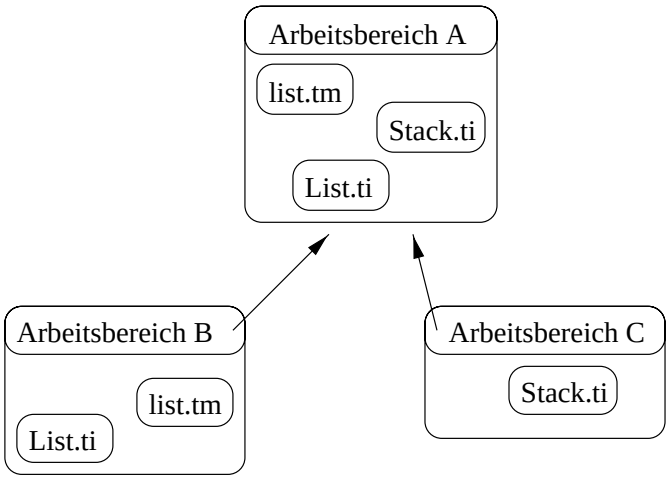

Abbildung 4.4: Hierachie von Arbeitsbereichen

Arbeitsbereiche kooperieren entlang ihrer Hierarchien, d.h. es kooperieren jeweils Kind- und Elternarbeitsbereiche miteinander. Es gibt verschiedene Kooperationsmodelle für Arbeitsbereiche, die nach folgenden Kriterien klassifiziert werden können:

Speicherbasis: Es kann unterschieden werden zwischen dateibasierten und datenbankbasierten Kooperationsmodellen. Der grundlegende Unterschied ist, daß Dateien, im Gegensatz zu Objekten einer Datenbank, nicht getypt und prinzipiell jedem Anwendungsprogramm zugänglich sind. Wird verteilt auf mehreren Computern entwickelt, so ist außerdem noch ein Netzwerkdatei-

system notwendig.

Nebenläufigkeit: Ist das gleichzeitige Ändern derselben Programmobjekte durch mehrere Softwareentwickler in mehreren Arbeitsbereichen möglich, so können beim Zurückschreiben der Programmobjekte in den Elternarbeitsbereich Konflikte auftreten. Wurde z.B. Programmobjekt A von den Entwicklern 1 und 2 verändert, und hat Entwickler 1 seine Änderungen am Programmobjekt A in den Elternarbeitsbereich zurückgeschrieben, so kann Entwickler 2 seine Änderungen nicht zurückschreiben, ohne daß die Änderungen von Entwickler 1 verloren gehen.

Hinterlegung (engl. backing): Kann der Inhalt eines Arbeitsbereiches innerhalb seiner Kindarbeitsbereiche automatisch sichtbargemacht werden, so wird dies als Hinterlegung bezeichnet.

Folgende vier Kooperationsmodelle von Projektverwaltungen ergeben sich aus den letzten beiden Kriterien:

Kopieren-Ändern-Zurückschreiben-Modell: Diese Variante erlaubt das exklusive Bearbeiten von Programmobjekten. Änderungen von Programmobjekten im Elternarbeitsbereich werden erst durch explizites Kopieren der Programmobjekte vom Elternarbeitsbereich in den Kindarbeitsbereich sichtbar.

Kopieren-Ändern-Vereinigen-Modell: Innerhalb dieses Modells ist das gleichzeitige Bearbeiten von Programmobjekten unterschiedlicher Arbeitsbereiche möglich. Aktualisierungen eines Elternarbeitsbereiches werden erst durch Kopieren der betroffenen Objekte in den Kindarbeitsbereich sichtbar. Dieses Modell ist das Kooperationsmodell der, von der Firma Sun Microsystems vertriebenen, Projektverwaltung SPARCWorks/TeamWare.

Ändern-Zurückschreiben-Modell: Dieses Modell erlaubt einem Entwickler zur Zeit das Verändern von Programmobjekten. In einen Elternarbeitsbereich zurückgeschriebene Änderungen werden sofort in den Kindarbeitsbereichen sichtbar.

Ändern-Vereinigen-Modell: Diese Variante erlaubt das nebenläufige Ändern von Programmobjekten. Änderungen eines Elternarbeitsbereiches werden sofort in allen Kindarbeitsbereichen sichtbar. Der von Bolinger und Bronson entwickelten Projektverwaltung TCCS[1], welche sie in Ihrem Buch [BB95] beschreiben, liegt dieses Kooperationsmodell zugrunde.

Der Vorteil der kopierenden Modelle ist ihre Unabhängigkeit, d.h. ein Softwareentwickler kann auch dann arbeiten, wenn er eine Zeitlang keinen Zugriff auf den

[1] Trivial Configuration Control System

Elternarbeitsbereich hat. Nachteile dieser Modelle sind mögliche Konsistenzprobleme sowie ein größerer Speicherplatzverbrauch.

4.1.2.1 TCCS

Das von Bolinger und Bronson entwickelte System TCCS setzt wahlweise auf SCCS[2] oder RCS[3] auf. TCCS unterstützt die Arbeit einer oder mehrerer Gruppen von Entwicklern sowie die Entwicklung für mehrere Plattformen. Weiterhin gibt es eine Unterstützung für einen Release-Prozeß. TCCS basiert auf dem Arbeitsbereichkonzept, wobei es noch weitere Strukturen gibt. Korrekt eingesetzt, erlaubt es TCCS, ohne Vorhaltung der Binärobjekte für jeden markierten Projektzustand (z.B. Release 1.0, Release 1.1, Release 2.0) diesen exakt wiederherzustellen. TCCS ist dateibasiert und strukturiert ein Projekt mit Hilfe von Verzeichnisstrukturen. TCCS erlaubt es mehreren Programmierern, gleichzeitig ein Projektobjekt zu verändern. Diese Änderungen eines Projektobjektes werden anschließend mit einem Verschmelzungsprogramm in ein resultierendes Projektobjekt überführt.

4.1.2.2 Projektverwaltung im Tycoon-System

Im Rahmen dieser Arbeit wurde eine Projektverwaltung für das TYCOON System entwickelt. Nachfolgend werden die Entscheidungen bezüglich des Kooperationsmodells dieser Projektverwaltung erörtert.

Die entwickelte Projektverwaltung ist nicht dateibasiert, sondern verwaltet die Entitäten als Objekte. Durch die Persistenz des umgebenden Systems ist sichergestellt, daß die Daten permanent gespeichert werden. Sollten die Daten innerhalb eines Dateisystems sichtbargemacht werden, so wäre dies durch zusätzliche Funktionalität zu realisisren.

Programmobjekte werden nicht von Elternarbeitsbereichen in Kindarbeitsbereiche kopiert, um Konsistenzprobleme zu vermeiden, sondern werden durch Hinterlegung automatisch sichtbar (dies betont unter anderem den Kooperationsaspekt zwischen Objektsystemen).

Programmobjekte unterschiedlicher Arbeitsbereiche können gleichzeitig verändert werden. Es ist oft notwendig, daß Softwareentwickler gleichzeitig an demselben Programmobjekt arbeiten.

Das Kooperationsmodell der entwickelten Projektverwaltung ist ein objektbasiertes Ändern-Vereinigen-Modell. Implementiert ist es auf Basis des Klient/Server Konzeptes. Jeder Arbeitsbereich einer Arbeitsbereichshierarchie kann sich innerhalb eines anderen Objektsystems befinden. Die einzelnen Arbeitsbereiche sind

[2]Source Code Control System
[3]Resource Control System

nur lose miteinander gekoppelt, so daß keine kontinuierliche Netzwerkverbindung aufrechterhalten werden muß. Programmobjekte werden bei Bedarf über das Netzwerk von einem Arbeitsbereich zu einem anderen übertragen.

Die entwickelte Projektverwaltung erlaubt in der aktuellen Version kein Einfrieren von Zuständen (engl. snapshot). Es wird also kein Release-Prozeß unterstützt (vgl. [BB95]).

4.1.3 Versionierung

Objekte in Computersystemen unterliegen einer ständigen Folge von Änderungen. Jede Änderung eines Datums erzeugt eine neue Version des urspünglichen Datums, wobei der ursprüngliche Inhalt überschrieben wird und somit verloren geht. Gerade in Nicht-Standard-Anwendungen (vgl. [STS97] und [BB95]) wie CASE[4] oder CSCW ist die historische Entwicklung von Datenobjekten von Interesse. Insbesondere das Experimentieren mit verschiedenen Versionen eines Objektes spielt bei diesen Anwendungen eine wichtige Rolle.

Aufgabe eines Versionierungssystems ist es, verschiedene Versionen von Daten zu speichern und auf Anforderung wieder verfügbar zu machen. Ein Datum, das durch eine Änderung aus einem anderen Datum hervorgegangen ist, ist von diesem Datum abgeleitet. Ausgehend vom urspünglichen Datum sind alle folgenden Versionen des Datums von diesem abgeleitet. Alle Versionen eines Datums stehen zueinander in Beziehung, wobei sich diese Beziehungen als gerichteter azyklischer Graph darstellen lassen (siehe Abb. 4.5)

Die erste Version eines Datums wird als Nummer Eins bezeichnet. Die zweite Version als Nummer Zwei und so weiter. Wird eine Version des Datums von einem Vorgänger abgeleitet, der schon einen Nachfolger hat, so wird diese Version als $X.Y.Z$ bezeichnet, wobei X die Nummer des Vorgängers benennt. Y steht für die Nummer der Ableitung minus eins. Ist diese Version also der zweite Nachfolger des Vorgängers, so steht Y für die Eins. Z beschreibt die Versionen der Nachfolger innerhalb dieses Nachfolgerzweiges. Für jeden folgenden ersten Nachfolger dieser Version, wird Z um eins erhöht. Dieses Benennungsschema wird rekursiv angewendet.

4.1.3.1 Versionsgraphen

Versionsgraphen sind ein Hilfsmittel, um Beziehungen zwischen voneinander abgeleiteten Objekten zu analysieren. Abbildung 4.5 ist ein Beispiel für einen Versionsgraphen. Knoten innerhalb eines Versionsgraphen stehen für Objekte einer bestimmten Version. Pfeile markieren den Weg der Entwicklung. Jedes Objekt,

[4] Computer Aided Software Engineering

bis auf das Wurzelobjekt (hier Objekt 1), ist abgeleitet von einem oder mehreren Vorgängerobjekten. Objekte, die mehr als einen Vorgänger haben (z.B. die Objekte 4 und 2.1.2), sind das Ergebnis der Verschmelzung (engl. merge) dieser Vorgänger (Objekte 4 und 2.1.2 sind das Ergebnis zweier verschiedener Verschmelzungen der Objekte 2.1.1 und 3), plus eventueller, zusätzlicher Änderungen. Um ein nicht qualifiziertes Hinzufügen von Objekten, also das Hinzufügen von Objekten ohne Angabe des oder der Vorgänger, zu ermöglichen, gibt es in jedem Versionsgraphen ein ausgezeichnetes Objekt (hier Objekt 4), das Standardbasisobjekt (in der Abb. weiß markiert). Wird ein Objekt zum Versionsgraphen hinzugefügt, ohne daß der Vorgänger spezifiziert wird, so erhält dieses Objekt das Standardbasisobjekt als Vorgänger und wird selbst zum neuen Standardbasisobjekt.

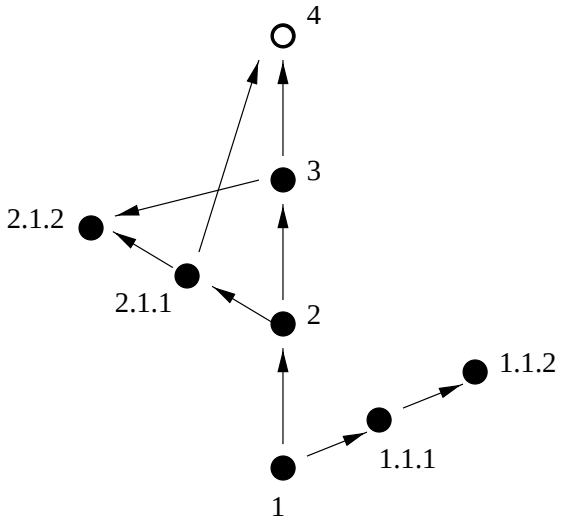

Abbildung 4.5: Versionsgraph

Versionsgraphen können miteinander in Bezug gebracht werden. Ein Versionsgraph wird als abgeleitet von einem zweiten Versiongsgraphen bezeichnet, wenn alle Knoten des Ursprungsgraphen im abgeleiteten Graphen mit gleichen Knoten wie im Ursprungsgraphen in Relation stehen und den gleichen Inhalt haben (siehe Abb. 4.6). Versionsgraph B ist aus Versionsgraph A durch nicht qualifiziertes Hinzufügen eines neuen Objektes (Objekt 3) entstanden. Die Versionsgraphen A und B stehen in der Relation B *ist-abgeleitet-von* A.

Versionsgraphen können miteinander verschmolzen werden (siehe Abb. 4.7). Sollen zwei Versionsgraphen A und B miteinander verschmolzen werden, so wird einer der beiden Graphen als Basis für den resultierenden Versionsgraphen R genommen, z.B. Versionsgraph A. Die Unterschiede des Graphen B zum Graphen A werden in den neuen Graphen eingefügt. Der resultierende Versionsgraph R enthält nun alle Objekte der beiden Ursprungsgraphen. Besitzen die zu verschmelzenden Versionsgraphen unterschiedliche Standardbasisobjekte, so wird die

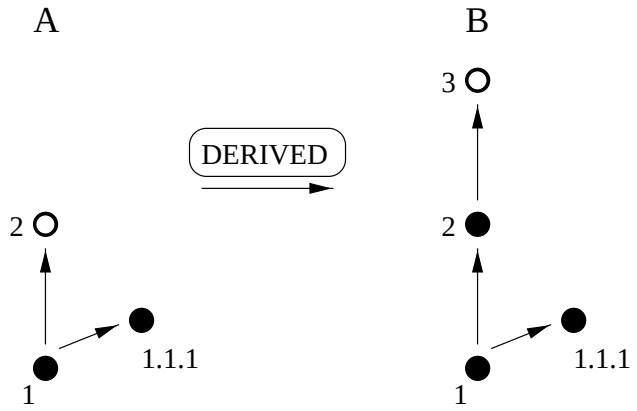

Abbildung 4.6: Graph B ist von Graph A abgeleitet

Verschmelzung dieser beiden Basisobjekte zum resultierenden Versionsgraphen, als Nachfolger der beiden Basisobjekte, hinzugefügt (siehe Abb. 4.8). Die Versionsnummer dieses neuen Objektes wird abgeleitet von der Versionsnummer des Basisobjektes des Basisversionsgraphen.

