Eine neue Architektur für Betriebssysteme

Thomas Schöbel-Theuer

23. März 2005

Vorwort

Dieses Buch richtet sich an Leser, die Grundlagen-Kenntnisse und möglichst auch praktische Erfahrungen im Bau von Betriebssystemen haben.

Wenn es von diesem speziellen fortgeschrittenen Leserkreis als Lehrbuch über Entwurfs-Methodik verstanden würde, dann hätte es seinen Zweck gut erfüllt.

Die hier vorgetragenen Denk- und Anschauungsweisen über Betriebssystem-Architekturen sind noch längst nicht vollständig verstanden, die Anwendungsbereiche noch längst nicht vollständig erkannt. Praktische Erfahrungen fehlen weitgehend. Dieses Werk soll daher auch Ansporn zu weiterer Forschung sein. Ein Fachgebiet kann nur dann langfristige Zukunfts-Perspektiven haben, wenn die Grundlagen, auf denen es aufbaut, möglichst weit tragen. Ich hoffe, zu diesen Grundlagen einen Beitrag geleistet zu haben.

Wenn dieses Buch auch in benachbarten Fachgebieten wie Datenbanken, Software-Engineering, Programmiersprachen, Rechnerarchitektur und Theoretischer Informatik Anklang finden, als Ideenlieferant dienen und weitere Forschungsarbeiten anregen würde, dann wäre es über seinen ursprünglichen Zweck hinaus gewachsen.

Zum Verständnis dieser Arbeit ist es hilfreich, die vorgegebene Kapitel-Reihenfolge beim Lesen im wesentlichen einzuhalten und lediglich Fußnoten zu überspringen, die überlesbare Seitenthemen aus dem Haupt-Lesefluss fern halten sollen. Zum Verständnis von Seitenthemen sollte der Leser diese bereits aus der Literatur kennen oder sich entsprechende Kenntnisse durch die zitierte Literatur aneignen. Ich verfolge nicht das Ziel einer die gesamte Literatur umfassenden Darstellung, da sonst der Umfang auf ein Mehrfaches aufgebläht worden wäre. Die eingehende Beschäftigung mit den Beispielen in Kapitel 1 ist zur Annäherung an die hier vorgestellte Denkweise zu empfehlen.

Ich habe mich bemüht, die neue deutsche Rechtschreibung zu verwenden. Alte Rechtschreibungs-Muster können dennoch gelegentlich auftauchen[1]. Der Leser wird um Verständnis gebeten.

Stuttgart, im Dezember 2002

Thomas Schöbel-Theuer

Vorwort zur überarbeiteten Fassung März 2005

Seit der ersten Fassung des Buchmanuskriptes hat sich etwas getan: die ursprünglich reinen Entwürfe der LEGO-artigen Betriebssystem-Architektur sind im Athomux-Projekt (`http://www.athomux.net`) prototypisch implementiert worden. Zum Zeitpunkt dieser Überarbeitung des Buches ist der Prototyp zwar noch in einem recht rudimentären Stadium im Vergleich zu ausgewachsenen Betriebssystemen, demonstriert jedoch bereits die Grundfunktionalität eines Betriebssystems durch Emulation der wichtigsten Posix-Systemaufrufe `open()`, `close()`, `read()`, `write()`, `stat()`, `fork()`, `exec()` und einiger weiterer elementarer Grundfunktionen, so dass eine sehr einfache Shell lauffähig ist. Weitere wesentliche Konzepte dieses Buches wie instanzorientierte Sichten auf mehrere im Netzwerk zusammengeschaltete Rechner sind in der Erprobungsphase; ein proof of concept ist damit bereits teilweise erfolgt und wird in Kürze weiter vervollständigt werden.

Aus dieser Implementierung habe ich weitere Erfahrungen gewonnen und weitere Vereinfachungen der Architektur entdeckt, die in die nunmehr vorliegende Fassung zum großen Teil eingearbeitet worden sind. Dennoch geht dieses Buch kaum auf die konkrete Implementierung von Athomux ein, das ich nach wie vor als nur eine von vielen möglichen Implementierungen der hier vorgetragenen Konzepte und Ideen betrachte. Wer näheres zum sich rapide fortentwickelnden Athomux-System wissen will, sei auf die Webseiten des Athomux-Projektes `http://www.athomux.net` und die dort veröffentlichte Projektdokumentation verwiesen.

In diesem Buch geht es nach wie vor um diejenigen Konzepte und Ideen, deren Lebensdauer diejenige einer konkreten Implementierung wie Athomux bei weitem übersteigen sollten.

Ich hoffe, damit den Gewinn für diejenigen Leser gesteigert zu haben, die gerne auf der konzeptionellen Ebene (Problem-Raum) arbeiten und nach Anregungen für ihre eigenen Entwürfe suchen.

Stuttgart, im März 2005

Thomas Schöbel-Theuer

[1] Nach neuer Schreibung soll man „X ist wohl definiert“ statt „X ist wohldefiniert“ schreiben, was jedoch einen völlig anderen Sinn ergibt (ersteres stellt unterschwellig in Frage, ob X überhaupt definiert ist, was beinahe das Gegenteil der zweiten Aussage bedeutet). Bei Fachbegriffen habe ich daher an vielen Stellen die alte Schreibung absichtlich weiter verwendet.

Inhaltsverzeichnis

Vorwort 3

Zusammenfassung 9

1. Einführung 11
1.1. Zur Intention und Methodik dieser Arbeit . . . 11
1.2. Beispiel eines LEGO-artigen Betriebssystem-Aufbaus . . . 12

2. Entwurfs-Prinzipien 15
2.1. Generizität: Wahl der angemessenen Abstraktionsebene . . . 15
2.1.1. Parametrische Generizität . . . 15
2.1.2. Erweiterungs-Generizität . . . 16
2.1.3. Kompositorische Generizität . . . 16
2.1.4. Universelle Generizität . . . 17
2.1.5. Fragen der Anwendung von Generizität . . . 17
2.2. Trennung von Mechanismus / Strategie / Repräsentation . . . 19
2.3. Das Verantwortungs-Prinzip . . . 19
2.3.1. Kontrollflüsse . . . 20
2.3.2. Schnittstellen-Mechanismen . . . 20
2.3.3. Abgrenzung von Verantwortung durch Kapselung . . . 21
2.4. Zerlegungs- und Rekombinationsstrategien . . . 21
2.4.1. Zerlegung der zu lösenden Aufgaben in möglichst wenige universelle Elementarteile . . . 21
2.4.2. Orthogonalität und Rekombinierbarkeit von Elementaroperationen . . . 22
2.5. Das Problem der „Eier legenden Wollmilchsau“ . . . 22
2.6. Automatismen . . . 22
2.7. Zugriffsrechte und Schutzmechanismen . . . 23
2.7.1. Grundlegende Betrachtung: Mandate . . . 23
2.7.2. Spezifikation von Schutzrechten . . . 24
2.7.3. Prüfung von Schutzrechten . . . 24
2.7.4. Sicherstellung von Schutzrechten . . . 25
2.8. Performanz-Fragen . . . 25
2.8.1. Performanz von Elementaroperationen . . . 25
2.8.2. Zero-Copy-Architektur . . . 25
2.8.3. Das Background-IO-Konzept . . . 26

3. Nester 29
3.1. Arten von Nestern . . . 29
3.2. Zugriffs-Modelle . . . 29
3.3. Elementaroperationen auf statischen und dynamischen Nestern . . . 31
3.3.1. `transfer` . . . 33
3.3.2. `wait` . . . 33
3.3.3. `get` . . . 33
3.3.4. `put` . . . 35
3.3.5. `lock` und `unlock` . . . 35
3.3.6. Freiwillige Selbstkontrolle des Verhaltens . . . 37
3.3.7. `get_maxlen` und `set_maxlen` . . . 37
3.3.8. Stream-IO, Pipes, Sockets und Speicherverwaltung . . . 38
3.4. Elementaroperationen auf dynamischen Nestern . . . 39
3.4.1. `get_map` . . . 39
3.4.2. `clear` . . . 40
3.4.3. `delete` . . . 40
3.4.4. `move` . . . 40
3.4.5. `get_meta` . . . 41

4. Bausteine **43**
4.1. Beispiel-Baustein-Arten . . . 43
4.1.1. `device_*` . . . 44
4.1.2. `buffer` . . . 44
4.1.3. `map_*` . . . 45
4.1.4. `selector` . . . 47
4.1.5. `dir_*` . . . 47
4.1.6. `union` . . . 49
4.1.7. `mmu_*` . . . 49
4.1.8. `adaptor_*` . . . 50
4.1.8.1. Adaption zwischen verschiedenen Zugriffs-Modellen . . . 50
4.1.8.2. Adaption zwischen verschiedenen `transfer_size`-Werten . . . 51
4.1.9. `redirect` . . . 53
4.1.10. `cow` . . . 53
4.1.11. `transaction` . . . 53
4.1.11.1. Sequentielle Transaktionen . . . 54
4.1.11.2. Parallele Transaktionen . . . 54
4.1.12. `remote` . . . 54
4.1.13. `mirror` . . . 55
4.1.14. `pipe` . . . 56
4.2. Instantiierung von Bausteinen . . . 56
4.2.1. Der Mechanismus . . . 56
4.2.2. Einige mögliche grundlegende Strategien . . . 57
4.3. Lösung von Infrastruktur-Aufgaben . . . 59
4.3.1. Kontrollfluss-Implementierung . . . 59
4.3.2. Reflektoren . . . 60
4.3.3. Anbindung an Hardware und Unterbrechungen . . . 61

5. Callbacks durch `notify`-Operationen **63**
5.1. Eigentums- und Besitzverhältnisse . . . 63
5.2. Rückforderung von Eigentum . . . 64
5.3. Synchronisation zwischen Kopien: spekulative Locks . . . 65

6. Generische Operationen **69**
6.1. Beispiel-Typsysteme . . . 70
6.1.1. Minimal-Typsystem . . . 70
6.1.2. Einfaches erweiterbares Typsystem . . . 71
6.2. Zusammenhang mit der Objektorientierung . . . 73
6.2.1. Simulation der Vererbung durch Bausteine . . . 73
6.2.2. Grundlegende Unterschiede zur Objektorientierung . . . 73
6.3. Zusammenhang mit der Aspektorientierung (AOP) . . . 75
6.4. Zusammenhang mit dem Funktionalen Paradigma . . . 76
6.5. Zusammenhang mit Datenbank-Schemata . . . 76

7. Instanz-Orientierung **79**
7.1. Historischer Rückblick . . . 79
7.2. Übertragung nach OO und AOP . . . 80

8. Multiversion-Modelle **83**
8.1. Anforderungen . . . 83
8.2. Allgemeines Modell . . . 83
8.3. Zeitintervall-Modell . . . 84
8.4. Container-Operationen: Locks . . . 84
8.5. Kausale Abhängigkeiten . . . 86
8.6. Aktualitätsgrade . . . 86
8.7. Schnittstellen von Lock-Operationen . . . 87
8.7.1. Suchintervalle . . . 87
8.7.2. Lock-Operationen im Multiversion-Modell . . . 87
8.8. Emulation einiger bekannter Transaktions-Strategien . . . 89
8.8.1. Timestamp-Ordering-Protokolle . . . 90
8.8.2. Striktes 2-Phasen-Locking . . . 90
8.9. Weitere Ideen . . . 90
8.9.1. Weitere Arten von Serialisierbarkeit bzw. Konsistenzmodellen . . . 90

8.9.2. Automatische Re-Evaluation 91

9. Schlussbemerkungen **93**

A. Simulation von Vererbung durch Generizität **95**

B. Beispiel-Problem der Objektorientierung **97**

C. Ansätze zur Vermeidung von Wettrennen **99**

Danksagung **101**

Literaturverzeichnis **103**

Zusammenfassung

In dieser Arbeit wird erklärt, wie sich Familien von Betriebssystemen nach einem LEGO-artigen Baukasten-Prinzip aufbauen lassen, wobei nur zwei wesentliche Haupt-Abstraktionen benutzt werden: Nester und Bausteine.

Beide Abstraktionen sind universell und erfüllen die Funktionen mehrerer konventioneller Abstraktionen konventioneller Betriebssysteme; insbesondere ist kein Dateisystem im konventionellen Sinne mehr notwendig, sondern kann vollständig durch die Nest-Abstraktion und rekursive Instantiierung von Bausteinen ersetzt werden.

Die wenigen universellen Abstraktionen eignen sich prinzipiell auch zum Aufbau von verteilten Systemen, Netzwerk-Betriebssystemen, und Datenbanksystemen. Damit sind auch fortgeschrittene Ziele wie die nahtlose Integration von Betriebssystemen und Datenbanken anvisierbar.

Die dem Entwurf zugrunde liegenden Ideen und Überlegungen werden ausführlich dargestellt: verschiedene Arten von Generizität, insbesondere universelle Generizität, kompositorische Generizität, und erst an dritter Stelle die Erweiterungs-Generizität.

Nester und Bausteine lassen sich dynamisch zur Laufzeit instantiieren und zu komplexen Netzwerken verdrahten, mit denen ein sehr weites Anwendungs-Spektrum abgedeckt werden kann. Diese Dynamik auf Instanz-Ebene wird als Realisierung eines Programmier-Paradigmas beschrieben, das *Instanz-Orientierung* genannt wird; die Zusammenhänge mit und die Unterschiede zur Objektorientierung (OO) und zur Aspektorientierten Programmierung (AOP) werden herausgearbeitet.

Die vorgestellte Methodik schreibt keine speziellen Implementierungs-Paradigmen vor; beispielsweise sind sowohl Implementierungen durch Server-Prozesse, als auch Implementierungen durch Rollenwechsel von Benutzer-Prozessen in verschiedene Schutzbereiche oder Kern-Modi, als auch in Mischformen möglich. Verschiedene Speicher-Aufteilungs-Modelle wie Single-Address-Space, gesonderte Adressräume im Stile monolithischer Kerne, rekursive Schachtelung virtueller Maschinen, oder beliebige Mischformen lassen sich dynamisch zur Laufzeit konfigurieren.

1. Einführung

1.1. Zur Intention und Methodik dieser Arbeit

Nach allgemeiner Überzeugung sind die Grundlagen der Betriebssysteme praktisch vollständig bekannt und erforscht; Fortschritte entstehen nach dieser weit verbreitenen Ansicht lediglich durch den Wandel der zugrundeliegenden Hardware und durch neue Benutzer-Anforderungen. Daher konzentrieren sich fast alle derzeitigen Forschungen auf spezielle (Rand-)Gebiete; eine Notwendigkeit zum Hinterfragen der hauptsächlich in den 1970er Jahren erforschten Grundkonzepte wird kaum bis gar nicht gesehen.

Diese Arbeit unternimmt einen Versuch, die Grundlagen des Fachgebietes von Grund auf neu zu durchdenken und in einem anderen Lichte zu betrachten, um neue Erkenntnisse zu gewinnen und neue Lösungswege sowohl für Bekanntes als auch für noch Unbekanntes zu finden. Sie befasst sich fast ausschließlich mit dem Problem- und Entwurfs-Raum.

Vom Leser wird daher erwartet, dass er sich von den bekannten Vorurteilen frei machen kann, wie ein Betriebssystem aufgebaut sein sollte. Er sollte sich zumindest probehalber auf einen anderen (anfangs vielleicht seltsam oder unmöglich anmutenden) Standpunkt stellen können, dessen Bewertung erst nach längeren Betrachtungs-, Verständnis- und Evaluations-Phasen möglich sein wird, da der hier gemachte Versuch über weite Strecken einem grundlegenden Neusatz für das gesamte Fachgebiet gleichkommt. Es wird zu bedenken gegeben, dass sich die historische Entwicklung des Fachgebietes nicht in kürzester Zeit vollzogen hat und viel Arbeit und Mühe erfordert hat, die bei bei einem grundlegenden Neuansatz ebenfalls notwendig sein wird. Ein grundlegender Neuansatz wird auch nicht von einem einzelnen Autor in einer einzigen Monographie vollständig darstellbar und evaluierbar sein - der eigentliche Nutzen wird sich erst erweisen, wenn die Ideen von einer größeren Gemeinschaft aufgegriffen, umgesetzt und ergänzt werden. Dies wird mehrere Jahre erfordern.

Trotz des grundlegenden Neuansatzes wird dem fachkundigen Leser viel Bekanntes begegnen; gelegentlich wird es aber auf vollkommen andere Weise dargestellt sein als in den gewohnten Lehrbüchern.

Nicht nur Betriebssysteme, sondern auch Entwurfs- und Programmier-Konzepte und -Paradigmen, Programmiersprachen und Laufzeitsysteme, Architekturen von Anwendungsprogrammen und vieles andere folgen heute meist unhinterfragt einer Grundidee, die Dijkstra 1968 im THE-System [Dij68] erstmals ausführlich beschrieben und klar dargelegt hat: man baue ein System als eine *Schichtung abstrakter Maschinen* auf, wobei sich jede höhere Schicht auf die Operationen und Funktionen der darunter liegender Schichten abstützt und zu den bekannten Operationen und Funktionen neue hinzufügt (auch als *Zwiebelschalen-Modell* bekannt). Diese einfache Grundidee liegt nicht nur allen heutigen Betriebsystemen, sondern auch der Objektorientierung (OO) zugrunde und stellt mithin die Basis praktisch aller heute gängigen Software-Methodologien dar.

Die Hinzunahme neuer Operationen und Funktionen durch höhere Schichten wird heute praktisch unhinterfragt fast immer so *interpretiert* und in Entwürfe umgesetzt, dass *Erweiterungen der Funktionalität* meistens auch *Erweiterungen von Schnittstellen*[1] nach sich ziehen. Nach gängiger herrschender Meinung lässt sich die Erweiterung von Schnittstellen bei Hinzunahme neuer Anforderungen bzw Funktionen nur in den seltensten Fällen vermeiden. Diese als selbstverständlich betrachtete (bzw. gelegentlich auch unbewusste) Grundannahme wird in dieser Arbeit jedoch grundlegend hinterfragt werden. Das Dijkstrasche Modell wird dadurch nicht prinzipiell außer Kraft gesetzt, sondern ganz anders interpretiert.

In dieser Arbeit wird der Frage nachgegangen, wie sich komplexe Software-Systeme mit noch teilweise *unbekannten*[2] (zukünftigen) Anforderungen so realisieren lassen, dass nicht andauernd neue Schnittstellen eingeführt werden müssen (bzw. wie dies möglichst oft bzw. möglichst weitgehend vermieden werden kann). Dazu setze ich methodisch an einer wesentlich früheren Stelle an als die meisten bisherigen Systementwürfe: bei der *Dekomposition* des *Problem-Raums*, und zwar nicht irgend eines speziellen Problem-Raums, sondern eines möglichst *universellen* und damit *offenen* Problem-Raums, der möglichst viele zukünftige Probleme zu umfassen sucht.

Diese Arbeit beschäftigt sich daher als erstes mit der Frage, wie man beliebige Problem-Räume möglichst effizient in Teilprobleme zerlegen kann, so dass möglichst viel *Redundanz* eingespart wird. Als Ergebnis dieser Überlegungen werden drei Arten von Generizität unterschieden, die jeweils unterschiedliches *Potential* zur Reduktion von Redundanz in einem Systementwurf besitzen: *universelle* Generizität (unbeschränktes Reduktions-Potential), *kompositorische* Generizität (exponentielles Reduktions-Potential), und *Erweiterungs*-Generizität (lineares Reduktions-Potential). Diese drei Arten lassen sich nicht nur *deduktiv* zur Analyse von Problem-Räumen einsetzen, sondern auch *konstruktiv* während der Entwurfs-Phase eines Systems, ja sogar bei der dynamischen *Rekonfiguration / Adaption* zur Laufzeit.

Diese gewonnenen Hilfsmittel werden nun beispielhaft

[1] Dies geschieht häufig auch *unabsichtlich*: praktisch alle heute gängigen Programmiersprachen erlauben die leichte Formulierung von Prozeduren, Funktionen oder Regeln (Prädikate) durch einfaches Hinschreiben, wobei jedesmal als Seiteneffekt eine neue Schnittstelle entsteht (z.B. in Form einer *Signatur* einer Prozedur oder Funktion). Selbst modularisierte Sprachen, in denen der Export von Schnittstellen ausschließlich durch explizite Angaben von Schnittstellen zu erfolgen hat, erlauben die Definition lokaler (Hilfs-)Prozeduren und -Funktionen. Die objektorientierte *Vererbung* erhebt die Erweiterung von Schnittstellen sogar zum *Grundprinzip*.