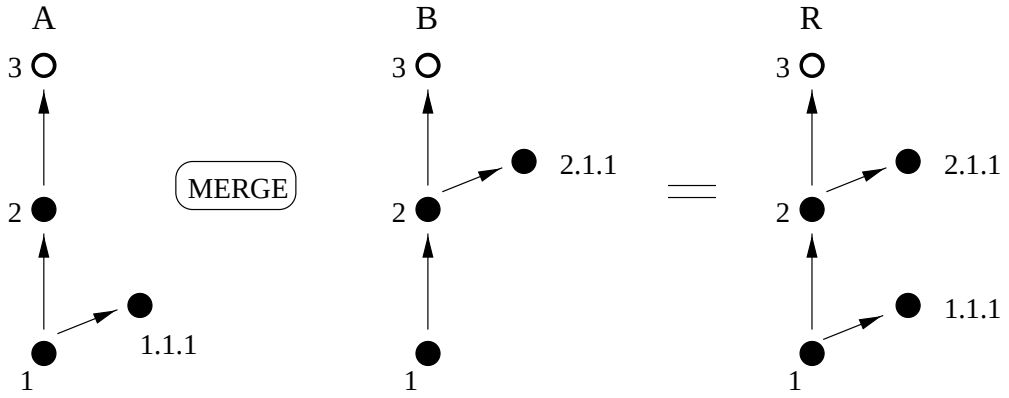

Abbildung 4.7: Verschmelzung von Versionsgraphen

Bei der Verschmelzung von Versionsgraphen, deren Äste gleiche Versionen, jedoch unterschiedliche Daten beinhalten, es also Konflikte zwischen den Versionsgraphen gibt, wird eine der Konfliktversionen vor dem Hinzufügen zum Ergebnisgraphen umbenannt. Bei der Verschmelzung zweier Versionsgraphen mit Konflikt gibt es jeweils zwei mögliche Ergebnisgraphen, je nachdem, welcher der beiden Versionsgraphen die Basis für den resultierenden Versionsgraphen ist. Enthält ein Versionsgraph *A* z.B. drei Objekte *1*, *2* und 3_a, und ein Versionsgraph *B* die drei Objekte *1*, *2* und 3_b (siehe Abb. 4.9) und sind die Objekte 3_a und 3_b unterschiedlich, so gibt es einen Konflikt. Dieser Konflikt kann gelöst werden, wenn entweder Objekt 3_a oder Objekt 3_b in *2.1.1* umbenannt wird. Anschließend werden die beiden Standardbasisobjekte verschmolzen (Objekte *3* und *2.1.1*).

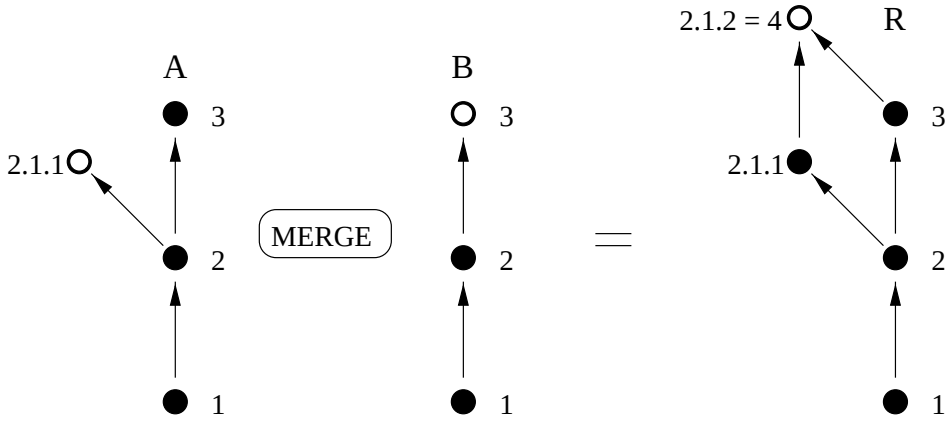

Abbildung 4.8: Verschmelzung mit unterschiedlichen Standardbasisobjekten

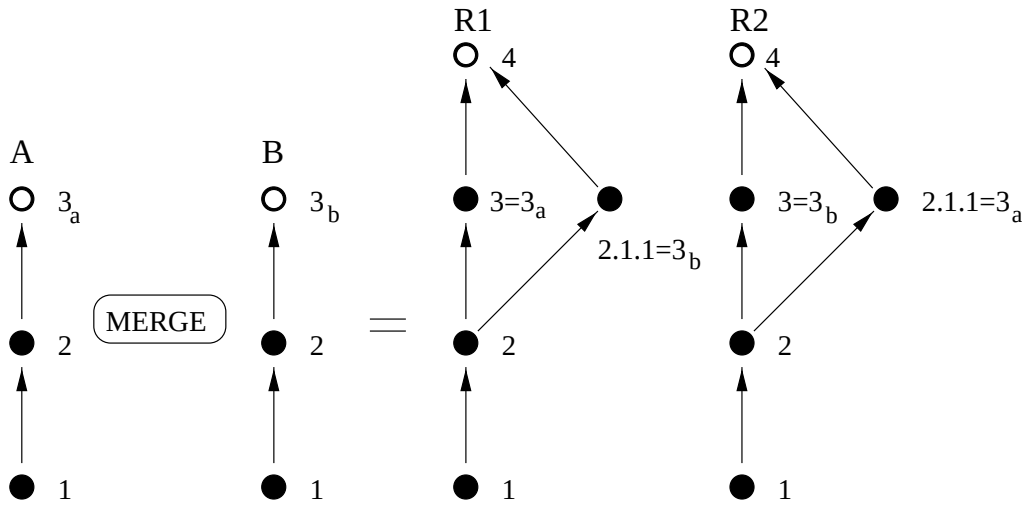

Abbildung 4.9: Verschmelzung mit Konflikt

Im Anhang C finden sich weitere Bespiele für die Verschmelzung von Versionsgraphen. Ist ein Versionsgraph R aus der Verschmelzung zweier anderer Versionsgraphen A und B hervorgegangen, so steht der abgeleitete Versionsgraph R mit den Versionsgraphen A und B jeweils in einer *ist-abgeleitet-von* Beziehung.

4.1.3.2 Versionierungssysteme (RCS und SCCS)

Zwei verbreitete Versionierungssysteme sind das RCS und das SCCS. Diese beiden Systeme erlauben es, Textdateien, insbesondere mit Progammtexten, zu versionieren. RCS und SCCS sind dateibasiert, wobei für jedes zu versionierende Objekt eine Datei (das Archiv) mit allen bisherigen Änderungen dieses Objektes verwaltet wird. Das Archiv läßt sich als eine linearisierte Form eines Versionsgraphen

betrachten.

Folgende grundlegende Funktionalität stellen diese Versionierungssysteme zur Verfügung:

Hinzufügen (engl. check in) Beliebige Objekte können zum Archiv hinzugefügt werden; wird keine Basisversion spezifiziert, so wird eine voreingestellte Basisversion (das Standardbasisobjekt) verwendet.

Extrahieren (engl. check out) Eine registrierte Version eines Objektes kann aus dem Archiv wieder extrahiert werden. RCS und SCCS erlauben es beide, diese Version anschließend für andere Benutzer zu sperren, d.h. es kann, so lange diese Version gesperrt ist, kein anderer Benutzer einen Nachfolger zu der extrahierten Version zum Archiv hinzufügen.

Vereinigen (engl. merge) Eine Version eines Objektes kann als die Vereinigung zweier Versionen des Objektes zum Archiv hinzugefügt werden.

4.1.3.3 Versionierung im Tycoon-System

Im Rahmen dieser Arbeit entstand ein Versionierungssystem für TYCOON. Die Basisfunktionalität ist vergleichbar der des RCS und des SCCS. Um die Orthogonalität von Projektverwaltung und Versionierung zu erreichen, wurde die Funktionalität um folgende Möglichkeiten ergänzt:

Ableitung: Für Archive kann getestet werden, ob sie in einer *ist-abgeleitet-von* Beziehung stehen.

Vereinigung: Zwei Archive können zu einem neuen Archiv vereinigt werden.

Abweichend zum RCS und SCCS gibt es keine Möglichkeit, eine bestimmte Version innerhalb eines Archives zu sperren. Da auf Grund der Unabhängigkeit von Projektverwaltung und Versionierung immer nur ein Benutzer zur Zeit Zugriff auf ein Archiv hat, wird eine solche Funktionalität auch nicht benötigt.

Die Implementierung der Archivierung ist typunabhängig, benötigt aber zwei Funktionen, um die Elemente eines instantiierten Archivs vergleichen bzw. verschmelzen zu können.

Die Schnittstelle des Moduls *Archive* ist im Anhang A angegeben.

4.1.4 Kompatibilität von Modulvarianten

Module und Schnittstellen in Objektsystemen unterliegen, ebenso wie andere Objekte, der Evolution. Neue Versionen von Modulen unterscheiden sich in Semantik

oder Funktionalität von vorherigen Versionen. Exportierte Typen und Schnittstellen können sich im Laufe der Weiterentwicklung und Anpassung an neue Umstände verändern. Im Zusammenhang mit dynamischer Rebindung stellt sich die Frage, wann zwei Module, die in der Versionsfolge aufeinander folgen, nicht mehr bzw. noch kompatibel sind.

Folgende Arten von Kompatibilität lassen sich unterscheiden:

- Zwei Module sind statisch kompatibel, wenn sie zur Übersetzungszeit ausgetauscht werden können (siehe Beispiel 4.1).
- Zwei Module sind dynamisch kompatibel, wenn sie zur Laufzeit ausgetauscht werden können (siehe Beispiel 4.2).

```
interface A
export
  T <: Ok

  new(value :Int) :T

  getValue(:T) :Int
  setValute(:T  value :Int) :Ok
end;
```

```
interface B
export
  T <: Ok

  new(value :Int) :T
  newWithName(value :Int  name :String) :T

  getValue(:T) :Int
  setValute(:T  value :Int) :Ok

  getName(:T) :String
  setName(:T  name :String) :Ok
end;
```

Beispiel 4.1: Modul B ist statisch abwärtskompatibel zu Module A

Prinzipiell kann gesagt werden, daß zwei Module dynamisch kompatibel zueinander sind, wenn die Typen der Tupel der exportierten Attribute kompatibel sind. Module, die die Typen ihrer Daten nicht exportieren, sind höchstens statisch kompatibel (z.B. abstrakter Datentyp). Die Kompatibilität zweier Module kann auch gerichtet sein (siehe Beispiele 4.1 und 4.2)

4.1.5 Dynamisches Ableiten von Programmobjekten

Migriert ein Objektgraph von einem Objektsystem in ein zweites Objektsystem und wird der Objektgraph vor der Migration um Programmobjekte reduziert, so müssen im Zielobjektsystem zu den entfernten Programmobjekten kompatible Programmobjekte für die Expansion zur Verfügung gestellt werden. Um nicht alle möglichen Programmobjekte vorhalten zu müssen, werden Objektsysteme

```
interface A
export
  Let T = Tuple value :Int end

  new(value :Int) :T

  getValue(:T) :Int
  setValute(:T  value :Int) :Ok
end;
```

```
interface B
export
  Let T = Tuple value :Int  name :String end

  new(value :Int) :T
  newWithName(value :Int  name :String) :T

  getValue(:T) :Int
  setValute(:T  value :Int) :Ok

  getName(:T) :String
  setName(:T  name :String) :Ok
end;
```

Beispiel 4.2: Modul B ist dynamisch abwärtskompatibel zu Module A

in Kooperationsverbünden bezüglich einer Projektverwaltung zusammengefaßt. Ein Objektsystem dient dabei als Elternarbeitsbereich für eine Menge anderer Objektsysteme. Wird in einem Objektsystem ein bestimmtes Programmobjekt benötigt, so wird es aus dem zugehörigen Quelltext abgeleitet. Ist der Quelltext nicht vorhanden, so kann er vom Elternarbeitsbereich geholt werden. Um z.B. Code-Programmobjekte dynamisch ableiten zu können, ist es notwendig den Übersetzer zur Laufzeit zu aktivieren. Außerdem kann es Abhängigkeiten zwischen den Modulen der Bibliotheken des Systems geben, welche berücksichtigt werden müssen (vgl. [Koc97]).

Können Programmobjekte dynamisch abgeleitet werden, so können in einem Objektgraphen Programmobjekte dynamisch aktualisert werden.

4.2 Immobile Objekte

Migrieren Objektgraphen innerhalb von Computersystemen oder sollen Objekgraphen persistent gespeichert werden, so stellen Objekte mit einer externen Bindung, also Objekte deren Inhalt semantisch mit einem Objekt außerhalb des Objektspeichers verknüpft ist (siehe Abb. 4.10), ein Problem dar. Wird der externe Kontext eines solchen Objektes verändert (z.B. nach der Speicherung und vor der Neuaktivierung) oder ausgetauscht (z.B. durch Migration des Objektes), so kann die Bindung zerstört werden.

Beispiele für Objekte mit externer Bindung:

- Ein Objekt enthält einen Ganzzahlwert. Dieser Wert ist mit einer, inner-

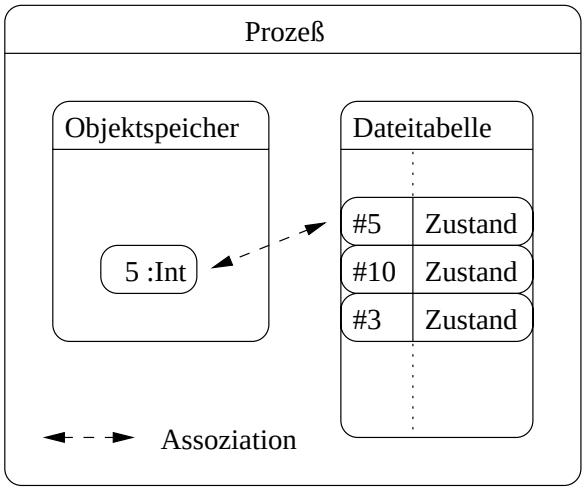

Abbildung 4.10: Semantische externe Objektbindung

halb des aktuellen Prozesses geöffneten, Datei assoziiert. Wird das Objekt gespeichert oder migriert es in einen anderen Objektspeicher, so geht die Assoziation zwischen Ganzzahlwert und Datei verloren, da diese kontext- bzw. prozeßabhängig ist.