[2] Ein gutes Beispiel für die historische Fortentwicklung einer ursprünglich einfachen Betriebssystem-Architektur zu einem extrem konplexen und kaum noch überschaubaren Gebilde unter dem Druck ständiger Hardware- und Benutzer-Anforderungen stellt die Entwicklung von Unix dar, deren Grundideen auch heute noch im Linux-Kern zu finden sind.

auf den Entwurf einer Familie von Betriebssystemen angewandt. Dabei entsteht kein Entwurf eines bestimmten Systems, sondern eine Meta-Architektur im Stile eines LEGO-Baukastens, die mit nur zwei universellen Grund-Abstraktionen (*Nester* und *Bausteine*) auskommt. Die bekannten LEGO-Steine sind nicht nur zum Aufbau eines einzigen speziellen Systems vorgesehen, sondern erlauben den Aufbau sehr vieler unterschiedlicher Systeme, darunter auch neue und zukünftige Systeme, an die beim Entwurf der *Steine* noch gar nicht gedacht worden ist; so können etwa Gebäude, die erst in 100 Jahren gebaut werden und heute noch gar nicht bekannt sind, ebenfalls in 100 Jahren mittels LEGO-Bausteinen nachgebaut und modelliert werden (*Universalität* des Baukasten-Prinzips).

Für die wesentliche Grund-Funktionalität eines Desktop-Betriebssystems in der Art von Unix werden einige konkrete Baustein-Entwürfe vorgestellt, die als keineswegs abschließende konkrete Beispiele zu verstehen sind, was mit der Meta-Architektur eines Baukastens beispielsweise an Funktionalität realisiert werden kann. Es sind jedoch immer wieder Hinweise auf fortschrittliche Funktionalitäten eingestreut, die sich mit klassischen Betriebssystem-Architekturen nicht auf einfache Weise realisieren lassen, wo also der Baukasten-Ansatz deutliche Vorteile verspricht.

Der Rest dieser Arbeit untersucht Konzepte und Fragen, die sich aus dem LEGO-artigen Baukastenprinzip ergeben. Dazu gehören Gemeinsamkeiten und Unterschiede zu bekannten Architekturen und Entwurfsparadigmen wie Objektorientierung und Aspektorientierung. Die allgemeinen Prinzipien der Instanz-Orientierung als neues Grund-Paradigma zur Kontrolle eines LEGO-artigen Systemaufbaus werden separat dargestellt. Abschließend wird ein Ausblick auf zukünftige Speichermodelle gewagt, mit denen multiple transaktionale Sichten durchgängig im gesamten Betriebssystem realisiert werden können.

1.2. Beispiel eines LEGO-artigen Betriebssystem-Aufbaus

In diesem einführenden Abschnitt soll beispielhaft erklärt werden, wie Teile eines LEGO-artigen Betriebssystems aussehen können, das auf allen Ebenen und Schichten im wesentlichen nur die zwei Grund-Abstraktionen[3] *Nester* und *Bausteine* benutzt.

Präzisere Definitionen folgen später; zum Verständnis dieser Einführung reicht es, sich eine Nest-Instanz als einen (virtuellen) Adressraum vorzustellen, in dem sich Daten befinden können. Nicht alle Stellen dieses Adressraums müssen zugreifbar sein, es können beliebige Löcher darin vorhanden sein. Dies ähnelt einem Sparse File unter Unix, nur mit dem Unterschied, dass Zugriffe auf Löcher als Fehler behandelt werden. Eine weitere Metapher für Nester stellen Dateien des `/proc`-Filesystems von Unix dar, deren (virtuelle) „Inhalte“ ebenfalls dynamisch zur Laufzeit durch Aufruf von Operationen hergestellt werden.

[3] Was man als *Abstraktion* betrachtet, ist wiederum eine Frage der Abstraktion bzw. der Betrachtungs-Ebene. Fast alle Abstraktionen zerfallen bei näherer Betrachtung in Teil- und Unter-Abstraktionen. Der Terminus „Abstraktion“ wird hier in der (informellen) Bedeutung einer *wesentlichen* Haupt-Abstraktion verwendet, d.h. auf relativ hoher Betrachtungs-Ebene, in der auf für die jeweilige Betrachtung *wesentliche* Eigenschaften fokussiert wird.

In Erweiterung zu bisher verwendeten Adressraum-Abstraktionen gibt es auf Nestern eine Verschiebe-Operation `move`, mit der ein ganzer Speicherblock im virtuellen Adressraum auf virtuelle Weise verschoben werden kann. Der Speicherblock erscheint anschließend unter verschobenen virtuellen Adressen; die Implementierung erfolgt jedoch so, dass keine tatsächlichen Kopieroperationen im Speicher ablaufen, sondern nur der *virtuelle Eindruck* einer Verschiebung entsteht.

Bausteine kann man sich als „Transformatoren“ vorstellen, die ein oder mehrere verschiedene Eingangs-Nest-Instanzen in ein oder mehrere *andere* Ausgangs-Nest-Instanzen transformieren. Bausteine lassen sich *instantiieren*, d.h. man kann beliebig viele Inkarnationen des gleichen Baustein-Typs herstellen, deren Eingangs-Instanzen sich mit den Ausgangs-Instanzen anderer Baustein-Instanzen verbinden oder „verdrahten“ lassen. Als für Menschen leicht verständliche Darstellung benutze ich Zeichnungen, wie sie in der Elektro- und Digitaltechnik für Schaltbilder verwendet werden. Eine Baustein-Instanz wird als Kästchen mit linksseitigen Eingängen und rechtsseitigen Ausgängen gezeichnet. Die „Verdrahtungsregeln“ sind gleich wie bei Schaltbildern in der Elektrotechnik.

Als erstes Beispiel sehen wir uns in Abbildung 1.1 ein Szenario an, wie es bei konventionellen Betriebssystemen auftritt. Am Ende der Baustein-Hierarchie steht eine Instanz von `mmu_i386`, die einen (nicht eingezeichneten) virtuellen Benutzer-Adressraum 1:1 auf ihr Eingangs-Nest abbildet, indem sie die MMU-Hardware (Memory Management Unit) benutzt. Der Benutzer-Adressraum enthält genau das, was sich im Eingangs-Nest dieses Bausteins „befindet“ bzw. was dort vom Vorgänger-Baustein (virtuell) zur Verfügung gestellt wird. Im Beispiel ist dies ein virtuelles Prozessabbild, das von der `union`-Instanz generiert wird und als wesentliche Grundelemente jeweils ein Code-, ein Daten-, ein Stack-„Segment“ sowie eine virtuelle Abbildung einer `mmap`-Datei zu einem ausführbaren Prozessabbild zusammenstellt. Der Begriff „Segment“ wird jedoch in dieser Arbeit nicht verwendet, da er mit der Nest-Abstraktion zusammenfällt bzw. einen Spezialfall davon darstellt (üblicherweise enthält ein Segment lediglich keine Löcher).

Der `union`-Baustein wird aus mehreren Quellen gespeist. Drei davon sind Nest-Instanzen, die man konventionell als „Dateien“ oder „Files“ bezeichnen würde: eine Quelle ist eine `device_ramdisk`-Instanz, die flüchtigen Hauptspeicher bereit stellt (in der Praxis ist es sinnvoll, hierfür eine gepufferte „temporäre Datei“ zu verwenden, damit bei Speichermangel ein Auslagern auf Hintergrundspeicher möglich ist).

Die „Datei“-Nester stammen wiederum aus Quellen, die die Bezeichnung `dir_*` tragen. Die `dir_*`-Bausteine erfüllen ungefähr die gleichen Funktionen wie *Verzeichnisse* in konventionellen Dateisystemen. Im Unterschied zu konventionellen Dateisystemen implementieren sie keine Verzeichnis-*Bäume*, sondern nur flache Verzeichnisse. Durch Schachtelung von `dir_*`-Instanzen lassen sich sehr leicht konventionelle Verzeichnisbaum-Hierarchien nachbilden. Man kann sich eine `dir_*`-Instanz als eine Art „Container“ vorstellen, der den *Platz* für seine Ausgangs-Nester in seinem Eingangs-Nest allokiert und verwaltet. Dabei wird die Tatsache ausgenutzt, dass das Eingangs-

Nest die relativ billige Verschiebeoperation `move` bereit stellt, mit deren Hilfe das *Platzmanagement* leicht lösbar wird. Verschiebe-Operationen können beispielsweise benutzt werden, wenn neue Ausgangs-Nester hinzukommen oder wenn sich die Größe oder der Platzbedarf eines Ausgangs-Nestes ändert.

Die billige virtuelle Verschiebeoperation muss irgendwo herkommen und implementiert werden; dies erledigt die `map_simple`-Instanz, die im Beispiel an der „Wurzel“ der Verzeichnis-Hierarchie steht. Dort werden diejenigen Probleme gelöst, die bei konventionellen Dateisystemen als *Fragmentierungs-Probleme* bekannt sind (Lokalitätsverhalten des Zugriffs).

Die vorgeschaltete `buffer`-Instanz sorgt für die Entkoppelung des zeitlichen Zugriffsverhaltens auf langsame Peripheriegeräte wie z.B. Festplatten (`device_ide`), und lässt sich von der Funktionalität her mit konventionellen Buffer-Caches vergleichen.

Sämtliche Leitungen in der Zeichnung repräsentieren Nest-Instanzen, die jeweils immer die gleiche Schnittstelle besitzen. Die Bausteine sind daher in beinahe beliebiger Weise mit einander kombinierbar (die Frage ist nur, welche Kombinationen Sinn machen).

Das Beispiel in Abbildung 1.2 soll die Flexibilität dieses Systems andeutungsweise demonstrieren. Im Vergleich zu vorhin kommen zwei `remote`-Instanzen hinzu, die das Client-Server-Paradigma auf Nester übertragen. Eine `remote`-Instanz macht ein Nest so auf einem anderen Rechner verfügbar, als wäre es dort lokal vorhanden. Die eine `remote`-Instanz sitzt im Beispiel am Ende der Hierarchie und macht ein komplettes Prozessabbild auf einem anderen Rechner verfügbar. Dort befindet sich eine weitere `mmu_i386`-Instanz, in der parallele Kontrollflüsse ablaufen können, die sich physikalisch auf einem anderen Rechner befinden. Mit ähnlichen Konfigurationen lässt sich beispielsweise verteiltes Rechnen (number-crunching) oder Prozess-Migration betreiben.

Die andere `remote`-Instanz stellt das andere Extrem von möglichen Einsatz-Szenarien dar: die bisherige „Wurzel“ der Verzeichnis-Hierarchie wird auf einem anderen Rechner verfügbar gemacht; diese Funktionalität entspricht etwa derjenigen von konventionellen Netzwerk-Dateisystemen. Wie man an diesem Beispiel sieht, braucht hierfür kein neuer Baustein-Typ implementiert zu werden. Die anderen Bausteine müssen dazu eine später genauer untersuchte Kompetenz implementieren, nämlich die `multiuser`- oder `multiversion`-Kompetenz, mit der die Konsistenthaltung durch wechselseitigen Ausschluss durchgeführt wird. Nur wenn diese Kompetenz gegeben ist, dürfen Eingänge mehrerer Baustein-Instanzen *parallel* am gleichen Ausgang angeschlossen werden.

Das Beispiel 1.3 soll andeuten, wie sich die Funktionalität von Datenbanken auf Basis der Abstraktionen Nest und Baustein implementieren lässt. Die Grundidee besteht darin, (relationale) Datenbank-Tabellen in Nestern abzulegen. Der Baustein `join` führt die bekannte Join-Operation durch, indem er an seinem Ausgang eine virtuelle Produkt-Tabelle zur Verfügung stellt, die lesbar und ggf. auch in gewissen Grenzen modifizierbar sein kann. Der nachfolgende `transaction`-Baustein dient als Isolations-Puffer für zwei *transaktionale Sichten*, die voneinander unabhängige *isolierte* Sichten mit der bekannten ACID-Eigenschaft (siehe z.B. [HR83, LS87, GR93]) bereit stellt. Die beiden Sichten werden im Beispiel in Benutzer-Adressräume eingeblendet; damit haben wir einen *Synergieeffekt* zwischen dem Fachgebiet der Betriebssysteme und der Datenbanken ermöglicht.

Transaktionale Sichten sind nicht nur für die klassischen Einsatzgebiete in Datenbanken nützlich, sondern können je nach Einsatz an verschiedenen Stellen einer Baustein-Hierarchie beispielsweise zur Isolation kompletter Prozessabbilder bzw rekursiv geschachtelter virtueller Maschinen, oder im anderen Extrem von Festplatten-Partitions-Abbildern dienen (z.B. Semantik von Journaling Filesystems).

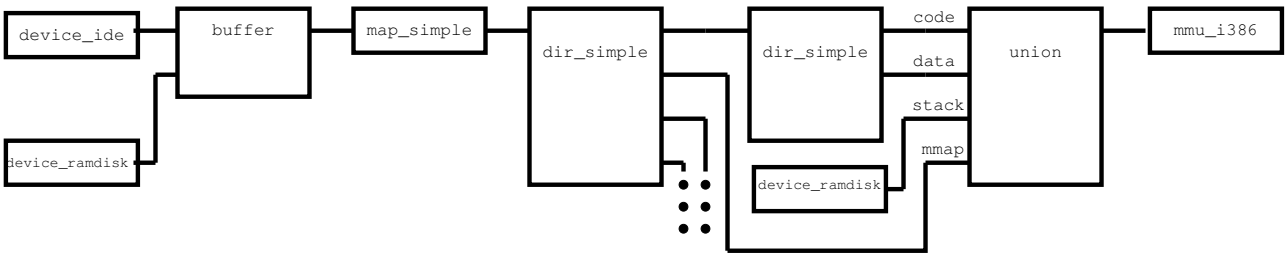


Abbildung 1.1.: Erstes Beispiel

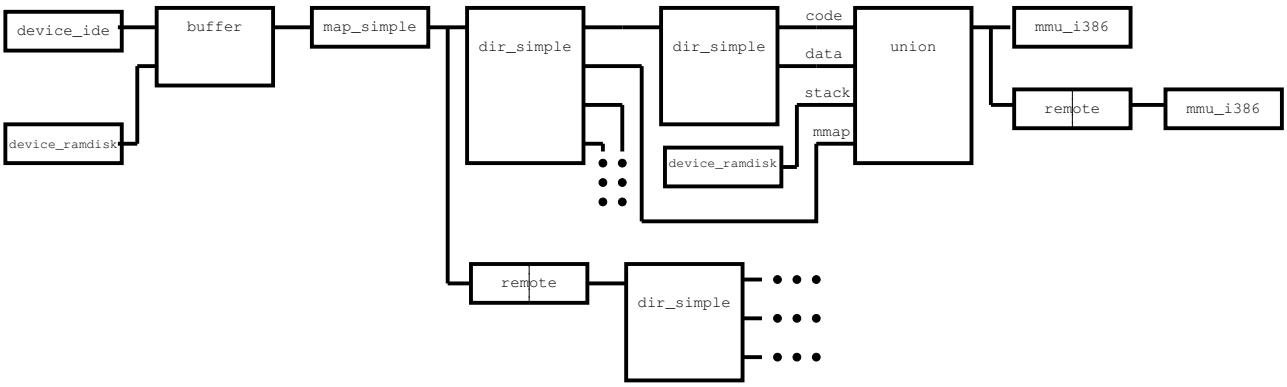


Abbildung 1.2.: Zweites Beispiel

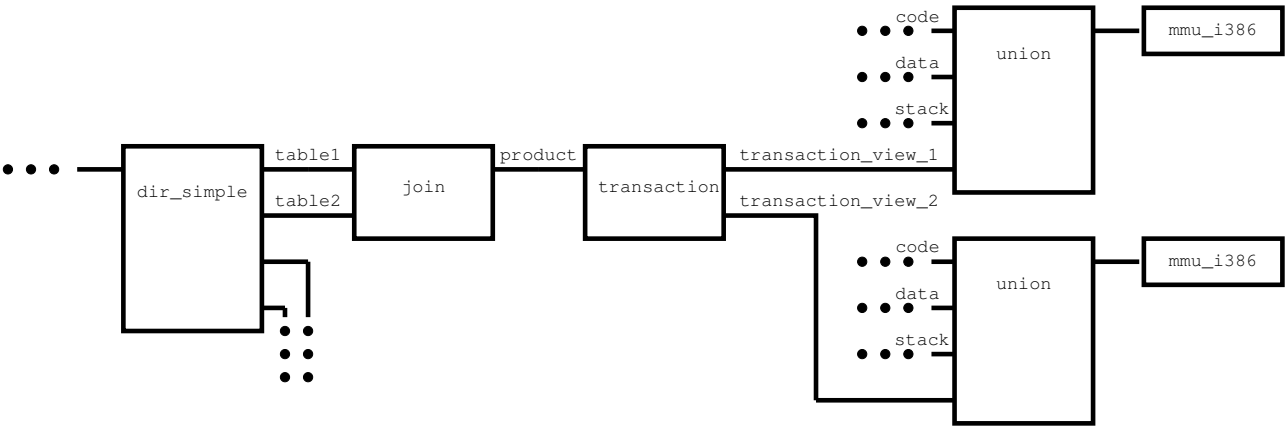


Abbildung 1.3.: Drittes Beispiel

2. Entwurfs-Prinzipien

In diesem Kapitel möchte ich die allgemeinen Prinzipien darstellen und begründen, auf denen die vorgestellte Betriebssystem-Architektur beruht (Herleitung des Baukasten-Ansatzes).

2.1. Generizität: Wahl der angemessenen Abstraktionsebene

Ausgangbasis aller Überlegungen ist die *Funktionalität*, die ein Betriebssystem implementieren muss, wenn es bestimmte *Anforderungen* erfüllen soll.

Die Anforderungen werden in dieser Arbeit als „freie Variable“ betrachtet[1]. Anforderungen können sich fortlaufend ändern; daher werden *Klassen von Anforderungen* auf informelle Weise betrachtet.

Für die zu implementierende Funktionalität gilt ein informelles Gesetz, das im Volksmund als “von nichts kommt nichts“ bekannt ist. Alles, was das System leisten soll (unabhängig davon, was das sein soll), muss irgendwo irgendwie von irgendwem implementiert werden.

Gängige Methoden zur Lösung dieser Problematik werden vor allem im Software-Engineering, speziell bei den objektorientierten Entwurfs-Methoden (z.B. [Mey88]) gelehrt (Stichwort Anpassbarkeit an neue Aufgabenstellungen). Der objektorientierte Ansatz versucht dabei die *Wiederverwendbarkeit* von Software-Modulen zu verbessern, indem er die *Erweiterung* bestehender Module und Schnittstellen zum Zwecke der Anpassung an neue Aufgabenstellungen ermöglicht. In der Praxis eingesetzte objektorientierte Software mit einer längeren Entwicklungsgeschichte zeigt nicht selten eine riesige Ansammlung von Klassen mit einer ziemlich komplizierten Klassen-Hierarchie und schwer durchschaubaren Vererbungs- und Enthaltenseins-Beziehungen („is-a“ und „has-a“-Beziehungen). Bei eingehender Betrachtung findet sich darin viel *Redundanz*, die (vielleicht) hätte vermieden werden können, wenn man die jetzigen Anforderungen von Anfang an gekannt hätte und bei sorgfältigem Systementwurf berücksichtigt hätte[2].

Die allgemeine Fragestellung lautet daher, wie man *unnötige Redundanz* in der Implementierung möglichst weit gehend vermeiden kann, selbst wenn (noch) gar nicht alle Anforderungen bekannt sind.

Ein Mittel zur Vermeidung unnötiger Redundanzen ist der Einsatz von *Generizität*.

Als Generizität betrachte ich jedes Mittel, das es erlaubt, ähnliche Dinge nur einmal hinzuschreiben, obwohl sie mehrmals vorkommen (evtl. in Varianten).

Diese Definition ist wesentlich weiter reichend als die oftmals in der Literatur anzutreffende Verwendung des Begriffs Generizität (siehe z. B. [Mey88, Kapitel 6 und 19]). Ich verwende diesen Begriff nicht in der Reinform, sondern betrachte immer nur Spielarten[3] von Generizität.

Eine ausführlichere Betrachtung von Generizität mit einer noch allgemeineren Definition findet sich in [ST03a].

2.1.1. Parametrische Generizität

Eine besondere Spielart ist die *parametrische Generizität*. Beispiele hierfür sind Ersetzungmechanismen wie z.B. der Lambda-Kalkül (siehe z.B. [Fie88]) und seine Anwendung in funktionalen Sprachen wie Lisp oder ML oder Haskell (vgl. [Tho96]), Parameter-Substitution in Makro-Präprozessoren, Parameter von Template-Mechanismen in objektorienten Sprachen wie C++, oder Anwendungen des Schlüsselwortes `generic` in Ada. Parametrizität bedeutet, dass im Unterschied zu allgemeineren Ersetzungsmechanismen wie z.B. Chomsky-Typ-0-Grammatiken oder L-Systeme keine beliebigen Ersetzungen möglich sind, sondern syntaktisch eindeutig gekennzeichnete Platzhalter, die *formale Parameter* genannt werden, durch Terme oder Ausdrücke mit einer fest gelegten Syntax (*aktuelle Parameter*) substituiert werden.