- Ein Objekt ist mit einer Datei über den Dateipfad und Dateinamen assoziiert. Migriert dieses Objekt, so wird, abhängig von der Datei und von Dateinetzwerkdiensten, dieser Pfad nicht mehr gültig sein.

- Ein Objekt ist mit einem Druckdienst assoziiert. Durch eine Migration dieses Objektes geht die Assoziation verloren.

- Ein Objekt, das mit einem Kommunikationsendpunkt verbunden ist, der wiederum mit einer computerlokalen Netzwerkkennung assoziiert ist, ist weder persistent speicherbar, noch migrationsfähig. Da jede computerlokale Kennung einmalig ist, kann es bei dem Versuch, ein derartiges Objekt wiederherzustellen, zu Problemen kommen.

Dynamische Rebindung erlaubt es, immobile oder nicht speicherbare Objekte aus einem Objektgraphen zu entfernen und durch Beschreibungsobjekte zu ersetzen, mit deren Hilfe zu den Ursprungsobjekten äquivalente Objekte erzeugt werden können (siehe Abb. 4.11). Nach der Migration bzw. Reaktivierung eines Objektgraphen werden diese Beschreibungsobjekte, wiederum mit Hilfe dynamischer Rebindung, durch aus den Beschreibungsobjekten erzeugte Objekte ersetzt. Die Beschreibungen für immobile oder transiente Objekte werden vom Anwendungsentwickler für jeden immobilen oder transienten Objekttyp explizit entwickelt (siehe Kapitel 5).

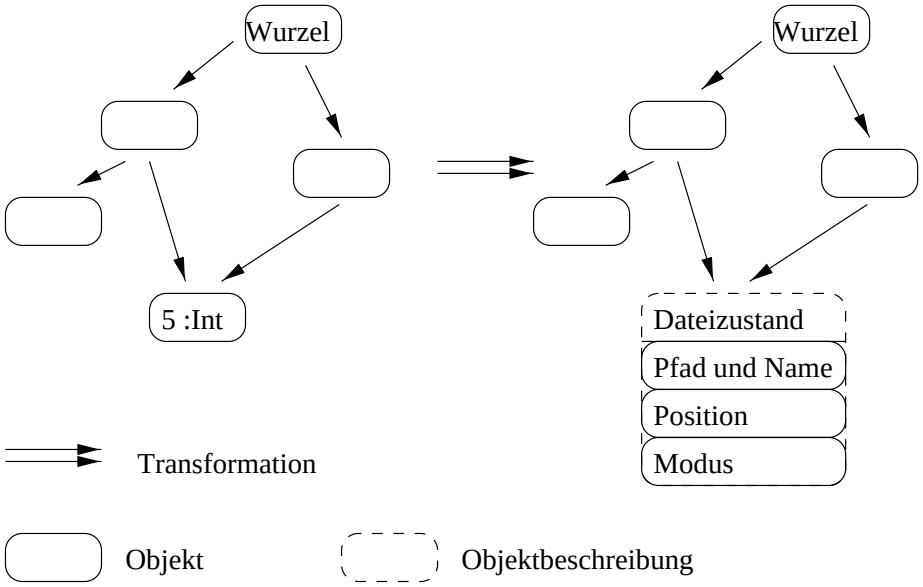

Abbildung 4.11: Mobilitätstransformation eines Objektgraphen

Es gibt prinzipiell zwei Arten, Beschreibungsobjekte für immobile Objekte zu erzeugen. Es wird kontextabhängig für jede Art semantischer Bindung eine eigene Beschreibung zur Wiederherstellung dieses oder eines äquivalenten Objektes erzeugt oder es wird eine Beschreibung für die Erzeugung eines Repräsentanten erzeugt (siehe Abb. 4.12). Die zweite Technik hat gewisse Voraussetzungen (siehe Kapitel 2), die nicht von allen Objektspeichern erfüllt werden.

4.2.1 Spezifische Erzeugung von Ersatzobjekten

Um Bindungen eines migrierenden Objektgraphen an immobile Objekte im Zielobjektspeicher durch Bindungen an äquivalente Objekte ersetzen zu können, ist es für jede Art externer semantischer Bindung notwendig, eine Funktion zur Erzeugung einer Beschreibung der Bindung zu entwickeln. Eine zweite Funktion wird benötigt, um aus einer Beschreibung wieder eine konkrete externe Bindung herstellen zu können.

4.2.2 Generische Erzeugung von Ersatzobjekten

Erfüllt der Objektspeicher eines Objektsystems die Voraussetzung für Repräsentanten, so können generische Funktionen, für die Erzeugung einer Beschreibung einer externen Bindung, entwickelt werden. Die Funktion, die aus der Beschreibung wieder ein Objekt erzeugt, erzeugt jetzt einen Repräsentanten, der auf das

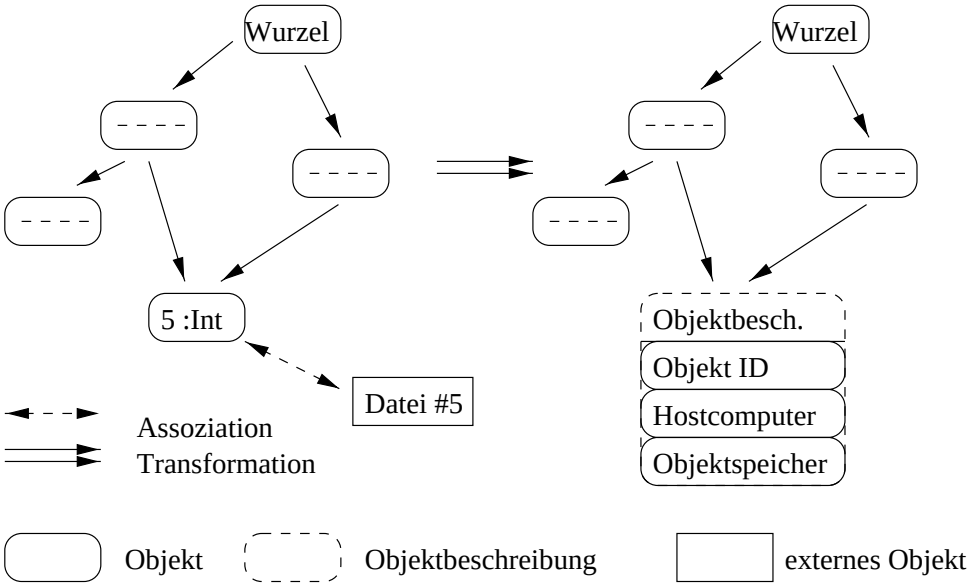

Abbildung 4.12: Generelle Mobilitätstransformation

ursprüngliche Objekt verweist. Diese Technik ist nur bei migratorischen Anwendungen anwendbar.

4.3 Ubiquitäre Objekte

Objekte, die in jedem Objektsystem vorhanden sind, werden als ubiquitär bezeichnet. Da dies eine semantische Eigenschaft ist, müssen derartige Objekte explizit als ubiquitär markiert werden. Ubiquitäre Objekte brauchen idealerweise nicht zu migrieren bzw. gespeichert zu werden, da sie immer vorhanden sind.

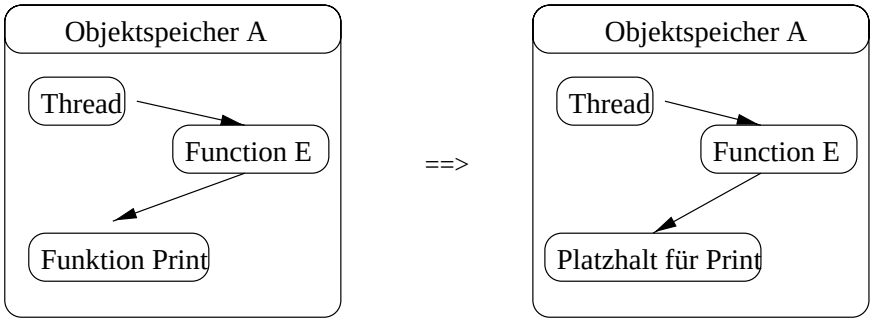

Abbildung 4.13: Ersetzen des ubiquitären Objektes durch einen Platzhalter

Ubiquitäre Objekte sind z.B. jene Objekte, die Standardfunktionalitäten der zum System gehörigen Bibliotheken implementieren. Softwarekomponenten werden im

allgemeinen unter Verwendung der Standardkomponenten entwickelt. Ein migrierender Thread wird in seiner transitiven referentiellen Hülle eine Menge von ubiquitären Objekten haben. Da diese Objekte überall verfügbar sind, ist es nicht notwendig diese migrieren zu lassen.

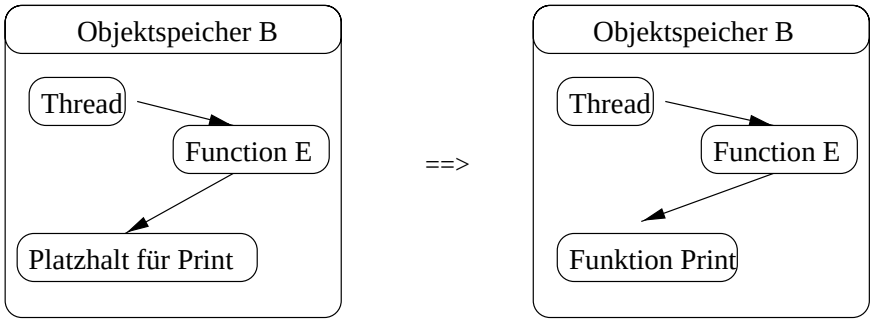

Abbildung 4.14: Ersetzen des Platzhalters durch ein Objekt

Abbildungen 4.13 und 4.14 zeigen einen vom Objektsystem A in das Objektsystem B migrierenden Thread. Vor der Migration werden ubiquitäre Objekte durch Platzhalter ersetzt. Nach der Migration werden die Platzhalter durch äquivalente Objekte des Zielobjektsystems ersetzt.

Kapitel 5

Dynamische Rebindung in kooperierenden Objektsystemen

Dieses Kapitel integriert die in den vorigen Kapiteln vorgestellten Konzepte und Techniken zu einem Gesamtkonzept. Zum einen wird dargestellt, wie auf Basis der Technik des dynamischen Rebindens ein generisches Rebindungskonzept implementiert werden kann, zum anderen wird gezeigt, wie all diese Dinge transparent zur Anwendung gebracht werden können. Die Anwendung ist in Form eines Arbeitskontextes implementiert. Dieser Arbeitskontext stellt die Basisumgebung dar, um Objektgraphen zu migrieren, persistent zu speichern oder zu aktualisieren. Neue Versionen von Modulen und Schnittstellen werden, mit Hilfe des Arbeitskontextes, in den Kooperationsverbund eingebunden. Auch das dynamische Ableiten von Programmobjekten und die Integration von Objekten in den Kontext des Übersetzers wird auf Basis des Arbeitskontextes geleistet.

5.1 Der Arbeitskontext

Der Arbeitskontext integriert dynamische Rebindung, Projektverwaltung, Versionierung und Rebindungsobjekte. Seine Aufgabe ist es, das Migrieren von Objekten sowie das Speichern von Objekten unter Aspekten dynamischer Rebindung zu ermöglichen. Gleichzeitig ist der Arbeitskontext die Basis für kooperative Arbeit, wie z.B. kooperative Softwareverwaltung. Arbeitskontexte sind, wie Arbeitsbereiche, baumartig organisiert, d.h. jeder Arbeitskontext hat einen oder keinen Elternarbeitskontext sowie beliebig viele Kindarbeitskontexte, wobei zyklische Strukturen nicht gestattet sind. Grund für diese Organisation der Arbeitskontexte sind die ubiquitären Objekte.

5.1.1 Integration in die Übersetzerumgebung

Für den Anwender eines Objektsystems, das sich in einer Kooperationsgemeinschaft befindet, soll die Kooperation des Objektsystems transparent sein. Ist z.B. ein Modul im Toplevel nicht verfügbar, so soll es dem Anwender möglich sein, dieses Modul durch eine einfache Anweisung in das System zu integrieren. Die Integration des, in dieser Arbeit entwickelten, Arbeitskontextes in den Toplevel des TlMin Systems erfolgt rein funktional. Für jede mögliche Operation auf einem Arbeitskontext wird im Toplevel eine Funktion definiert, die an die Instanz des Arbeitskontextes für dieses Objektsystem gebunden ist. Möchte der Anwender eine Operation auf dem Anwendungskontext ausführen, so verwendet er die im Toplevel definierte Funktion (siehe Beispiele 5.1 und 5.2).

```
importObject(array 1 2 1 end  "list");
(* Modul "list" in der Version 1.2.1 wird importiert *)
```

Beispiel 5.1: Modulimport

```
let f(context :workItf.T) :Ok = begin
    ...
end;

remoteExecute("localhost" "clientA" f);
(* Ausführung der Funktion "f" im Objektsystem "clientA" *)
```

Beispiel 5.2: Entfernte Funktionsausführung

5.1.2 Softwarebibliotheken

Innerhalb des Toplevels eines TlMin Systems kann ein Anwender Eingaben vornehmen und ausführen. Alle innerhalb des Toplevels sichtbaren Module kann der Anwender dabei verwenden. Z.B. kann der Anwender eine Funktion definieren, die anschließend in einem entferten Arbeitskontext ausgeführt werden soll. Verwendet der Anwender vordefinierte Module für die Implementation dieser Funktion, so ist es Aufgabe des Arbeitskontextes den migrierenden Objektgraphen vor der Migration um diese Objekte zu reduzieren und Aufgabe des Arbeitskontextes des Zielobjektsystems den reduzierten Objektgraphen wieder zu expandieren. Damit der Ausführungskontext den Objektgraphen reduzieren kann, müssen ihm alle im Toplevel verfügbaren Module bekannt gemacht werden. Um einen Objektgraphen expandieren zu können, ist es notwendig, im System nicht vorhandene Module dynamisch abzuleiten. Voraussetzung für das dynamische Ableiten von Modulen ist das Laden der zugehörigen Bibliotheken in die Projektverwaltung.

5.1.3 Schwache Referenzen

Referenzen werden als schwach (engl. weak references; vgl. [Sch97]) bezeichnet, wenn die Objekte, die die schwachen Referenzen halten, mit dem referenzierten Objekt abgebaut werden. Wird ein Objekt im Objektspeicher nur noch schwach referenziert, so werden die Objekte benachrichtet, die diese Referenzen halten, anschließend wird das schwach referenzierte Objekt abgebaut (siehe Abb. 5.1). Eine Tabelle mit Objekten zur Vorhaltung (engl. cache) entfernt Einträge für Objekte, wenn diese Objekte nur noch von der Vorhaltetabelle gehalten werden. Da die ehemals vorgehaltenen Objekte daraufhin nicht mehr referenziert werden, werden sie abgebaut.

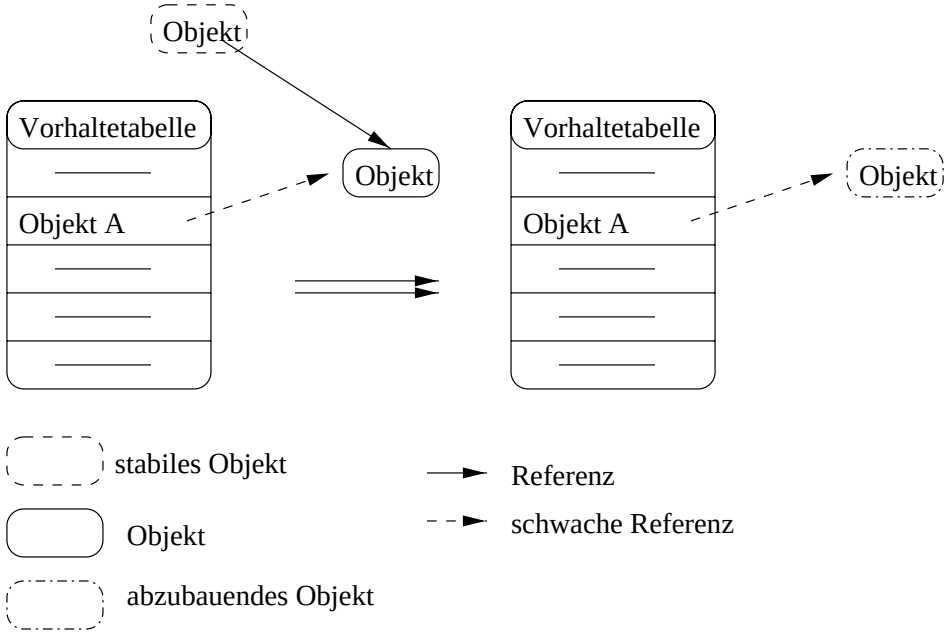

Abbildung 5.1: Schwache Referenz

5.1.4 Begrenztes Vorhalten von Codeobjekten

Da Codeobjekte, insbesondere Instantiierungen von Bibliotheken, von vielen Anwendungen genutzt werden, ist es sinnvoll, Anwendungen, die gleiche Codeobjekte referenzieren, an eine Instanz dieser Codeobjekte zu binden (engl. sharing). Um dies zu erreichen, dürfen nicht für jedes migrierende oder persistente Objekt, bei Integration in das Objektsystem, die zugehörigen Codeobjekte dynamisch neu erzeugt werden, sondern es sollen möglichst vorhandene Objekte verwendet werden.

Um Codeobjekte wiederverwenden zu können, ist es notwendig, eine Tabelle mit im System vorhandenen Codeobjekten zu führen. Wird ein Objektgraph in das

Objektsystem integriert, so wird für jedes benötigte Codeobjekt erst in dieser Tabelle nachgesehen, ob dieses oder ein kompatibles Objekt vorhanden ist. Wird ein passendes Objekt gefunden, so wird dieses verwendet, anstatt ein neues zu erzeugen.

Da im TYCOON System Objekte abhängig von ihrer Erreichbarkeit abgebaut werden und da auch Codeobjekte abgebaut werden sollen, ist es notwendig, die Referenzen der Tabelle auf Codeobjekte als schwache Referenzen zu definieren. So wird erreicht, daß, falls ein Codeobjekt nur von der Tabelle aus erreichbar ist, der zugehörige Eintrag aus der Tabelle entfernt wird und so das Codeobjekt abgebaut werden kann.

5.1.5 Vorhalten ubiquitärer Objekte und Replikation

Ubiquitäre Objekte sind allgegenwärtige Objekte. Wird ein Objekt als ubiquitär registriert, so wird eine Kopie dieses Objektes im transitiv obersten Elternarbeitskontext registriert. Anschließend ist dieses Objekt in allen untergeordneten Arbeitskontexten verfügbar.

Analog zum begrenzten Vorhalten von Codeobjekten gibt es ein begrenztes Vorhalten ubiquitärer Objekte. Der Anwendungskontext führt eine Tabelle mit ubiquitären Objekten. Wird ein Objektgraph in das Objektsystem integriert und werden ubiquitäre Objekte referenziert, so wird zunächst in dieser Tabelle nachgesehen, ob sich dieses ubiquitäre Objekt bereits innerhalb des Objektsystems befindet, ist dies nicht der Fall, so wird der Elternarbeitskontext nach diesem Objekt befragt, ist dieses Objekt in einem transitiv erreichbaren Elternarbeitskontext registriert, so wird es in den aktuellen Arbeitskontext migriert (automatische Replikation) bzw. es wird ein Repräsentant erzeugt (je nach den Möglichkeiten des zugrundeliegenden Objektspeichers und der Art des Objektes) und hier in der Tabelle für begrenztes Vorhalten registriert. Die Verweise der Tabelle sind, wie bei der Tabelle für Codeobjekte, als schwache Referenzen definiert, so daß nicht mehr referenzierte Kopien ubiquitäre Objekte bzw. Repräsentanten abgebaut werden können.

5.2 Der Rebindungsmanager

Der Rebindungsmanager ermöglicht es, Objekte in Objektgraphen dynamisch neu zu binden. Ein Objekt eines Objektspeichers, das Teil eines Objektgraphen ist, der persistent gespeichert werden soll, und das semantisch extern gebunden ist (z.B. an einen Kommunikationsendpunkt), bedarf vor der Speicherung des Objektgraphen besonderer Aufmerksamkeit. Die externe Bindung des Objektes muß vor der persistenten Speicherung explizit aufgelöst werden, damit sie, nach der

Restauration des Objektes in einem Objektspeicher, später wieder hergestellt werden kann.

Der Rebindungsmanager soll derart konfigurierbar sein, daß der Programmierer das Verhalten des Rebindungsmanagers für jedes Objekt individuell bestimmen kann. Es soll dem Programmierer möglich sein, jederzeit das Verhalten des Rebindungsmanagers bei der Ersetzung einer Bindung (Rebindung) zu bestimmen.

Unterschiedliche Kategorien von Objekten, um die ein migrierender Objektgraph reduziert werden kann, sind erkennbar (siehe Kapitel 4):

- Programmobjekte
- Immobile bzw. transiente Objekte
- Ubiquitäre Objekte

Die Aufgabe des Rebindungsmanagers ist die mögliche Assoziation zweier Funktionen mit einem Objekt. Eine Funktion erlaubt es, ein Objekt in eine allgemeine Darstellung des Objektes (ein Objekt mit einer externen Bindung an einen Kommunikationsendpunkt wird in ein Objekt konvertiert, das die externe Bindung beschreibt) zu überführen, und eine zweite Funktion erzeugt aus der allgemeinen Darstellung wieder ein konkretes Objekt.

5.2.1 Reduktion von Objektgraphen

Der Rebindungsmanager rebindet einen Objektgraphen, indem er, ausgehend von einem Objekt, die Menge aller von diesem Objekt referentiell transitiv erreichbaren Objekte bearbeitet. Bindungen, die ersetzt werden sollen, werden durch das bindende Objekt identifiziert. Alle bindenden Objekte eines Objektgraphen, deren Bindung verändert werden soll, müssen vor der Rebindung beim Rebindungsmanager registriert werden. Zu erneuernde Bindungen des Objektgraphen werden anhand der OIDs der bindenden Objekte erkannt.

Bindende Objekt können folgendermaßen durch andere Objekte ersetzt werden:

statisch: Ein bindendes Objekt wird beim Rebindungsmanager registriert, das ersetzende Objekt wird mit dem bindenden Objekt registriert.

dynamisch: Zusammen mit dem zu ersetzenden bindenden Objekt wird eine Funktion registriert. Diese Funktion wird für jedes ursprünglich bindende Objekt aktiviert und muß als Ergebnis das neu bindende Objekt zurückgeben. Diese Funktion kann für jede Aktivierung ein anderes Objekt zurückgeben, so ist es z.B. möglich, daß sich ursprünglich gleichende Verweise nach der Rebindung unterscheiden.

Abbildung 5.2 zeigt einen Objektgraphen und einen Rebindungsmanager. Das bindende Objekt **A** ist beim Rebindungsmanager registriert. Mit dem Objekt **A** ist das ersetzende Objekt **B** assoziiert. Wird der Rebindungsmanager für den Objektgraphen aktiviert, so werden alle Referenzen auf **A** durch Referenzen auf **B** ersetzt.

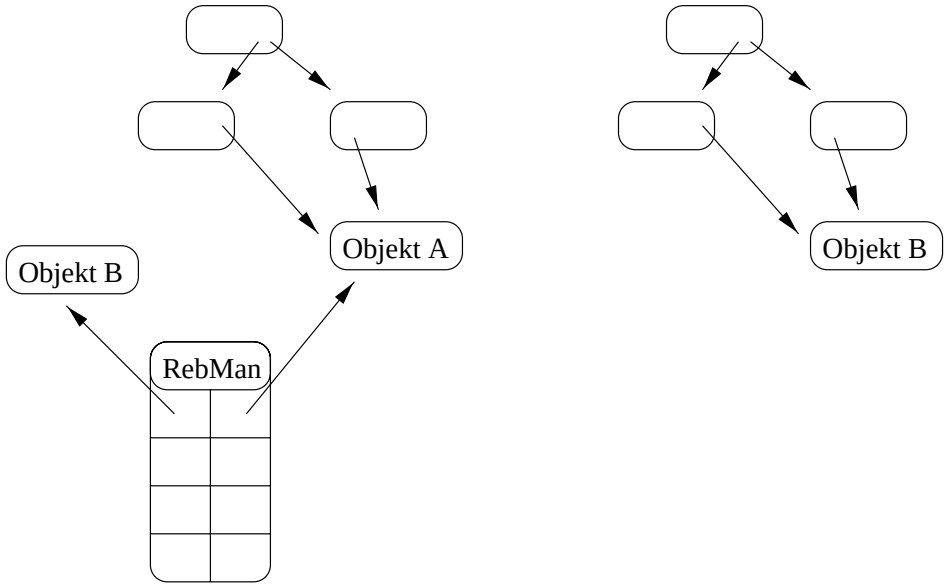

Abbildung 5.2: Reduktion eines Objektgraphen (statisch)

Objekte für die Rebindung können auch dynamisch generiert werden (siehe Abb. 5.3).

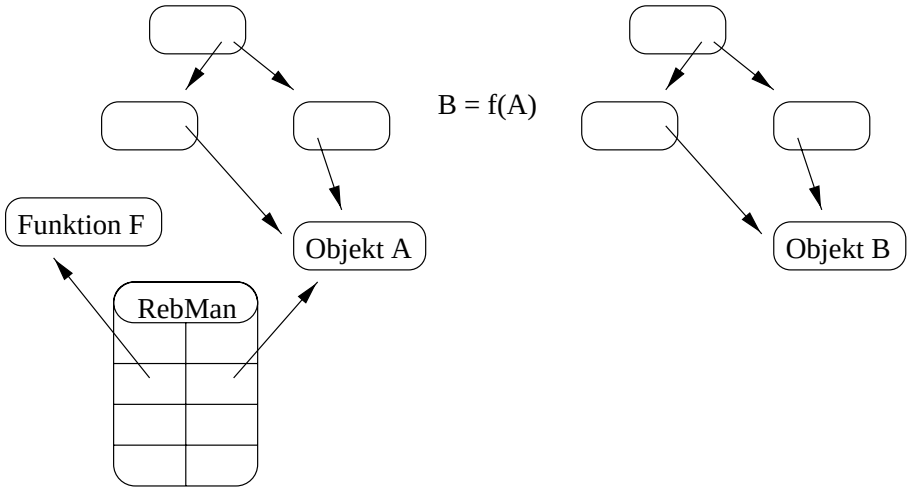

Abbildung 5.3: Reduktion eines Objektgraphen (dynamisch)

5.2.2 Expansion von Objektgraphen

Reduzierte Objektgraphen, die z.B. persistent gespeichert werden oder die von einem Objektspeicher in einen anderen migrieren, müssen nach der Restaurierung im Objektspeicher wieder expandiert werden. Z.B. werden immobile Objekte vor der Migration eines Objektgraphen durch beschreibende Objekte ersetzt. Die Beschreibung enthält, bei dynamischer Reduktion, eine Funktion, die aus der Beschreibung im Zielobjektspeicher wieder ein konkretes, zum ursprünglichen Objekt kompatibles, Objekt erzeugt (siehe Abb. 5.4).

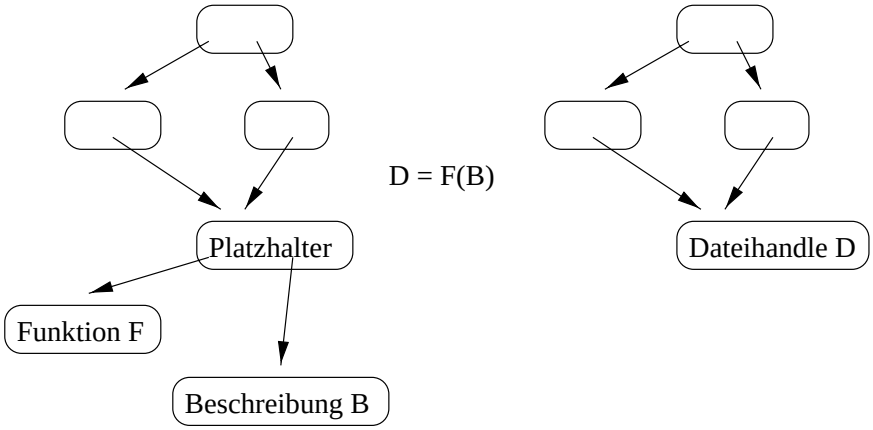

Abbildung 5.4: Expansion eines Objektgraphen (dynamisch)

5.2.3 Netzwerkreferenzen

Migriert ein Objekt von einem Objektsystem in ein anderes und sollen Bindungen an dieses Objekt erhalten bleiben, so werden Netzwerkreferenzen benötigt. Netzwerkreferenzen können entweder auf Objektspeicherebene vom Objektsystem zur Verfügung gestellt werden oder z.B. mittels Rückruffunktionen (siehe Kapitel 2) realisiert werden.