In der Literatur wird häufig der allein stehende Begriff der Generizität in ungefähr der gleichen Bedeutung wie hier die parametrische Generizität verwendet, oder er steht in noch weiterer Einschränkung synonym für parametrische Polymorphie von Typen (vgl. [CW85]), beispielsweise bei Meyer [Mey88, Kapitel 6 und 19][4]. Das Konzept der parametrischen Generizität setzt nicht das Vorhandensein von Typprüfungen voraus[5], jedoch sollten im Idealfall der-

[1] Der oft übliche Weg besteht in der Erfassung von Anforderungen vor dem Beginn jeglicher Entwurfs-Arbeiten. Wenn dies nicht möglich ist, könnte man der Versuchung erliegen, auf Anforderungen einfach nicht zu achten. Dies kann schwer wiegende Nachteile zur Folge haben. Es geht mir daher nicht darum, eine Anforderungs-Analyse abzuschaffen, sondern diese in *spätere* Entwurfs-Phasen, insbesondere den Detail-Entwurf konkreter Baustein-Typen, zu verschieben.

[2] Dieser Effekt ist nicht auf objektorientierte Systeme beschränkt; er scheint mir ein soziologisches und psychologisches Problem bei jeglicher Software-Entwicklung mit vielen Beteiligten zu sein, und dürfte daher auch für die hier vorgestellte Architektur ab einer gewissen Größenordnung relevant werden (Conway’s Law: „systems mirror the structure of organizations producing them“). Das objektorientierte Paradigma wird jedoch manchmal so gelehrt, als wäre die Erweiterung bestehender Schnittstellen ein Allheilmittel gegen die Problematik jeglicher Veränderungen von Anforderungen. Diese oft nur unterschwellig vorhandene Einstellung führt zusammen mit der durch die objektorientierten Sprachen stark erleichterten Handhabung der Erweiterungsmechanismen sehr leicht zur Ausuferung von Klassenhierarchien (bzw. bei konventionellen Entwürfen zum Anflanschen von “Balkonen“) und damit zur vermeidbaren Reduplikation von implementierter Funktionalität.

[3] Über die in dieser Arbeit untersuchten Spielarten von Generizität hinaus kann es auch noch weitere geben. Der allein stehende Begriff der Generizität sollte daher nicht mit einer speziellen Ausprägung verknüpft werden, sondern für spätere Erweiterungen offen gehalten werden.

[4] Insbesondere im Vergleich zu den Möglichkeiten der Textersetzung durch Makro-Prozessoren scheint Meyers Begriff von Generizität deutlich enger gefasst zu sein als der hiesige Begriff der parametrischen Generizität.

[5] Meyer meint in den Kapiteln 6 und 19, dass (parametrische) Generizität ohne Typprüfung sinnlos wäre. Dem widerspricht jedoch u.a. die Lisp-Tradition seit den 1960er Jahren, in der parametrische Generizität ohne *automatisierte* Typprüfungen benutzt wurde. Man muss le-

artige Mechanismen vorhanden sein. Moderne funktionale Sprachen wie Haskell erlauben Typsicherheit durch automatisierte Typ-Inferenz auch bei Typ-Parametern (parametrische Polymorphie von Typen, vgl. [CW85]). Makro-Präprozessoren führen dagegen meist nur eine reine Text-Substitution durch, die zwar universell für beliebige programmiersprachliche Konstrukte anwendbar ist, aber keine separate Prä-Compilierung und keine Typprüfung von Makros ermöglicht[6].

Parametrische Generizität erlaubt die Automatisierung immer wiederkehrender Programmierungs- und Formulierungs-Vorgänge. Formale Parameter können mehrmals auf verschiedene Weise durch verschiedene aktuelle Parameter substituiert werden. Dadurch ist eine kompakte Notation möglich, die Redundanz bei der Formulierung eines Software-Systems sparen hilft. Dieser Effekt ist nicht nur bei der Programmierung im Kleinen, sondern auch bei der Programmierung im Großen nutzbar.

Im Folgenden werde ich weitere Arten von Generizität auf informelle Weise vorstellen, die sich durch parametrische Generizität darstellen und simulieren lassen. Diese informell verwendeten Begriffe stellen kein eigenständiges theoretisches Konzept dar, sondern sind vielmehr als *Denk-Kategorien* oder besser *Denk-Hilfsmittel* für den Entwurf von komplexen Systemen durch Menschen gedacht (bei der Beschäftigung mit dem Problem-Raum).

Die Grundregel beim Entwurf lautet: verwende Generizität, wo immer es auf einfache Weise möglich ist und wo immer es Redundanz einsparen kann[7].

diglich die Typprüfungen notfalls „von Hand“ zur Laufzeit machen, wenn man das Konzept der parametrischen Polymorphie in solchen Sprachen wie der Urform von Lisp nutzen will. Vielleicht kommt dieses Missverständnis auch daher, dass der allein stehende Begriff der Generizität oftmals synonym für den vergleichweise eingeschränkten Begriff der parametrischen Polymorphie verwendet wird. Diese eingeschränkte Verwendung verstellt den Blick für die wesentlich weiter reichenden Möglichkeiten von Generizität im Sinne von Methoden und Mechanismen zur Reduktion von Redundanz.

[6] Einige Beispiele sollen kurz verdeutlichen, dass Makro-Prozessoren trotz ihrer geringeren Sicherheit gegen versehentliche Fehlbenutzung einige nützliche Anwendungen von parametrischer Generizität ausführen können, die mittels parametrischer Polymorphie nicht auf einfache Weise simulierbar sind: Beispielsweise kann man mittels Makro-Prozessoren die Felder von Strukturen (Records oder Klassen-Mitglieder) alphabetisch nach dem Namen oder nach kombinierten Schlüsseln wie (Typ,Name) sortieren. Man kann die Namen der Mitglieder einem bestimmten Namens-Schema unterziehen oder die Einhaltung eines bestimmten Namens-Schemas überprüfen, beispielsweise indem die Namen aller Mitglieder einer Klasse `A` die Form `A_*` haben müssen. In statisch typisierten Sprachen wie Eiffel oder C++ lassen sich statisch auswertbare Iteratoren über Klassen-Mitglieder definieren, die ansonsten nur in typlosen Sprachen wie Smalltalk auf einfache Weise generisch realisierbar sind. Mittels genügend mächtiger Makro-Sprachen wie beispielsweise TeX, die vom Typ Chomsky-0 sind, kann man theroretisch sogar automatische Typ-Inferenz zwischen Klassen betreiben und abstrakte Basisklassen aus einem gegebenen Klassenschema automatisch extrahieren. Derartige Techniken könnten beispielsweise zur Entdeckung bisher unentdeckt gebliebener Redundanz in großen Software-Projekten dienen.

[7] Übermäßiger Einsatz von Generizität kann die Redundanz auch erhöhen, die Durchschaubarkeit von Software verschlechtern, die Performanz verschlechtern, und damit zu negativen Effekten führen. Eine objektiv exakte Einschätzung und Abwägung der Einsatzstellen scheint schwierig und ist daher oftmals der subjektiven Einschätzung des jeweiligen Entwerfers unterworfen.

2.1.2. Erweiterungs-Generizität

Die in der Objektorientierung verwendete Erweiterung bzw. Spezialisierung von Schnittstellen durch Vererbung (gelegentlich auch die Einschränkung von Schnittstellen) lässt sich als eine besondere Art bzw. als Spezialfall von parametrischer Generizität charakterisieren: *Erweiterungs-Generizität*. Erweiterungs-Generizität lässt sich auch durch theoretische Begriffe wie Untertypen (subtyping, vgl. [CW85]) oder praxisnähere Ausprägungen wie Abstrakte Basisklassen beschreiben.

Im Anhang A wird durch ein Beispiel aufgezeigt, dass sich die Vererbung prinzipiell durch parametrische Generizität simulieren lässt[8]. Dieses Beispiel soll kein Vorbild guten Software-Engineerings darstellen, sondern lediglich zeigen, dass man *prinzipiell* die Vererbung sogar mit primitiven Mitteln wie dem C-Präprozessor nachbilden kann, obwohl der C-Präprozessor nicht einmal bedingte Ausdrücke *innerhalb* von Makro-Expansionen auswerten kann (was jedoch zur Nachbildung überschriebener Methoden sehr nützlich wäre). Der interessierte Leser, der das Beispiel aus Anhang A nachzuvollziehen versucht, wird auf den Kunstgriff verwiesen, dass man in einigen Makro-Sprachen die Namen von zu expandierenden Makros auch *berechnen* kann; im C-Präprozessor benötigt man dazu den Konkatenations-Operator ##. Dieser Kunstgriff wird u.a. auch gerne in Kreisen von TEX-Programmierern eingesetzt (vgl. \csname in [Knu94]).

Aus theoretischer Sicht handelt es sich hierbei um Funktionen höherer Ordnungen. Einige Verfechter objektorientierter Entwurfsmethoden wie z.B. [Mey88, Abschnitt 3.6.3] benutzen einen eingeschränkten Begriff von Generizität, dem sie dann nicht die Mächtigkeit zutrauen, den er bei entsprechender Fassung in der Theorie jedoch besitzt. Diese Mächtigkeit umfasst auch wesentliche Grundkonzepte der Objektorientierung. Daher betrachte ich Generizität schlechthin als Oberbegriff, unter den die Vererbung als Spezialfall untergeordnet wird.

2.1.3. Kompositorische Generizität

Eine weitere besondere Form von Generizität spielt in Betriebssystemen bei uniformen IO-Schnittstellen wie [Che87], in stapelbaren Dateisystem-Layern wie [HP94][9], in den Streams von System V Release 4 [GC94, Kapitel 7], oder beim Port-Konzept von Gnu Hurd [Hur] eine Rolle, die man als Vorläufer der hier vorgestellten Architektur

[8] Dieses Beispiel widerspricht nicht der scheinbar genau umgekehrten Aussage von Meyers Analyse in [Mey88, Abschnitt 19.3]. Meyer verwendet den Begriff der Generizität synonym zu einer Ausprägung von parametrischer Polymorphie, wie sie speziell in Ada auftritt. Im Gegensatz dazu verstehe ich unter parametrischer Generizität ein sehr viel allgemeineres Konzept, unter Generizität schlechthin sogar eine nochmals allgemeinere Denk-Kategorie. Man muss sich diese unterschiedlichen Begriffsbildungen vor Augen halten, wenn man die Aussagen miteinander vergleicht. Das hier vorgestellte Beispiel stützt jedoch die hier vorgeschlagene Begriffsbildung.

[9] Die in [HP94] vorgestellte hierarchische Modularisierung von Dateisystemen verwendet einige der hier vorgestellten Konzepte, beschränkt sich jedoch nicht nur auf die Funktionalität von Dateisystemen und verwendet deren aufwendigere Schnittstellen (insbesondere zum Management von Dateisystem-Unterbäumen), sondern basiert auch inhärent auf konventionellen Dateisystem-Konzepten und -Abstraktionen wie `vnodes`, `vnode`-Operationen und `vfs`, die in der vorliegenden Arbeit keine Rolle spielen.

ansehen kann (vgl. Pipe-and-Filters-Style in [SG96]). Konzeptuell wird bei den genannten Systemen die gleiche Art von Generizität wie bei der vorgestellten Architektur favorisiert, die ich *kompositorische Generizität* nennen möchte.

Kompositorische Generizität versucht, eine Funktionalität durch Verknüpfung von Baustein-Instanzen aus einem „Baukasten“ zu erzeugen (kartesische Produktbildung).

Auf den ersten Blick scheint kompositorische Generizität keine parametrische Generizität zu sein, da die syntaktische Darstellung oftmals anders als in funktionalen Sprachen oder Makro-Sprachen aussieht. Wie wir jedoch in Abschnitt 6.4 sehen werden, implementieren Bausteine Funktionen in einem weiteren Sinn, wobei die Verdrahtung von Bausteinen zu komplexeren Netzwerken sich als eine spezielle Art der Komposition von Funktionen auffassen lässt. Kompositorische Generizität ist ebenfalls eine Variante von parametrischer Generizität, die sich allerdings von der Erweiterungs-Generizität in der Entwurfs-Philosophie (nicht notwendigerweise jedoch in den verwendeten Mechanismen) stark unterscheidet: während objektorientierte Paradigmen ebenso wie die in den 1970er Jahren populären Hierarchie-Konzepte [Dij68, Dij71, Lis72, BS75, HFC76] ausdrücklich die Erweiterung von Schnittstellen bzw. von abstrakten Maschinen um neue Funktionalität durch Hinzunahme weiterer Operationen und Abstraktionen bezwecken und anstreben, wird bei kompositorischer Generizität das Gegenteil angestrebt: die Schnittstelle muss in den Grundzügen auf allen oder möglichst vielen Hierarchieebenen gleich oder zumindest weit gehend kompatibel bleiben, damit eine vielfältige Kombinierbarkeit möglich ist. Bei kompositorischer Generizität wird die Erweiterung der Funktionalität nicht durch Erweiterung der Schnittstellen, sondern durch Hinzunahme weiterer Funktionalität *innerhalb* der kombinierten Funktionen erzielt; neue Funktionalität ergibt sich mithin durch die *besondere Art der Verdrahtung zu einem Netzwerk*.

2.1.4. Universelle Generizität

Um kompositorische Generizität über das bisherige Anwendungsfeld von IO-Schnittstellen [Che87] hinaus auf das gesamte Betriebssystem ausdehnen zu können, müssen nicht nur die Urbild- und Bildbereiche der beteiligten Funktionen zueinander passen, sondern diese müssen darüber hinaus eine weitere Art von Generizität ermöglichen, die ich *universelle Generizität*[10] nennen möchte. Beispiele für universelle Konstrukte sind Turingmaschinen oder Registermaschinen, die prinzipiell andere Turing- oder Registermaschinen *simulieren* können. Universelle Generizität ist analog dazu die Fähigkeit, alternative Funktionalitäten, Repräsentationen, Zugriffsverhalten, Granularitäten etc. *auf relativ einfache Weise*[11] simulieren zu können; der Begriff ist daher kein absoluter Begriff, sondern gilt immer nur relativ zum Universum der jeweils simulierbaren alternativen Konzepte.

Ein Beispiel für universelle Generizität in Betriebssystemen ist die bekannte File-IO-Schnittstelle von Unix [RT74], die aufgrund ihrer Fähigkeit zu IO mit beliebiger dynamischer Blocklänge und beliebigen Zugriffsmustern auch ältere File-Konzepte wie z.B. Lochkartenstapel mit fester Satzlänge simulieren kann, darüber hinaus aber auch unterschiedliche Zugriffsmuster verschiedener Leser und Schreiber ermöglicht, die mit den vorher üblichen satzorientierten IO-Schnittstellen nicht auf einfache Weise realisierbar waren. Dieses inzwischen zur Selbstverständlichkeit gewordene historische Beispiel einer Vereinfachung durch universelle Generizität in einer Schnittstelle zeigt auf, dass „weniger oft mehr“ ist: der Verzicht auf das Konzept eines „Datensatzes“ (record) in Unix macht nicht nur die Schnittstellen und die Implementierung einfacher, sondern verbessert darüber hinaus die Funktionalität. Universelle Generizität ist daher in hohem Maße erstrebenswert und wird insbesondere beim Entwurf der Nest-Abstraktion (Kapitel 3) eine herausragende Rolle spielen.

2.1.5. Fragen der Anwendung von Generizität

Die zentralen Fragen bei der Anwendung von kompositorischer und universeller Generizität lauten:

- Wie sollen die Bild- und Urbildbereiche aussehen?
- Welche Funktionalitäten sollen durch die Bausteine realisiert werden?

Diesen Fragen liegt das gemeinsame Problem der *Wahl der am besten geeigneten Abstraktionsebene* für die zu lösenden Aufgaben (bei teilweise unbekannten Anforderungen) zu Grunde. Die Vorstellung und Propagierung eines bestimmten Mechanismus alleine sagt nicht viel über dessen konkrete Anwendung in einem bestimmten Gebiet aus. Entwurfs-Entscheidungen sind nicht immer eindeutig aufgrund gesicherter Methoden und Erkenntnisse lösbar, da bereits die Betrachtung der Problemstellung ein menschlicher Interpretationsprozess ist. In dieser Arbeit werde ich daher der *Pragmatik* einen vergleichsweise großen Raum einräumen und syntaktische und semantische Konstrukte nur zur Verdeutlichung und Notation der zugrundeliegenden Ideen verwenden.

Eine konkrete Wahl von Abstraktionsebenen könnte die berechtigte Kritik auf sich ziehen, dass präzise Aussagen über Vor- und Nachteile bestimmer Entwurfs-Entscheidungen nicht oder nicht ausreichend möglich sind. Im Forschungsgebiet der Betriebssysteme hat es bisher meines Wissens nach keine Evaluation verschiedener Entwurfs-Konzepte auf der Ebene ganzer Betriebssysteme gegeben, die diese Konzepte *als solche* auf einer wirklich *vergleichbaren* Grundlage (d.h. deren *isolierte* Auswirkung auf gut messbare Größen wie Performanz oder weniger gut messbare Größen wie die Änderungsfreundlichkeit) *ohne Einfluss weiterer versteckter Parameter* wie z.B. die Kenntnisse und Fähigkeiten der jeweiligen Implementierer un-

[10] Auch universelle Generizität läßt sich durch parametrische Generizität ausdrücken. Die Unterscheidung der verschiedenen Arten von Generizität ist daher letzlich nur ein menschlicher Verstehensansatz, der eine bestimmte *Denkweise* beim Entwurf *kommunizierbar* machen soll.

[11] Die Problematik bei einer formaleren Fassung des Begriffes der universellen Generizität besteht darin, was man unter *einfach* verstehen soll. Lässt man „komplizierte“ Simulationen zu, dann kann man wesentliche Teile der Funktionalität in der Abbildung auf das zu simulierende Konzept verstecken, anstatt es im benutzten Grundkonzept zu lösen. Die „Einfachheit“ einer Simulation lässt sich auf mehrere verschiedene Weisen formal fassen; da es mir bei dieser Diskussion nur um Denk-Kategorien während der Entwurfs-Phase geht, habe ich auf formale Fassungen all dieser Begriffe verzichtet.

tersucht hätten[12]; da solche Evaluationen nach aktuellem Stand kaum möglich erscheinen, ist es statt dessen üblich, das Für und Wider verschiedener Entwurfs-Alternativen durch informelle Argumentation darzustellen.

Der Begriff der „Architektur“ wird in dieser Arbeit in einem weiteren Sinn verwendet als vielerorts in der Literatur (vgl. z.B. [PC75, Jon80, MP81, B$^+$95, Lie95b, EKO95, Ass96, NS01])[13], da er weder feste Implementierungsparadigmen wie „communicating sequential processes“ (siehe [Hoa78]) vorschreibt[14], noch die Frage nach Kern-Größen und -Umfängen (vgl. [Lie95b]) in festliegender Weise beantwortet, noch die Frage nach den Grenzen zwischen Benutzer-Adressräumen und Kern- oder Server-Adressräumen festlegt, sondern diese Fragen je nach *Konfiguration* des Systems anders lösen kann, ohne den Quelltext ändern zu müssen. Im Vergleich zur konventionellen Handhabung des Architektur-Begriffes könnte man die hier vorgestellte Architektur daher auch als Meta-Architektur bezeichen; ich habe davon jedoch Abstand genommen, da der Begriff „Architektur“ laut Brockhaus schlicht „Baukunst“ bedeutet, und daher die in der Literatur teilweise vorkommende engere Auffassung nicht impliziert. In der Bedeutung als „Meta-Architektur“ werden einige konventionelle Architekturfragen, die bisher als „festverdrahtet“ betrachtet worden sind und deren Diskussion in der Literatur große Beachtung gefunden hat, zu *dynamischen Parametern*, deren Variation einen direkten Vergleich dieser Paradigmen bei einer Evaluation erleichtert[15].

Zur Begriffsbildung: statt mit Architektur oder Meta-Architektur könnte man die vorliegende Entwurfs-Methodik auch als *Framework*[16] oder als *Komponenten-Architektur*[17] bezeichnen. Da diese Begriffe nach weit verbreiteter Auffassung zur Zeit sehr eng mit der Objektorientierung verknüpft sind, die ich in der vorliegenden Arbeit zwar ermögliche, aber nicht zur zwingenden Voraussetzung der Architektur erhebe, habe ich von einer derartigen Begriffsbildung vorerst Abstand genommen. Im Unterschied zu Szyperskis Komponenten-Begriff [Szy98] könnte man die hier vorgeschlagenen Bausteine auch als *leichtgewichtige*[18] *instantiierbare*[19] *Komponenten* charakterisieren. Es gibt jedoch einen fundamentalen Unterschied zwischen Komponenten im Sinne Szyperskis und der Komposition von LEGO-artigen Bausteinen: bei klassischen Komponenten geht es um die Komposition von statischem Programm-Code, hier jedoch um die Komposition von Laufzeit-*Instanzen*, d.h. um *individuelle* Beziehungen zwischen einzelnen Objekt-Instanzen (vgl. [ST04b]). Während klassische Komponenten-Software a priori nur an vordefinierten Plug-In-Schnittstellen erweiterbar ist, erlaubt das hier vorgestellte LEGO-artige Baustein-Konzept die Erweiterung potentiell an allen nur denkbaren Stellen. Aus diesem Grunde favorisiere ich den Begriff „Baukasten“ (engl. „brick system kit“)[20].