Der Rebindungsmanager ermöglicht es prinzipiell, Netzwerkreferenzen für im Ursprungsobjektspeicher verbleibende Objekte zu generieren. Objekte, die nicht mit dem Objektgraphen migrieren sollen, werden beim Rebindungsmanager registriert. Die Funktion für die Restaurierung eines solchen Objektes erzeugt im Zielobjektspeicher daraufhin einen Repräsentanten (siehe Abb. 5.5).

Da das TSP keine Funktionalität für transparenten indirekten Objektzugriff bereistellt, werden transparente Netzwerkreferenzen in dieser Arbeit nicht betrachtet.

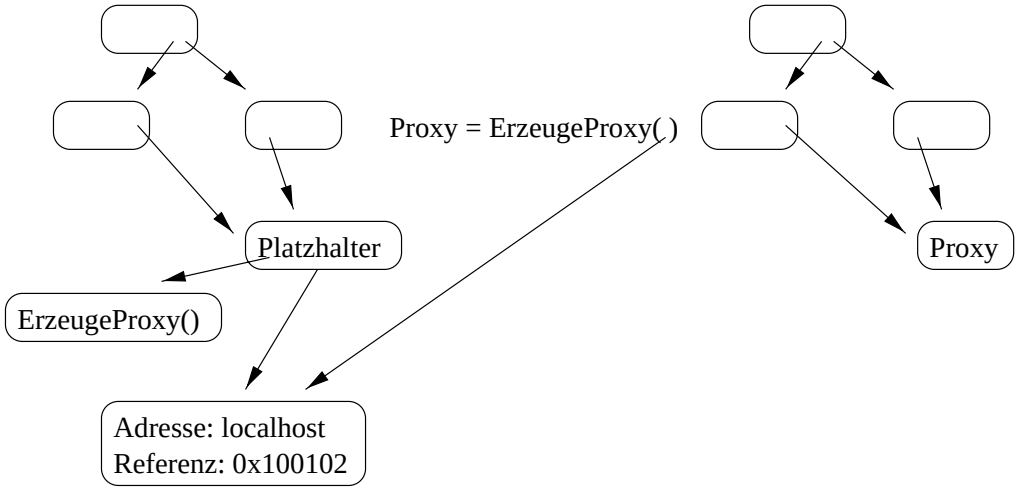

Abbildung 5.5: Netzwerkreferenz mit Repräsentant

5.3 Reflektion

Mit Hilfe von Reflektion kann z.B. das Verhalten der Programmierumgebung eines Objektsystem verändert werden. Wird z.B. ein Fehler im Syntaxanalysator (engl. parser) des Übersetzers entdeckt, so kann dieser zur Laufzeit korrigiert werden. Erlaubt es ein Objektsystem reflektiv auf die Komponenten des Übersetzer zuzugreifen, so können mit dem integrierten Übersetzer Programmobjekte dynamisch abgeleitet werden.

Objektsysteme können Programmierumgebung und Anwendungsumgebung in ein System integrieren. Ist die Programmierumgebung des Objektsystems ebenfalls mit der integrierten Programmierumgebung, also mit sich selbst (reflektiv; vgl. [KCMS96]), entwickelt worden, so ist es möglich, reflektiv auf diese zuzugreifen. Voraussetzung hierfür ist, daß die Programmierumgebung eine Schnittstelle für den Zugriff auf ihre Komponenten zur Verfügung stellt.

5.3.1 Reflektion der Übersetzer Komponenten

Die Programmierumgebung bzw. der Übersetzer eines Objektsystems sind Programme. Ist der Übersetzer in der gleichen Programmiersprache entwickelt worden, wie die, die er zu verarbeiten versteht, und ist der Übersetzer integriert in das Objektsystems, so können die Komponenten des Übersetzers zur Laufzeit im Objektsystem sichtbar gemacht werden. Vorteil des Sichtbarmachens gegenüber dem Nachladen ist der reduzierte Speicherverbrauch sowie die konsistente Verwendung der Implementierung der Komponenten.

5.3.2 Reflektion des Übersetzers

Reflektion kann nicht nur dazu verwendet werden, statische Komponenten der aktuellen Programmierumgebung sichtbar zu machen, auch dynamisch erzeugte Daten können exportiert werden. Will ein Anwendungsprogramm beispielsweise den aktuellen Übersetzerkontext verändern, so erlaubt ein reflektiver Zugriff auf die aktuelle Umgebung, mit Hilfe der Übersetzermodule, das dynamische Einbinden bespielsweise extern gehaltener Module oder Daten.

5.3.3 Die Übersetzerschnittstelle

Mit Hilfe der Übersetzerschnittstelle ist es möglich, zur Laufzeit programmiersprachliche Konstrukte zu übersetzen. Eine idealsierte Übersetzerschnittstelle impliziert Programmcode folgender Art:

let *result = Compile("let a = 5;");*

Die meisten Übersetzungen werden jedoch nicht kontextfrei ausgeführt, weshalb folgendes Beispiel einen Kontext integriert:

let *result = Compile("let a = 5;" context);*

Der Kontext stellt den Zustand des Übersetzers dar, der mit jedem Übersetzungsvorgang erweitert werden kann. Der Kontext kann als Übersetzer und das Übersetzen als eine Methode des Kontextes betrachtet werden.

Folgende Basisfunktionalität sollte ein Übersetzer zur Verfügung stellen:

- Ein Übersetzer muß instantiiert werden können.
- Ausdrücke müssen übersetzt werden können. Dies ist abhängig vom aktuellen Zustand des Übersetzers.
- Der Zustand des Übersetzers sollte von außen manipuliert werden können, d.h. bereits definierte Objekte des Übersetzers müssen zugreifbar sein, neue Objekte müssen, außerhalb eines Programmes, definiert werden können.

Obige Punkte sind die Voraussetzung um einen minimalen Übersetzer systematisch zu erweitern.

5.4 Freiheitsgrade der Kombinationen der Komponenten

Bei der Migration eines Objektgraphen gibt es verschiedene Möglichkeiten, Objekte zu bestimmen, die nicht migrieren sollen. Auch immobile Objekte können unterschiedlich gehandhabt werden. Es ist Aufgabe des Anwenders dynamischer Rebindung zu entscheiden, ob und wann Objekte dynamisch neu gebunden werden.

Folgende Liste gibt einen Überblick über Freiheitsgrade dynamischer Rebindung und Anwendungen dynamischer Rebindung:

- Welches Ziel soll mit dem Einsatz dynamischer Rebindung erreicht werden? (Aktualisierung, Persistenz / Mobilität)
- Welche Objekte bzw. welche Kategorien von Objekten sollen neu gebunden werden? (Programmobjekte, immobile / transiente Objekte, ubiquitäre Objekte)

Insbesondere migrierende Objektgraphen erschließen eine Menge weitere Freiheitsgrade bzgl. dynamischer Rebindung:

- Sollen ubiquitäre Objekte migrieren? Soll eine Teilmenge der ubiquitären Objekte migrieren?
- Sollen Codeobjekte migrieren?
- Wenn Codeobjekte nicht migrieren, soll der Objektgraph im Zielobjektspeicher an die jeweils aktuellste kompatible Codeobjektversion gebunden werden?
- Sollen für immobile Objekte im Zielobjektspeicher neue kompatible Objekte erzeugt werden?
- Sollen Verweise auf immobile Objekte durch Netzwerkverweise ersetzt werden?

Die entwickelte Beispielanwendung verwendet dynamische Rebindung zur Aktualisierung von Programmobjekten. Migrierende Objektgraphen werden um Programmobjekte und ubiquitäre Objekte sowie beispielhaft um Dateiidentifikatoren reduziert. Für in das System integrierte Programmobjekte wird vom Anwender die Kompatibilität beschrieben. Bei Expansion eines Objektgraphen werden Programmobjekte und ubiquitäre Objekte soweit wie möglich wiederverwendet.

Kapitel 6

Erfahrungen und Ausblick

Die in dieser Arbeit entwickelten Konzepte und Prinzipien erlauben es, migratorische Anwendungen für kooperative Objektsysteme zu entwickeln, ohne spezielle Vorkehrungen für immobile, transiente oder ubiquitäre Objekte treffen zu müssen. Mit den hier vorgestellten Verfahren ist es z.B. möglich, migrierende Agenten zu entwerfen, ohne daß Rücksicht genommen werden muß auf eventuell immobile Objekte oder auf Objektgraphen, die zuviele Objekte in ihrer transitiven referentiellen Hülle haben, um effizient im Netzwerk übertragen zu werden. Aktuelle Ausführungszustände von laufenden Prozessen können persistent gespeichert und später wieder restauriert werden, ohne daß spezielle Vorkehrungen für transiente Objekte getroffen werden müssen. Voraussetzung dafür ist, daß Generatoren immobiler bzw. nicht persistenter Objekte diese Objekte nach der Erzeugung beim Rebindungsmanager registrieren, da das Ersetzen von Bindungen eines Objektgraphen in Abhängigkeit von den bindenden Objekten geschieht. Immobile oder nicht persistente Objekte, die nicht vom Rebindungsmanager registriert sind, können vom Rebindungsmanager bei Rebindung nicht beachtet werden.

Die Sprache TLMIN der TYCOON Systeme wurde um Möglichkeiten der externen Modulverwaltung, verglichen mit TL, reduziert. Die in dieser Arbeit vorgenommene Beispielimplementation ergänzt TLMIN um eine externe Modulverwaltung. Module und Schnittstellen können über ein Netzwerk in den aktuellen Objektspeicher integriert werden und sind anschließend im Toplevel verfügbar. Die aktuelle Implementierung erlaubt es zur Zeit noch nicht, bereits vorübersetzte Module und Schnittstellen zu importieren, Module und Schnittstellen werden derzeit noch vor Ort übersetzt.

Bezüglich der Integration der Softwarekomponentenverwaltung und Archivierung ist diese Arbeit z.T. als Weiterentwicklung der Arbeit von [Koc97] zu betrachten. Die generische Lösung der Probleme, die immobile bzw. transiente Objekte für migrierende Objektgraphen aufwerfen, kann als Ergänzung zur Arbeit von [Mat96] gesehen werden, wobei gerade durch die Integration der Softwarekomponenten-

verwaltung wiederum der Kooperationsaspekt betont wird. Unter anderem durch die Entwicklung eines Arbeitskontextes, ist es gelungen, die Konzepte von [VP96] zu generalisieren und zu integrieren.

6.1 Administration kooperierender Objektsysteme

Die entwickelten Konzepte dieser Arbeit sind Basiskonzepte für eine zentrale Administration kooperierender Objektsysteme. Z.B. können migrierende Agenten entwickelt werden, die die Funktionalität erreichbarer Objektsysteme aktualisieren. Die Möglichkeiten, die sich aus automatischen Aktualisierungen ergeben, sind noch zu analysieren. Beispielsweise wäre es möglich, statistische Informationen über Funktionen eines Objektsystems zu sammeln (engl. profiling), um diese Funktionen, durch bezüglich dieser Informationen optimierte Funktionen, zu ersetzen. Die zentrale Administration von Computersystemen ist die Motivation für den augenblicklich zu erkennenden Trend, hin zu sogenannten Netzwerkcomputern (engl. nc = network computers). Diese Computer besitzen keine eigenen persistenten lokalen Speicher mehr und importieren die gesamte Anwendungssoftware dynamisch über das Netzwerk. Hierdurch ist gewährleistet, daß die Anwendungen zentral administrierbar sind. Die vorgestellten Konzepte würden es erlauben, auch Netzwerkcomputer mit persistentem Speicher auszustatten, Aktualisierungen würden durch migrierende Agenten vorgenommen werden. Prozesse des Anwenders könnten zyklisch in den Server migrieren, so daß bei Ausfall des Computers der letzte aktuelle Zustand der aktiven Anwendung wiederhergestellt werden kann.

6.2 Repräsentanten

Mit Hilfe von Objektspeichern, die es erlauben, Rückruffunktionen mit Objektoperatoren zu assoziieren, kann eine Art von dynamischer Rebindung implementiert, die in dieser Arbeit nicht weiter betrachtet wurde, da das TSP eine derartige Funktionalität nicht bietet. Repräsentanten erlauben es, Objekte im Netzwerk transparent zu bewegen, Netzwerkreferenzen können mit ihrer Hilfe transparent implementiert werden. Das Übersetzen auf Verlangen (engl. compiling on demand) läßt sich ebenfalls mit Repräsentanten realisieren. Ein Modul wird z.B. erst dann übersetzt und in das Objektsystem integriert, wenn es benötigt wird. In diesem Fall könnte z.B. auch nativer Code erzeugt werden, der die Ausführung der Funktionen des Moduls erheblich beschleunigen würde.

6.3 Implementation

Teil dieser Arbeit ist die Implementation einer Beispielanwendung. Es wurden die, in den vorherigen Kapiteln beschriebenen, Konzepte und Techniken implementiert. Zentrale Schnittstellen sind im Anhang aufgeführt. Um die Funktionalität einzelner Module vorzuführen sind einige Beispielskripte formuliert.

6.3.1 Module

Folgende Module sind implementiert:

- Das Modul *rebind* implementiert elementare Rebindungstechniken, wie z.B. Erzeugung einer Kopie eines Objektgraphen mit dynamischer Objektersetzung oder Aktualisierung eines Objektgraphen (siehe Anhang A.1).
- Der Rebindungsmanager erlaubt es, Objekte für den Rebindungsvorgang zu registrieren sowie spezielle Methoden für Reduktion und Expansion eines Objektes anzumelden. Implementiert ist der Rebindungsmanager im Modul *rebMan* (siehe Anhang A.2).
- Die Funktionalität des Portmappers wird im Modul *portmapper* definiert (siehe Anhang A.3).
- Mit Hilfe des Moduls *genCS* ist es möglich, auf einfache Weise Klienten und Server zu entwickeln, die mit dem Portmapper zusammenarbeiten (siehe Anhang A.4).
- Das Modul *archive* implementiert Versionierung, wie es im Kapitel über Versionsgraphen beschrieben ist (siehe Anhang A.5).
- Die Projektverwaltung befindet sich im Modul *workspace* (siehe Anhang A.6).
- Das Modul *pCompiler* ermöglicht dynamisches Übersetzen von Programmobjekten (siehe Anhang A.7).
- Die Module *pInterface* und *pModule* implementieren Programmobjekte für TYCOON Module und Schnittstellen (siehe Anhänge A.8, A.9 und A.10).
- Vom Modul *workItf* werden die anderen Module zu einer Beispielanwendung zusammengefaßt (siehe Anhang A.11).

Die Schnittstellen dieser Module sind im Anhang A aufgeführt. Im Anhang B sind Beispiele für die Anwendung der einzelnen Module, sowie für die Gesamtanwendung angegeben.

6.3.2 Demos

Im Anhang B sind drei Demos und drei Initalisierungsskripte angegeben. Mit Hilfe der Initialisierungsskripte wird ein Kooperationsverbund, bestehend aus drei Objektsystemen, erzeugt.

Folgendes wird demonstriert:

remoteExecuteDemo: Es wird eine Funktion definiert, die einmal im Objektsystem A und einmal im Objektsystem B ausgeführt wird. Vor der Migration von A nach B wird die Funktion um Programmobjekte reduziert und anschließend im Zielobjektsystem wieder expandiert. Da die Beispielanwendung so konfiguriert ist, für die Expansion möglichst ein vorhandenes, kompatibles Programmobjekt zu verwenden, und da in die Arbeitsumgebungen der Objektsysteme A und B unterschiedliche Versionen des Moduls *test* eingebunden sind, hat die Funktion abhängig vom Objektsystem unterschiedliche Ergebnisse.

mobileDemo: Es wird im Objektsystem A eine Datei geöffnet und anschließend eine Funktion definiert, die an den zugehörigen Dateiidentifikator gebunden ist. Diese Funktion migriert vom Objektsystem A in das Objektsystem B, wobei der Dateiidentifikator mit migriert. Im Zielobjektsystem wird der Inhalt der Datei ausgelesen und dann ausgegeben.

updateDemo: Eine Funktion F wird definiert, die an eine Funktion des Moduls *test* gebunden ist. Das Ergebnis der Ausführung der Funktion F hängt ab von der gebundenden Funktion. Das Modul *test* wird in neuer Version registriert und die Funktion F anschließend aktualisiert. Das Ergebnis der aktualisierten Funktion F hängt nun vom neuen Modul *test* ab.

Anhang A

Ausgesuchte Schnittstellen

A.1 Rebind.ti

```
interface Rebind
(* System : Workenv
   File   : Rebind.ti
   Author : Kay Ramme
   Date   : 30.01.1997
   Purpose: rebinding
   Version: $Id: Rebind.ti,v 1.4 1997/09/11 14:01:06 kay Exp $
*)
import
  tsp

export
  mark(oid :tsp.OID  bit :Int) :Ok
  (* mark a store object *)

  unmark(oid :tsp.OID  bit :Int) :Ok
  (* unmark a store object *)

  marked(oid :tsp.OID  bit :Int) :Bool
  (* test if store object is marked *)

  deepCopy(O <: Ok  obj :O
   pre(:tsp.OID) :tsp.OID
   mark(:tsp.OID) :Ok
   post(:tsp.OID) :tsp.OID) :O

  update(O <: Ok  obj :O
    pre(:tsp.OID) :tsp.OID
    post(:tsp.OID) :tsp.OID) :O
```

```
  dcopy(O <: Ok  obj :O) :O
end;
```

A.2 RebMan.ti

```
interface RebMan
(* System : Workenv
   File   : RebMan.ti
   Author : Kay Ramme
   Date   : 07.05.1997
   Purpose:
   Version: $Id: RebMan.ti,v 1.6 1997/09/11 14:01:25 kay Exp $
*)
import
  tsp

export
  T <: Oper(A <: Ok) Ok

  new(A <: Ok) :T(A)
  (* instantiate *)


  addObjDynamic(A <: Ok
    O <: Ok
    D <: Ok
    :T(A)
    obj :O
    toDcp(:O) :D
    toInst(:D :A) :O
    update(:D :A) :O
  ) :Ok
  (* add a service *)

  addObjStatic(A <: Ok  O <: Ok  :T(A)  obj :O  name :String) :Ok
  (* add an object replacement, rebinding via >name< *)

  delObjServ(A <: Ok  O <: Ok  :T(A)  obj :O) :Ok
  (* del a service *)


  isIn(A <: Ok  O <: Ok  :T(A)  obj :O) :Bool
  (* is object already added ? *)


  disable(A <: Ok  O <: Ok  :T(A)  obj :O) :Ok
  (* disable cutting obj *)

  enable(A <: Ok  O <: Ok  :T(A)  obj :O) :Ok
  (* enable cutting obj *)
```

```
  portablize(A <: Ok  O <: Ok  :T(A)  obj :O) :O
  (* replace akkumulated oids *)

  concretize(A <: Ok  O <: Ok  :T(A)  obj :O  :A) :O
  (* stabelize oids *)

  update(A <: Ok  O <: Ok  :T(A)  obj :O  :A) :O
  (* update oids *)

  (*** DEBUGGING ***)

  listEntrys(A <: Ok  t :T(A)) :Ok
  (* list all entrys *)


  display(A <: Ok  t :T(A)  obj :tsp.OID) :Ok
  (* displays an object graph *)
end;
```

A.3 PortMap.ti

```
interface PortMap
(* System : Workenv
   File   : PortMap.ti
   Author : Kay Ramme
   Date   : 14.04.1997
   Purpose:
   Version: $Id: PortMap.ti,v 1.2 1997/06/02 18:56:21 kay Exp $
*)
import

export

  portmap() :Ok
  (* portmapper *)

  new() :Ok
  (* create a portmapper thread *)

  getPort(host :String  service :String) :Int
  (* get port of service <name> *)

  setPort(host :String  service :String) :Int
  (* add service *)

  delPort(host :String  service :String) :Ok
  (* remove service *)

  quitPort(host :String) :Ok
  (* terminate portmapper *)
end;
```

A.4 GenCS.ti

```
interface GenCS
(* System : Workenv
   File   : GenCS.ti
   Author : Kay Ramme
   Date   : 08.05.1997
   Purpose:
   Version: $Id: GenCS.ti,v 1.5 1997/08/03 15:19:20 kay Exp $
*)
import
  socket

export
  connect(host :String  service :String  talk(:socket.T) :Ok) :Ok
  (* connect to a service *)

  serv(service :String  talk(:socket.T) :Ok  pure :Bool) :Ok
  (* create a service *)

  terminate(host :String  service :String) :Ok
  (* terminate a service *)

  talk(ReqMsg <: Ok  RepMsg <: Ok
       reqMsg :ReqMsg
       var repMsg :RepMsg
       reqMsgStr :String
       repMsgStr(:RepMsg) :String
       send(s :socket.T  :ReqMsg) :Ok
       receive(s :socket.T) :RepMsg   )
      (sock :socket.T) :Ok
  (* send request and receive reply *)
end;
```

A.5 Archive.ti

```
interface Archive
(* System : Workenv
   File   : Archive.ti
   Author : Kay Ramme
   Date   : 30.01.1997
   Purpose: versioning of objects
   Version: $Id: Archive.ti,v 1.7 1997/09/11 13:52:59 kay Exp $
*)
import
  dynArray
  nSet
  iter

export
  Let Version = Array(Int)
  default :Version

(*---for testing only---*)
  (* versions *)
  versionNew() :Version

  versionNext(ver :Version  branch :Int) :Version

  versionPrev(ver :Version) :Version

  versionString(ver :Version) :String

  versionEqual(op1 :Version  op2 :Version) :Bool

  versionDerived(op1 :Version  op2 :Version) :Bool

(*---for testing only---*)
  T(A <: Ok) <: Ok

  new(A <: Ok  a :A
      cmt :String
      merge(:A :A) :A
      equal(:A :A) :Bool
      setVerFun(:A  :Fun() :Version) :Ok
      dup(:A) :A) :T(A)
  (* instantiate archive *)

  dup(A <: Ok  ar :T(A)) :T(A)
  (* duplicate archive *)

  getDefault(A <: Ok  :T(A)) :Version
  (* get basis for unqualified checkIn *)
```

```
  setDefault(A <: Ok :T(A) version :Version) :Ok
  (* set basis for unqualified checkIn *)

  extract(A <: Ok :T(A) version :Version) :A
  (* get obj of version *)

  checkIn(A <: Ok :T(A) basis :Version a :A cmt :String) :Version
  (* check in a descendent of version basis and get version *)

  checkInAsMerge(A <: Ok :T(A) basis :Version merge :Version a :A cmt :String) :Version
  (* merge two versions to a new version *)

  aprint(A <: Ok :T(A)) :Ok
  (* print archive *)

  derived(A <: Ok op1 :T(A) op2 :T(A)) :Bool
  (* is op2 derived of op1 ? *)

  merge(A <: Ok :T(A) :T(A)) :T(A)
  (* merge two archives *)

  collectVersions(A <: Ok t :T(A)) :iter.T(Version)
  (* collect all versions in archive *)
end;
```

A.6 Workspace.ti

```
interface Workspace
(* System : Workenv
   File   : Workspace.ti
   Author : Kay Ramme
   Date   : 12.02.1997
   Purpose:
   Version: $Id: Workspace.ti,v 1.5 1997/08/03 15:13:41 kay Exp $
*)
import

export
  Let M(A <: Ok) = Tuple
    merge(:A  :A) :A
    derived(:A  :A) :Bool
    dup(:A) :A
    fmt(:A) :String
    send(A <: Ok  s :socket.T  a :A) :Ok
    receive(A <: Ok  s :socket.T) :A
  end

  T <:Oper(A <:Ok  Mt <: M(A)) Ok


  new(A <:Ok  Mt <: M(A)  e :Mt  wsName :String) :T(A Mt)
  (* instantiate a workspace *)

  (* locking *)
  tryLock(A <: Ok  Mt <: M(A)  :T(A Mt)  var comment :String) :Bool
  unlock(A <: Ok  Mt <: M(A)  :T(A Mt)) :Ok


  delete(A <:Ok  Mt <: M(A)  t :T(A Mt)) :Ok
  (* destroy workspace *)

  newProxy(A <:Ok  Mt <: M(A)  host :String  name :String  e :Mt) :T(A Mt)
  (* instantiate a proxy *)


  (* hierachie *)
  getParent(A <: Ok  Mt <: M(A)  :T(A Mt)):T(A Mt)
  (* get parent workspace *)

  setParent(A <: Ok  Mt <: M(A)  :T(A Mt)  parent :T(A Mt)) :Ok
  (* set parent workspace *)

  getMethodTable(A <: Ok  Mt <: M(A)  :T(A Mt)) :Mt
  (* get methode table *)
```

```
(* access *)
get(A <: Ok  Mt <: M(A)  :T(A Mt)  name :String  bo :Bool) :A
(* get an object, search this workspace and backing workspaces *)

register(A <: Ok  Mt <: M(A)  :T(A Mt)  name :String  a :A) :Ok
(* add or update an object to this workspace *)

release(A <: Ok  Mt <: M(A)  :T(A Mt)  name :String) :Ok
(* release a previously registered object *)


(* transfer *)
bringOver(A <:Ok  Mt <: M(A)  :T(A Mt)  comment :String  nms :Array(String)) :Ok
(* bringover locally registered objects from parent workspace and merge *)

putBack(A <:Ok  Mt <: M(A)  :T(A Mt)  comment :String  nms :Array(String)) :Ok
(* putback locally registered objects to parent workspace, reject if conflict *)


getNames(A <: Ok  Mt <: M(A)  :T(A Mt)) :Array(String)
(* get local namespace *)

(* misc *)
print(A<:Ok  Mt <: M(A)  :T(A Mt)) :Ok
(* list names of all objects, search this and backing workspaces *)

end;
```

A.7 PCompiler.ti

```
interface PCompiler
(* System : Workenv
   File   : PCompiler.ti
   Author : Kay Ramme
   Date   : 26.06.1997
   Purpose:
   Version: $Id: PCompiler.ti,v 1.2 1997/09/11 14:02:29 kay Exp $
*)
import
(*  tlCompiler *)

export
  T <: Ok
  Binding <: Ok

(*  Let T = tlCompiler.Context *)

  c :T

  reset() :Ok
  (* reset module and create a new compiler *)

  new() :T
  (* create a new compiler *)

  compile(:T  src :String) :Ok
  (* compile a string *)

  getBinding(t :T  name :String) :Binding

  bindingObj(R <: Ok  binding :Binding) :R
  (* get an oid *)

  link(:T  binding :Binding) :Ok
  (* link binding to compiler *)
end;
```

A.8 ProgObj.ti

```
interface ProgObj
(* System : Workenv
   File   : ProgObj.ti
   Author : Kay Ramme
   Date   : 11.06.1997
   Purpose:
   Version: $Id: ProgObj.ti,v 1.6 1997/09/11 14:01:48 kay Exp $
*)
import
  tsp
  pCompiler

export
  T <: Ok

  Let VerName = Tuple
    ver  :Archive.Version
    name :String
  end

  Let Linkable = Tuple
    D <: Ok
    obj :D
    link(obj :D  c :pCompiler.T) :Ok
    getOid(obj :D) :tsp.OID
  end

  Let MethodeTable(R <: Ok) = Tuple
    obj   :Fun(:R) :Linkable
    getSrc   :Fun(:R) :String
    setSrc   :Fun(:R  :String) :Ok
    getVersionFun :Fun(:R) :Fun() :Archive.Version
    setVersionFun :Fun(:R  :Fun() :Archive.Version) :Ok
    dup   :Fun(:R) :R
    fit   :Fun(:R  :iter.T(Archive.Version)):Archive.Version
  end

  new(R <: Ok  mt :MethodeTable(R)  obj :R) :T
  (* instantiate new progam object *)

  getLinkable(:T) :Linkable
  (* get linkable *)

  getSrc(:T) :String
  (* get source of progObj *)

  getVersionFun(:T) :Fun() :Archive.Version
```

```
  (* get version of progObj *)

  setSrc(:T  src :String) :Ok
  (* set source of progObj *)

  setVersionFun(:T  verFun() :Archive.Version) :Ok
  (* set version of progObj *)

  dup(:T) :T
  (* duplicate progObj *)

  merge(op1 :T  op2 :T) :T
  (* merge two progObjects *)

  equal(op1 :T  op2 :T) :Bool
  (* equal ? *)

  fit(:T  versions :iter.T(Archive.Version)) :Archive.Version
  (* select best fitting object, raise exception if non fits *)
end;
```