[12] Ausnahmen hiervon dürfte es nur bei Untersuchungen über konkrete Alternativ-Mechanismen bei speziellen wohldefinierten Teilproblemen geben.

[13] In [Ant90] beschreibt Antonov eine „reguläre Architektur“, die trotz anderer formeller Darstellung dem Kerngedanken der hier vorgestellten Architektur sehr nahe kommt, dabei jedoch vorwiegend auf Erweiterungs-Generizität und weniger auf kompositorische Generizität ausgelegt ist. Im Detail gibt es weitere gravierende Unterschiede, so etwa bei der enormen Anzahl von Schnittstellen-Funktionen, bei den Instantiierungs-Mechanismen, den fest eingebauten Schutzmechanismen, und möglicherweise auch bei der Anzahl der „Ausgänge“ pro „Baustein“ (in allen Beispielen kommt immer nur ein einziger Ausgang vor; einen grammatisch klaren expliziten Hinweis auf mehrere „Ausgänge“ konnte ich nicht finden). Weiterhin fehlt bei Antonov der hier vorgestellte Ersatz von Dateisystemen durch dynamische hierarchische Instantiierung von Bausteinen; eine Unterscheidung verschiedener Betriebsarten oder Betriebsmodelle ist ebenfalls nicht zu erkennen.

[14] Eine Ausnahme ist [Str78], wo Stroustrup die Unabhängigkeit und Austauschbarkeit der Implementierungsparadigmen von den Modulschnittstellen aufzeigt, und damit den Architekturbegriff ebenfalls weiter auffasst. Leider scheint dieser grundlegende Artikel wenig Beachtung in der Betriebssystem-Literatur gefunden zu haben, da auch Jahrzehnte später die Verwendung fester Implementierungsparadigmen immer noch zum Stand der Technik zu gehören scheint.

[15] Dies gilt nur für Entwurfs-Entscheidungen, die *innerhalb* der hier vorgestellten Architektur offengelassen und daher parametrisierbar sind; auf der Ebene der Meta-Architektur selbst werden jedoch auch von mir feste Entscheidungen gefällt, die prinzipiell (wie auch bei bisherigen Architekturen) der oben genannten Kritik unterliegen und im Einzelfall weiterer Evaluation und ggf. einer Revision bedürfen.

[16] Dieser Begriff wird in der Praxis meist mit der Objektorientierung verknüpft (vgl. z.B. [JH02, S. 273]: „Frameworks stellen komplexe, partielle Programme dar, deren Instantiierung zu einer Applikation ohne Vererbung nicht denkbar wäre“), bis hin zur Bezeichnung einer *konkreten Implementierung* einer Sammlung von abstrakten Basisklassen, und könnte daher zu Missverständnissen führen. Eigentlich würde der Begriff Framework gut auf die hier vorgestellte Architektur passen, wenn er nicht bereits auf die zitierte Art eingeschränkt wäre. Eventuell wäre es sinnvoll, über eine Erweiterung dieses Begriffs nachzudenken; darüber sollte jedoch ein weitgehender Konsens erzielt werden.

[17] Zum aktuellen Diskurs zwischen Broy / Sindersleben [BS02] und Jähnichen / Herrman [JH02] über Wesen, Vor- und Nachteile der Objektorientierung, insbesondere zum Zusammenhang zwischen Komponenten und Objektorientierung, kann die vorliegende Arbeit eine Erweiterung der Perspektive beitragen. Man könnte die hier vorgestellte Architektur ohne weiteres auch als eine Komponenten-Architektur bezeichnen (jedoch nicht ausschliesslich, da es im Unterschied zu konventionellen Komponenten nicht um Komposition von ausführbarem Code, sondern von Laufzeit-*Instanzen* geht), obwohl sie nicht *unbedingt* auf der Objektorientierung aufbaut (vgl. Abschnitt 6.2.2) und damit von der vorherrschenden Meinung in dieser Hinsicht abweicht. Wie in Abschnitt 6.2 gezeigt wird, wird objektorientierte Vererbung von der hier vorgestellten Architektur *ermöglicht*, aber nicht zur *zwingenden Voraussetzung* erhoben. Ich habe in den Beispielen des Kapitels 4 besonderen Wert darauf gelegt, die Grundfunktionalität eines Betriebssystems möglichst nur mit Hilfe kompositorischer Generizität zu realisieren und Erweiterungs-Generizität weit gehend zu vermeiden. Damit existiert ein Beispiel, das eine Komponenten-Architektur ohne *feste* Verknüpfung mit einer wesentlichen Grundidee der Objektorientierung (egal ob man sie mit „Vererbung“ oder als Spezialfall einer „Delegation“ bezeichnet) realisiert. Damit dürfte genügend Substanz zur Erhärtung der These vorgebracht worden sein, dass Komponenten-Architekturen als weitgehend orthogonal zur Anwendung spezifisch objektorientierter Entwurfs-Methoden betrachtet werden *können*. Eine solche Betrachtung hängt allerdings von der Definition von „Objektorientierung“ ab; je nach Geschmack oder Herkunft aus einer bestimmten Schule kann man die jeweiligen Begriffe so definieren, dass Objektorientierung einen Oberbegriff darstellt, unter den auch Komponenten-Achitekturen als ergänzende Anwendung des Prinzips der kompositorischen Generizität fallen, oder genau umgekehrt, oder dass zwei zueinander orthogonale Begriffe entstehen. Ich plädiere aus Gründen der methodischen Sparsamkeit für die dritte Variante.

[18] Vgl. [Szy98, Abschnitt 4.1.8]. Das vermutete Phänomen „maximizing reuse minimizes use“ scheint mir auf der impliziten Annahme von Erweiterungs-Generität als Maximierungs-Strategie über Komponenten-Grenzen hinweg zu beruhen; dieses Phänomen wird durch das Konzept der kompositorischen Generizität und der universellen Generizität in sein Gegenteil verkehrt.

[19] Szyperski [Szy98] betont mehrmals (Abschnitte 1.5, 4.1.1), dass die Instantiierung von Komponenten wegen ihrer Statuslosigkeit sinnlos wäre. Im Unterschied dazu haben beispielsweise mehrfache Instanzen von `dir_*`-Bausteinen durchaus Sinn, da es durchaus auf die *Stellung* in einer Baustein-Hierachie bzw. auf den *Verdrahtungs-Kontext* ankommt.

[20] Eventuell käme auch noch der Begriff „Meta-Middleware“ in Betracht, wenn man auf die instanzorientierte Möglichkeit zur Zusammenschaltung von Systemen (vgl. Kapitel 7) fokussiert.

Bei Konflikten zwischen verschiedenen Arten von Generizität bei zu treffenden Entwurfs-Entscheidungen propagiere ich zumindest für die Anwendung in Betriebssystemen die folgende Priorisierung miteinander konkurrierender Arten von Generizität:

1. Universelle Generizität
2. Kompositorische Generizität
3. Erweiterungs-Generizität

Wie in [ST03a] genauer[21] dargestellt wurde, besitzt die universelle Generizität das höchste Potential zur Reduktion von Redundanz, während die Erweiterungs-Generizität das geringste Potential besitzt. Alleine aufgrund dieser Überlegung ist daher die Frage berechtigt, ob die fast ausschließlich auf der Erweiterungs-Generizität beruhende Objektorientierung das theoretisch vorhandene Potential genügend ausschöpfen *kann*. Diese Arbeit stellt einen Versuch dar, das bisher möglicherweise brachliegende Potential wenigstens teilweise verfügbar zu machen.

2.2. Trennung von Mechanismus / Strategie / Repräsentation

Im Betriebssystem-Bau ist das Prinzip der Trennung von *Mechanismus* (mechanism) und *Strategie* (policy) schon sehr lange bekannt und benutzt worden; eine Darstellung dieses Prinzips und ihrer Vorteile findet man z.B. in [LCC+75, CJ75].

Als unabhängig von Mechanismen und Strategien ist fernerhin die *Repräsentation* zu betrachten[22]. Beispielsweise lässt sich ein Monitor [Han73, Hoa74] als programmiersprachliche Repräsentation eines Mechanismus verstehen, der prinzipiell wie ein binäres Semaphor funktioniert; die programmiersprachliche Repräsentation eines Monitors lässt sich bekanntermaßen auf einfache Weise mittels Semaphoren und konventionellen Sprachmitteln simulieren[23]. Hieraus ist zu erkennen, dass die Auswahl einer einzigen Repräsentation ausreichend ist und Redundanz vermeiden hilft[24]; diese Repräsentation sollte jedoch möglichst universell und einfach handhabbar / verstehbar sein. Die Entscheidung für die Auswahl einer bestimmten syntaktischen Repräsentation ist daher der Pragmatik zuzurechnen; im Rahmen dieser Arbeit wird nicht weiter darauf eingegangen; insbesondere werden Repräsentationsfragen nicht zur Auswahl von Mechanismen oder Strategien herangezogen.

Die Abgrenzung zwischen Mechanismus und Strategie ist nicht immer eindeutig möglich, da sie vom Kontext und der gewählten Abstraktionsebene abhängt. Eine Strategie kann beispielsweise ihrerseits wieder in Unter-Strategien und Unter-Mechanismen zerfallen. Auch Mechanismen können in ihrer Feinstruktur wiederum Strategien enthalten.

Auf relativ grober Abstraktionsebene korrespondieren Mechanismen und Strategien mit den hier propagierten Abstraktionen Nest und Baustein auf ziemlich direkte Weise: ein Nest stellt einen Satz von abstrakten Mechanismen zur Verfügung, ein Baustein implementiert eine Strategie oder eine Menge von Strategien.

Die Nest-Schnittstelle stellt insofern *abstrakte* Mechanismen bereit, als dass die konkrete Implementierung der Mechanismen durch späte Bindung (late binding) zu Stande kommen kann.

Die Trennung zwischen Mechanismus und Strategie wird durch die hier vorgestellte Architektur nicht nur direkt unterstützt, sondern geradezu zum Prinzip erhoben. Da Bausteine wiederum andere Bausteine enthalten oder über ihre Eingänge benutzen können, lässt sich das Prinzip der *schrittweisen Verfeinerung* auf die Trennung in Mechanismen und Strategien anwenden: eine Baustein-Hierarchie fungiert gleichzeitig als hierarchisch-rekursive Zerlegung des zu lösenden Problems in Mechanismen und Strategien.