A.9 PInterface.ti

```
interface PInterface
(* System : Workenv
   File   : PInterface.ti
   Author : Kay Ramme
   Date   : 26.06.1997
   Purpose:
   Version: $Id: PInterface.ti,v 1.4 1997/09/11 14:02:13 kay Exp $
*)
import
  tsp
  pCompiler
  :Archive
  progObj :ProgObj

export

  T <: Ok

  new(src      :String
    itfName :String
    deps     :Array(progObj.VerName)
    fit      :Fun(:T  :iter.T(Archive.Version)) :Archive.Version) :T
  (* create a new interface *)

  getLinkable(:T  getProg(:progObj.VerName) :progObj.Linkable) :progObj.Linkable
  (* get oid of compiled interface,
     this does not work -> raises an exception *)

  getSrc(:T) :String
  (* get source code of interface *)

  setSrc(:T  src :String) :Ok
  (* set source code of interface *)

  getVerFun(:T) :Fun() :Archive.Version
  (* get version function *)

  setVerFun(:T  verFun() :Archive.Version) :Ok
  (* set version function *)

  dup(:T) :T
  (* duplicate interface *)

  fit(:T  versions :iter.T(Archive.Version)) :Archive.Version
  (* select best fitting compiled object or raise exception if non fits *)

 methTable(getProg(:progObj.VerName) :progObj.Linkable) :progObj.MethodeTable(T)
```

```
  (* create a methode table *)

end;
```

A.10 PModule.ti

```
interface PModule
(* System : Workenv
   File   : PModule.ti
   Author : Kay Ramme
   Date   : 26.06.1997
   Purpose:
   Version: $Id: PModule.ti,v 1.4 1997/09/11 14:02:01 kay Exp $
*)
import
  pCompiler
  :Archive
  progObj :ProgObj

export

  T(R <: Ok) <: Ok

  new(R <: Ok
    src     :String
    modName :String
    itf     :ProgObj.VerName
    deps    :Array(progObj.VerName)
    fit     :Fun(:T(R)  :iter.T(Archive.Version)) :Archive.Version) :T(R)
  (* create a new module *)

 getLinkable(R<: Ok :T(R) getProg(:progObj.VerName) :progObj.Linkable) :progObj.Linkable
  (* get linkable module *)

  getSrc(R <: Ok :T(R)) :String
  (* get source code of module *)

  setSrc(R <: Ok :T(R) src :String) :Ok
  (* set source code of module *)

  getVerFun(R <: Ok :T(R)) :Fun() :Archive.Version
  (* get version function *)

  setVerFun(R <: Ok :T(R) verFun() :Archive.Version) :Ok
  (* set version function *)

  dup(R <: Ok :T(R)) :T(R)
  (* duplicate module *)

  fit(R <: Ok :T(R) versions :iter.T(Archive.Version)) :Archive.Version
  (* select best fitting compiled object or raise exception if non fits *)

 methTable(R <: Ok getProg(:progObj.VerName) :progObj.Linkable) :progObj.MethodeTable(T(R))
```

```
  (* create a methode table *)
end;
```

A.11 WorkItf.ti

```
interface WorkItf
(* System : Workenv
   File   : WorkItf.ti
   Author : Kay Ramme
   Date   : 07.05.1997
   Purpose:
   Version: $Id: WorkItf.ti,v 1.6 1997/09/11 14:00:42 kay Exp $
*)
import
  :Archive
  rebMan
  progObj

export

  T <: Ok

  newMaster(thisStore :String) :T
  (* instantiate an working enviroment *)

  newClient(masterHost :String  masterStore :String  thisStore :String) :T
  (* instantiate an working enviroment *)

  (* manage program objects *)
  bringOver(:T  nms :Array(String)) :Ok
  (* bringover interfaces and modules from parent *)

  putBack(:T  nms :Array(String)) :Ok
  (* putback interfaces and modules to parent *)


  registerProgObj(t :T  pobj :progObj.T  name :String) :Ok
  (* add a progObj to the context *)

  registerModule(:T
    modName :String
    itf :ProgObj.VerName
    src :String
    fit(R <: Ok  :pModule.T(R)  versions :iter.T(Archive.Version)) :Archive.Version
    deps :Array(progObj.VerName)) :Ok
  (* add a module to the context *)

  registerInterface(:T
    itfName :String
    src :String
    fit(:pInterface.T  verions :iter.T(Archive.Version)) :Archive.Version
    deps :Array(progObj.VerName)) :Ok
```

```
(* add an interface to the context *)


importObject(t :T  ver :Archive.Version  name :String) :Ok
(* import object <name> into toplevel *)

getSourceCode(:T  name :String  ver :Archive.Version) :String
(* get source of a progObj *)

checkInSourceCode(:T  name :String  src :String  ver :Archive.Version  cmt :String) :Ok
(* check in source *)


update(O <: Ok  :T  obj :O) :O
(* update references to module *)

printArchive(:T  name :String) :Ok
(* print archive of <name> *)

(* manage ubiquitions *)
registerUbiquitousObj(O <: Ok (*dynamisch?*)  :T  obj :O  name :String) :Ok
getUbiquitousObj(O <: Ok (*dynamisch?*)  :T  name :String) :O
removeUbiquitousObj(:T  name :String) :Ok

remoteExecute(R <: Ok  :T  host :String  store :String  f(:T):R) :R
(* remote function excution *)


Let LibObj = Tuple
case Interface with
  name :String
  deps :Array(String)

case InterfaceSource with
  name :String
  source :String
  deps :Array(String)

case Module with
  name :String
  interfaceName :String
  deps :Array(String)

case ModuleSource with
  name :String
  source :String
  interfaceName :String
  deps :Array(String)
```

```
  case ModuleReflect with
    name :String
    interfaceName :String
    deps :Array(String)
  end

  Let Library = Tuple path :String  libObjs :Array(LibObj) end
  registerLibrary(:T  lib :Library) :Ok
  (* register a libraray at workspace *)

  listNames(:T) :Ok
  (* list all names in namespace, inclusive backing spaces *)

  listObjectCache(:T) :Ok
  (* list objects in progObj cache *)

  listUbiCache(:T) :Ok
  (* list objects in ubiquitous object cache *)

  getRebMan(t :T) :rebMan.T(T)
  (* access rebind manager *)
end;
```

A.12 UxFileWrp.ti

```
interface UxFileWrp
(* System : workenv
   File   : UxFileWrp.ti
   Author : Kay Ramme
   Date   : 01.06.97
   Purpose: Wrapper for UxFile
   Version: $Id: UxFileWrp.ti,v 1.3 1997/06/17 20:32:35 kay Exp $
*)

import
  workItf

export

  T <: Ok
  (* A file descriptor *)

  Flags, Mode, Base <:Ok

  readonly, writeonly, readwrite, create, truncate :Flags

  defaultMode :Mode

  bof, current, eof :Base
  (* Begin Of File, Current Positions, End Of File. *)


(* — Standard Unix file operations ————————————— *)

  openF(:workItf.T path :String  flags :Flags  mode :Mode) :T
  (* Open a unix file and return file descriptor. *)

  close(file :T) :Ok
  (* Close a unix file. *)

  read(file :T  buff :String  size :Int) :Int
  (* size <= size(buff) *)

  write(file :T  buff :String  size :Int) :Int
  (* *)

  lseek(file :T  offset :Int  base :Base) :Int
  (* Set the file pointer according to value supplied for base. Return
     the new position. *)

  tell(file :T) :Int
  (* Return the file pointer position. Equivalent to lseek(file 0 bof). *)
```

```
(* — Abbreviations ———————————————————— *)

  seek(file :T  pos :Int) :Ok
  (* Set the file pointer absolute to pos. *)

  size(file :T) :Int
  (* Return the size of file without change the file pointer position. *)

  pos(file :T) :Int
  (* Return the position of the file pointer. *)


(* — for debugging only — *)
  Dcp <: Ok

  fileToDcp(:T) :Dcp
  dcpToFile(:Dcp  t :workItf.T) :T

end;
```

Anhang B

Beispiele und Demos

B.1 master.tyc

```
(*
** instantiate a master workspace and load some librarys
** 27.08.97  Kay Ramme
** file: $TYCOON_ROOT/src/workenv/demos/setupMaster.tyc
*)

log.setEnterLeaveState(log.notLogState);
log.logFunction("misc" "receive" log.notLogState);
log.logFunction("misc" "send" log.notLogState);
log.logModule("rebind" log.notLogState);
log.logModule("pCompiler" log.notLogState);


let context = workItf.newMaster("MasterStore");
(* create a new context for master workspace *)

do load "workenv/demos/setWrapper.tyc";
(* define methode wrappers for context *)

do load "workenv/demos/registerLibs.tyc";
(* register librarys *)
```

B.2 clientA.tyc

```
(*
** instantiate client workspace A
** 27.08.97  Kay Ramme
** file: $TYCOON_ROOT/src/workenv/demos/setupClientA.tyc
*)

log.setEnterLeaveState(log.notLogState);
log.logFunction("misc" "receive" log.notLogState);
log.logFunction("misc" "send" log.notLogState);
log.logModule("rebind" log.notLogState);
log.logModule("pCompiler" log.notLogState);

let masterHost = "dbis11.informatik.uni-hamburg.de";
let masterName = "MasterStore";
let clientName = "ClientA"
let context = workItf.newClient(masterHost masterName clientName);
(*workItf.regDirObj(context "WORKITF" workItf);*)
(* create a new context for client workspace *)

do load "workenv/demos/setWrapper.tyc";
(* define methode wrappers for context *)

do load "workenv/demos/registerLibs.tyc";
(* register librarys *)
```

B.3 clientB.tyc

```
(*
** instantiate client workspace B
** 11.09.97 Kay Ramme
** file: $TYCOON_ROOT/src/workenv/demos/setupClientA.tyc
*)

log.setEnterLeaveState(log.notLogState);
log.logFunction("misc" "receive" log.notLogState);
log.logFunction("misc" "send" log.notLogState);
log.logModule("rebind" log.notLogState);
log.logModule("pCompiler" log.notLogState);

let masterHost = "dbis11.informatik.uni-hamburg.de";
let masterName = "MasterStore";
let clientName = "ClientB"
let context = workItf.newClient(masterHost masterName clientName);
(*workItf.regDirObj(context "WORKITF" workItf);*)
(* create a new context for client workspace *)

do load "workenv/demos/setWrapper.tyc";
(* define methode wrappers for context *)

do load "workenv/demos/registerLibs.tyc";
(* register librarys *)
```

B.4 remoteExecuteDemo.tyc

```
(*
** remote execution demonstration
**
** 27.08.1997   Kay Ramme
**
** Ziel:
**   Diese Demo zeigt, daß das Ergebnis der
**   Ausführung einer Function von den Versionen
**   der eingebundenen Module abhängt.
**
** Es sind ein Master und ein Client Workspace zu
** erzeugen. Innerhalb des Masters sind folgende
** Kommandos auszuführen:
*)

(*
** Es wird Sourcecode eines Interfaces "Test" und zweier
** Implementationen dieses Interfaces erzeugt.
*)

let testInterfaceSource = "interface Test export a :Int end;";
(* definiere den Quelltext für Interface "Test" *)

let testInterfaceFitFun = fun(:pInterface.T vers :iter.T(Archive.Version)) :Archive.Version
  array 1 end;
(* Interface "Test" ist nur zu sich selbst kompatibel *)

let testInterfaceDeps   = array end;
(* Interface "Test" hängt von keinem anderen Programmobjekt ab *)


registerInterface("Test"
  testInterfaceSource
  testInterfaceFitFun
  testInterfaceDeps
);
(* registriere Interface "Test" *)


let testModuleSource_V1 = "module test export let a = 27 end;";
(* Quelltext des Moduls "test" in der Version 1 *)

let testModuleInterface = tuple array 1 end "Test" end;
(* Modul "test" paß t zu Interface "Test" in der Version 1 *)

let testModuleFitFun(R <: Ok
  :pModule.T(R)
```

```
  versions :iter.T(Archive.Version)) :Archive.Version =
begin
  let var ver = array 0 end

  iter.forEach(versions fun(v :archive.Version) :Ok begin
    print.string("testModuleFileFun: " <> archive.versionString(v) <> "\n")
    if v[0] > ver[0] then
      ver := v
    end
  end)

  ver
end;
(* wähle immer die neueste vorhandene Version von "test" *)

let testModuleDeps = array end;
(* Modul test hängt von nichts ab, auß er seinem Interface *)

registerModule("test"
  testModuleInterface
  testModuleSource_V1
  testModuleFitFun
  testModuleDeps
);
(* registriere Modul "test" *)


let testModuleSource_V2 = "module test export let a = 35 end;";
(* Quelltext des Moduls "test" in der Version 2 *)

let testModule_V2_basis = array 1 end;
(* Version 2 von "test" basiert auf Version 1 *)

let testModule_V2_comment = "zweite Version";
(* Kommentar für Version 2 *)

checkInSourceCode("test"
  testModuleSource_V2
  testModule_V2_basis
  testModule_V2_comment
);
(* registriere die zweite Version von "test" *)


putBack(array end);

(*
** Linken der unterschiedlichen Module:
** "test" Version 1 -> master toplevel
```

```
** "test" Version 2 -> client toplevel
*)

importObject(array 1 end "test");
(* Linken des Moduls "test" in der Version 1 in den aktuellen TopLevel.
   Nach Bedarf wird vorher compiliert. *)


remoteExecute("localhost" "ClientA" fun(c :workItf.T) :Ok begin
  workItf.importObject(c array 2 end "test")
end);
(* Linken des Moduls "test" in der Version 2 in den Toplevel des
   Clients A. Bei Bedarf wird compiliert.
   Dieses Modul befindet sich anschließ end im Cache. *)



(*
** Definition einer Funktion f,
** die einmal lokal und einmal entfernt
** ausgeführt wird.
*)

let f = fun(:workItf.T) begin
  test.a
end;
(* Funktion f gibt den Wert des Objektes a des Moduls "test"
   zurück. *)


f(context);
(* f wird lokal ausgeführt, das Ergebnis ist 27. *)

remoteExecute("localhost" "ClientA" f);
(* f wird remote ausgeführt, das Ergebnis ist 35,
   da "test" Version 2 sich im Cache befand und zum Rebinden
   verwendet wurde. *)
```

B.5 mobileDemo.tyc

```
(*
** persist demo:
**
** Ein Datei-Handle wird pseudo mobile gemacht und
** migriert anschließ end mit einer Funktion in einen
** anderen Workspace.
** Im Zielworkspace wird der Inhalt der Datei ausgelesen
** und ausgegeben.
** Anschliessend wird die Datei geschlossen.
**
** 27.08.1997   Kay Ramme
*)


let hd = uxFileWrp.openF(context "test.txt" uxFileWrp.readonly uxFileWrp.defaultMode);

let f(:workItf.T) = begin
  let str = string.new(512 ' ')
  uxFileWrp.read(hd str 512)
  print.string("readed string: " <> str <> "\n")
end;

remoteExecute("localhost" "ClientA" f);

uxFileWrp.close(hd);
```

B.6 updateDemo.tyc

```
(*
** function update demonstration
**
** 04.11.1997  Kay Ramme
**
** Ziel:
**  Diese Demo zeigt, daß das Ergebnis der
**  Ausführung einer Function von den Versionen
**  der eingebundenen Module abhängt.
**
*)

(*
** Es wird Sourcecode eines Interfaces "Test_update" und zweier
** Implementationen dieses Interfaces erzeugt.
*)

let test_updateInterfaceSource = "interface Test_update export a :Int end;";
(* definiere den Quelltext für Interface "Test_update" *)

let test_updateInterfaceFitFun = fun(:pInterface.T vers :iter.T(Archive.Version)) :Archive.Version
  array 1 end;
(* Interface "Test_update" ist nur zu sich selbst kompatible *)

let test_updateInterfaceDeps   = array end;
(* Interface "Test_update" hängt von keinem anderen Programmobjekt ab *)


registerInterface("Test_update"
  test_updateInterfaceSource
  test_updateInterfaceFitFun
  test_updateInterfaceDeps
);
(* registriere Interface "Test_update" *)


let test_updateModuleSource_V1 = "module test_update export let a = 27 end;";
(* Quelltext des Moduls "test_update" in der Version 1 *)

let test_updateModuleInterface = tuple array 1 end "Test_update" end;
(* Modul "test_update" paß t zu Interface "Test_update" in der Version 1 *)

let test_updateModuleFitFun(R <: Ok
  :pModule.T(R)
  versions :iter.T(Archive.Version)) :Archive.Version =
begin
  let var ver = array 0 end
```

```
  iter.forEach(versions fun(v :archive.Version) :Ok begin
    print.string("test_updateModuleFitFun: " <> archive.versionString(v) <> "\n")
    if v[0] > ver[0] then
      ver := v
    end
  end)

  ver
end;
(* wähle immer die neueste vorhandene Version von "test_update" *)

let test_updateModuleDeps = array end;
(* Modul test_update hängt von nichts ab, auß er seinem Interface *)

registerModule("test_update"
  test_updateModuleInterface
  test_updateModuleSource_V1
  test_updateModuleFitFun
  test_updateModuleDeps
);
(* registriere Modul "test_update" *)


let test_updateModuleSource_V2 = "module test_update export let a = 35 end;";
(* Quelltext des Moduls "test_update" in der Version 2 *)

let test_updateModule_V2_basis = array 1 end;
(* Version 2 von "test_update" basiert auf Version 1 *)

let test_updateModule_V2_comment = "zweite Version";
(* Kommentar für Version 2 *)

checkInSourceCode("test_update"
  test_updateModuleSource_V2
  test_updateModule_V2_basis
  test_updateModule_V2_comment
);
(* registriere die zweite Version von "test_update" *)




importObject(array 1 end "test_update");
(* Linken des Moduls "test_update" in der Version 1 in den TopLevel.
   Bei Bedarf wird vorher compiliert. *)


(* Definition einer Funktion f. *)
```

```
let f = fun(:workItf.T) begin
  test_update.a
end;
(* Funktion f gibt den Wert des Objektes a des Moduls "test_update" zurück. *)


f(context);
(* f ist an "test_update" Version 1 gebunden, das Ergebnis ist 27. *)


update(f);
(* f wird aktualisiert und somit an "test_update" Version 2 gebunden. *)

f(context);
(* Das Ergebnis ist nun 35 *)
```

Anhang C

Versionsgraphenbeispiele

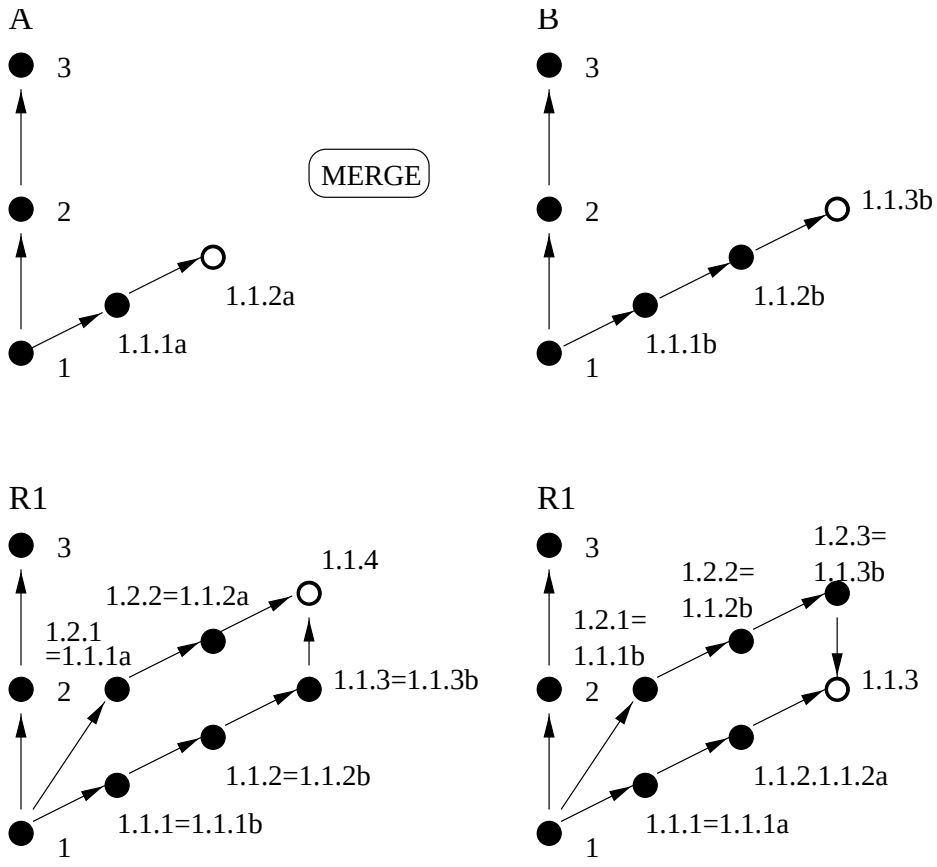

Abbildung C.1: Verschmelzung von Versionsgraphen mit Konflikt

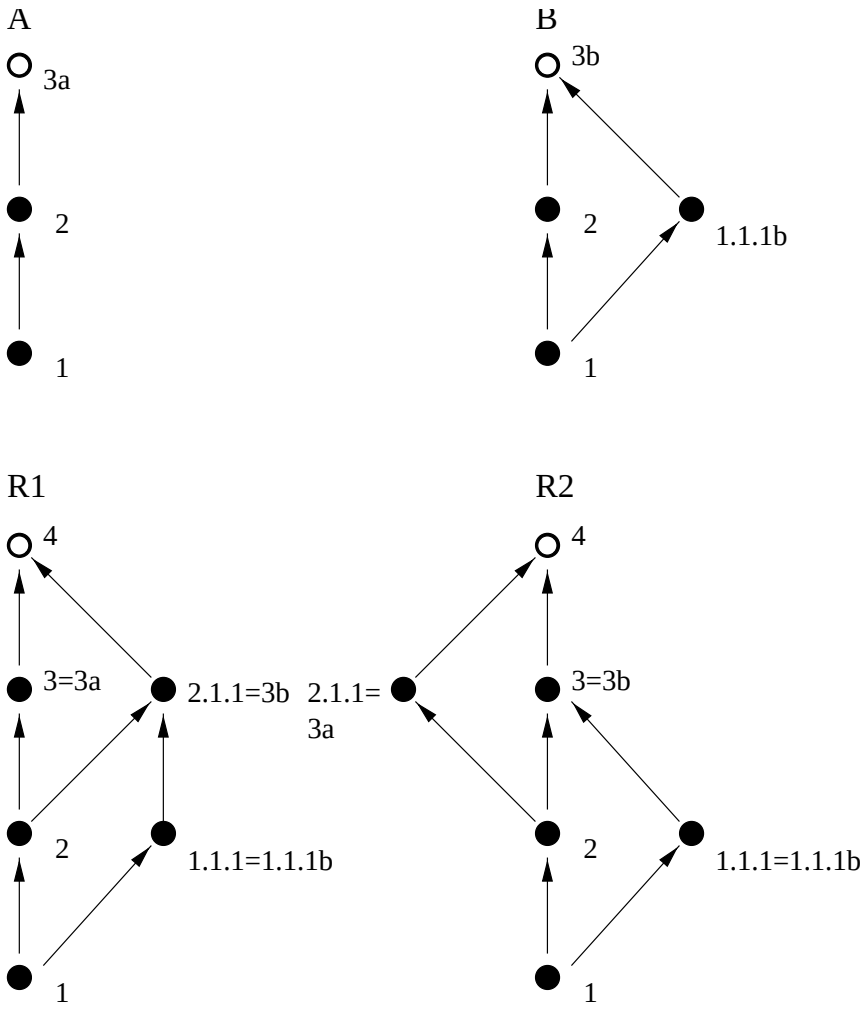

Abbildung C.2: Verschmelzung von Versionsgraphen mit Konflikt

Literaturverzeichnis

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau.* Addison-Wesley (Deutschland) GmbH, 1988.

[BB95] D. Bolinger and T. Bronson. *Applying RCS and SCCS, From Source Control to Project Control.* O Reilly, 1995.

[BM92] Alfred Leonard Brown and R. Morrison. A generic persistent object store. *Software Engineering Journal 7, 2 pp 161-168*, 1992.

[Bre96] Gerd Bremer. Typüberprüfung in polymorphen Programmiersprachen – Aufgaben und Lösungsansätze. Master's thesis, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, August 1996.

[Bro88] Alfred Leonard Brown. *Persistent Object Stores.* PhD thesis, Department of Computational Science, University of St. Andrews, 1988.

[CLZ92] Alberto Coen, Luigi Lavazza, and Roberto Zicari. Assuring type-safety of object oriented languages. Technical Report 4/92, Universität des Saarlandes, 1992.

[CMA94] L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing, pages 11–31. Springer-Verlag, Berlin u.a., February 1994.

[Det89] Reinhard Detering. *UNIX Handbuch System V.* Sybex, 1989.

[DI393] *Duden Informatik: Sachlexikon für Studium und Praxis.* Bibliographisches Institut & F.A. Brockhaus AG, Mannheim; Leipzig; Wien; Zürich, 1993.

[Fla97] David Flanagan. *Java, in a Nutshell, Second Edition.* O'Reilly, 1997.

[Hei86] R. Heigenmoser. *Das C-Anwender Handbuch.* Hofacker, 1986.

[Hen90] Andreas V. Hense. Polymorphic type inference for a simple object oriented programming language with state. Technical Report Nr. A 20/90, Universität des Saarlandes, 1990.

[Hey92] Michael L. Heytens. The design and implementation of a parallel persistent object system. Technical Report MIT/LCS/TR-529, Massachusetts Institute of Technology, February 1992.

[Joh97] Nico Johanisson. Eine Umgebung für mobile Agenten: Agentenbasierte verteilte Datenbanken am Beispiel der Kopplung autonomer Internet Web Site Profiler. Master's thesis, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, 1997.

[JV86] E. Jessen and R. Valk. *Rechensysteme, Grundlagen der Modellbildung.* Springer-Verlag Berlin Heidelberg New York, 1986.

[KBM+89] H. Kerner, G. Bruckner, Th. Mayr, S. Vogel, and F. Vogt. *Rechnernetze nach ISO-OSI, CCITT.* H. Kerner Taborstraße 23, A-3012 Wolfsgraben, 1989.

[KCMS96] G.N.C. Kirby, R.C.H. Connor, R. Morrison, and D. Stemple. Using reflection to support type-safe evolution in persistent systems. Technical report, University of St. Andrews, 1996.

[Koc97] Christian Koch. Entwicklungsunterstützung für persistente strukturierte Systeme. Master's thesis, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, 1997.

[Kur97] Jürgen Kuri. Der mit dem Wolf tanzt..., Ausfallsicherheit für Server durch Clustering. *C'T Magazin für Computertechnik*, 1997.

[LPJ+94] Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus, and Adrian Nye. *Managing Internet Information Services.* O'Reilly, 1994.

[LS87] P.C. Lockemann and J.W. Schmidt, editors. *Datenbankhandbuch.* Springer-Verlag, Berlin u.a., 1987.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, Berlin u.a., 1993.

[Mat96] B. Mathiske. *Mobilität in persistenten Objektsystemen.* PhD thesis, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, 1996.

[Mey90] B. Meyer. *Objektorientierte Softwareentwicklung.* Prentice-Hall International Inc., London, 1990.

[MM94] Florian Matthes and Sven Müßig. The tycoon language tl: An introduction. Technical Report 112-93, DBIS Tycoon Report, December 1994.

[MMM93] B. Mathiske, F. Matthes, and S. Müßig. The tycoon system and library manual. DBIS Tycoon Report 212-93, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, December 1993.

[MMS95a] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995. (Also appeared as TR FIDE/95/136).

[MMS95b] F. Matthes, R. Müller, and J.W. Schmidt. Towards a unified model of untyped object stores: Experience with the Tycoon store protocol. In M.P. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1995.

[MS94] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc, 1996.

[Sch94] Gerald Schröder. Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennungs-, Bindungs- und Typisierungsinvarianzen. Master's thesis, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, 1994.

[Sch97] Gerald Schröder. *Kooperierende Objektsysteme: Entwicklung und Betrieb*. PhD thesis, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, 1997. in Vorbereitung.

[SM92] Gerald Schröder and Florian Matthes. Using the tycoon compiler toolkit. Technical Report 061-92, DBIS Tycoon Report, June 1992.

[STS97] Gunter Saake, Can Türker, and Ingo Schmitt, editors. *Objektdatenbanken: Konzepte, Sprachen, Architekturen*. International Thomson Publishing GmbH, 1997.

[VP96] Nastaran Vaziri Pour. *Flexible Bindungstechniken für ubiquitäre Ressourcen in verteilten Anwendungen*. Studienarbeit, 1996.

[WMF] WMFC95. WWW Home Page of the Workflow Management Coalition. http://www.aiai.ed.ac.uk/WFMC/.

Erklärung

Ich versichere hiermit, die vorliegende Arbeit selbstständig und ausschließlich unter Zuhilfenahme der angegebenen Quellen durchgeführt zu haben.

Ort, Datum	Kay Ramme