2.3. Das Verantwortungs-Prinzip

Verantwortung und ihre Aufteilung ist ein allgemeines Prinzip, das in komplexeren menschlichen Gesellschaften und Systemen wie Firmen oder Behörden seit Jahrtausenden mit Erfolg praktiziert wird; als Ergebnis eines Aufteilungsvorgangs von Verantwortung entstehen *Zuständigkeiten*. Im Kontext von Betriebssystemen wurde Verantwortung ebenfalls als Strukturierungs-Leitlinie eingesetzt (ohne dass dies den jeweiligen Autoren völlig bewusst gewesen sein muss[25]): die in den 1970er Jahren populären hierarchischen Strukturen [Dij68, Dij71, Lis72, Lag75] lassen sich beispielsweise durch Aufteilung von Verantwortung gewinnen, ebenso die Weiterführung dieser Idee in so genannten objektorientierten Betriebssystemen (beispielsweise [K+81, Y+90, B+95, Ass96] u.v.m.), aber auch die auf bestimmten festen Mechanismen aufgebauten Betriebssysteme (z.B. auf Capabilities basierend [NW77, T+90]). Auch das im Software-Engineering oft zitierte Lokalitäts-Prinzip als Mittel zur Strukturierung von Systemen beruht

[21] Eine genaue Evaluation der Vor- und Nachteile dieser Priorisierung für die Anwendung in Betriebssystemen steht noch aus.

[22] Eine ähnliche Auffassung vertritt Parnas in [Par78].

[23] Hiervon unabhängig sind wiederum die Warte-Strategien mehrerer eintrittswilliger Prozesse wie beispielsweise die FIFO-Strategie (als interessantes Realisierungsbeispiel hierzu siehe [RK79]).

[24] In der Praxis ist dieser Vorsatz nicht immer konsistent durchzuhalten, vor allem in größeren Projekten mit vielen Beteiligten, die unterschiedliche Interessen verfolgen. So könnte es beispielsweise vorkommen, dass eine Interessengruppe die Programmiersprache Ada bevorzugt, die andere die Sprache C (Argumente für die jeweiligen Positionen sind sattsam bekannt: Ohne Zweifel ist Ada die sicherere, produktivere und wartungsärmere Sprache, der Großteil bereits existierender und ausgetesteter Software ist jedoch in C oder C++ geschrieben, zu der unbedingt Kompatibilität zu wahren ist). Prinzipiell lässt sich die hier vorgestellte Architektur nicht nur einheitlich in einer der beiden Sprachen realisieren, sondern auch bausteinweise in beliebigen Mischungen, sofern die Aufruf-Konventionen der jeweiligen Sprachen und Compiler zueinander kompatibel sind. Da dies nicht nur die Redundanz, sondern auch den Integrationsaufwand und mögliche Fehlerquellen erhöhen kann, sollte dies möglichst vermieden, auf jeden Fall aber sehr gut überlegt und begründet werden.

[25] Wie [Mey88] und viele andere Autoren bemerken, verlaufen die Haupt-Auseinandersetzungen um Strukturierungs-Kriterien hauptsächlich entlang der Demarkations-Linien „Strukturierung nach Daten“ versus „Strukturierung nach Kontrollfluss“ (vgl. [Par72]). Die von mir propagierte Strukturierung nach *Verantwortungsbereichen* nimmt demgegenüber eine übergeordnete Perspektive ein: Verantwortung ist fast immer an Datenstrukturierung und nur selten an Kontrollflüsse gekoppelt. Die Abstraktionsebene ist jedoch von konkreten Datenstrukturen abgekoppelt: bei hoher Abstraktionsebene kann sich Verantwortung auf *Mengen* von Datenstrukturen erstecken, im anderen Extrem aber auch bis auf Teil-Datenstrukturen verfeinert werden.

letztlich auf der Aufteilung von Verantwortung. Es gibt unzählige weitere Beispiele.

2.3.1. Kontrollflüsse

Ein *sequentieller Kontrollfluss* [Lag78] (thread, konzeptuelle Beschreibung bereits in [DH66]) lässt sich als Aktivitätsträger charakterisieren, der von außen beobachtbar nur eine sequentielle Folge von (Maschinen-)Operationen ausführt. Kontrollflüsse sind bereits in [Dij68] zur Strukturierung von Verantwortung in Betriebssystemen benutzt worden; diese feste Verbindung von Kontrollflüssen mit der Verantwortung für bestimmte Bereiche wird in der vorliegenden Arbeit jedoch aufgegeben. Kontrollflüsse *können* zur Strukturierung von Verantwortung benutzt werden, jedoch gibt es in der hier vorgestellten Architektur keine *notwendigerweise feste Verbindung* mit Verantwortungsbereichen (wobei im Einzelfall auch feste Verbindungen hergestellt werden können, aber nicht müssen).

Kontrollflüsse werden als vollkommen unabhängig von konventionellen Konzepten wie *Prozesse* oder Mechanismen wie *Schutzbereiche* angesehen. Im Gegensatz zu anderen Arbeiten werde ich den Begriff des Prozesses vermeiden, da es für ihn mehrere verschiedene Definitionen gibt[26]. Seit der Wiederentdeckung der Schutzbereiche (spheres of protection [DH66], protection domains [NW74, CJ75]) im Kontext so genannter Single-Address-Space-Systeme [C$^+$94, Ros94] (vgl. Beschreibung und Schutzmechanismen von CTSS in [BPS81]) hat sich die Bedeutung der Begriffe weiter verkompliziert. Ich beschränke mich daher auf die Begriffe Kontrollfluss und Schutzbereich.

Jeder Kontrollfluss hat zu jedem Zeitpunkt einen eindeutig bestimmten *Kontext*. Die Bedeutung dieses Kontexts hängt vom jeweils gewählten *Ausführungs-Modell* ab: bei der Simulation eines „Prozesses“ wäre dies beispielsweise das Prozessabbild, in dem sich der Kontrollfluss befindet und seine Operationen ausführt; im Fall von Single-Space- oder Hybridsystemen wäre dies der jeweils aktuelle Schutzbereich.

Kontexte können auf Anforderung *gewechselt* werden (vgl. [DH66]): ein Kontrollfluss wechselt damit in einen anderen „Prozess“ bzw. in einen anderen Schutzbereich, der sich ggf. auch auf einem anderen Rechner befinden kann. Hiervon sind zwei verschiedene Varianten denkbar: eine ohne Abspeichern des alten Kontextes im Stile von Wechseln zwischen Coroutinen, die andere mit Abspeichern aller durchlaufenen Kontexte in Form eines Stapelspeichers mit der Möglichkeit zur Rückkehr in einen früheren Kontext ähnlich einem synchronen RPC oder LRPC. Eine Konsequenz aus der Wechselmöglichkeit ist, dass die Anzahl der in einem „Prozess“ oder Schutzbereich tätigen Kontrollflüsse sich dynamisch ändern und auch zeitweise zu Null werden kann.

Man sollte sich vor Augen halten, dass Schutzbereiche einen Mechanismus zur Durchführung und Überwachung von Sicherheits-Konzepten darstellen, wohingegen Verantwortungsbereiche ein davon prinzipiell unabhängiges *logisches Konzept* zur Strukturierung eines Systems darstellen. Die Abbildung von Verantwortungsbereichen auf Schutzbereiche ist eine Frage von Strategien und kann auf unterschiedliche Weise, ggf. auch dynamisch zur Laufzeit, gelöst werden.

Kontrollflüsse werden in konventionellen Architekturen als eigenständige Abstraktion betrachtet. Da ihre Implementierung in einem eigenen Baustein erfolgen kann (vgl. `thread_*` bzw. `cpu_*` in Abschnitt 4.3.1), ist die Behandlung als separate Abstraktion nicht unbedingt notwendig. Kontrollflüsse können prinzipiell in jeder Nest-Instanz entstehen, indem eine intern asychron arbeitende Implementierung der Operation aufgerufen wird. Über den Weg einer Implementierung als spezielle Strategie in Bausteinen können Kontrollflüsse praktisch die gesamte Infrastruktur durchdringen, ohne sie notwendigerweise als eigenständige Abstraktion behandeln zu müssen (Spezialfall der Nest- und Baustein-Abstraktion bzw. Anwendungsfall von universeller Generizität).

Im Folgenden betrachten wir unser Betriebssystem zur Vereinfachung standardmäßig (sofern nicht ausdrücklich abgewichen wird) weder mit Hilfe von Prozess- oder Schutzbereichen, noch von Kontrollflüssen oder anderen Aktivitätsträgern, sondern ausschließlich auf Basis der jeweils zu implementierenden und zu verantwortenden Funktionalität.

2.3.2. Schnittstellen-Mechanismen

Die Gesamtverantwortung eines Betriebssystems kann auf sehr viele verschiedene Arten aufgeteilt werden (vgl. [Par72]); die von mir bevorzugten Aufteilungs-Strategien werden in Abschnitt 2.4 näher dargestellt. Jegliche Art von Aufteilung von Verantwortung führt dazu, dass *Modularisierungs-Grenzen* in ein System eingeführt werden, die bei formalisierter Darstellung auch als *Schnittstellen-Instanzen* bezeichnet werden.

Durch die Einführung von Schnittstellen-Instanzen zur Begrenzung von Modul-Instanzen entsteht eine Unterscheidung der durchzuführenden *Operationen* in *Aufrufer*- und *Bearbeiter*-Instanzen, die jeweils immer nur relativ zu einer festen Schnittstelle so bezeichnet werden; dieselbe Instanz kann bezüglich verschiedener Schnittstellen gleichzeitig als Aufrufer oder Bearbeiter fungieren. Bei verschiedenen unterscheidbaren Operationen können diese Rollen unterschiedlich verteilt sein, beispielsweise kann eine Modul-Instanz A gegenüber der Instanz B bei der Durchführung der Operation op_1 als Aufrufer, bei op_2 als Bearbeiter fungieren.

Als Mechanismen zur Weitergabe der Verantwortung einer Aufrufer- an eine Bearbeiter-Instanz kommen mehrere bekannte und bewährte Methoden in Betracht. Beispielsweise kommen hierfür in Frage, in der Reihenfolge fallenden Overheads bzw. fallender Kosten:

- RPC (remote procedure call) [BN84, TA90]
- LRPC (local / lightweight RPC) [B$^+$90]
- Indirekte Prozeduraufrufe
- Direkte Prozeduraufrufe
- Makro- oder Inline-Prozeduraufrufe

[26] Heute wird unter Prozess meist ein in Ausführung befindliches Programm verstanden, das eine Menge von Ressourcen allokiert hat und mindestens einen Kontrollfluss (thread) besitzt. In den 1960er und 1970er Jahren waren teilweise noch einfachere Definitionen üblich, die mit dem heutigen Begriff des Kontrollflusses (thread) zusammen fallen.

Diese Beispiel-Mechanismen eignen sich in ihrer ursprünglichen Form teilweise nur für die *synchrone* Weitergabe von Verantwortung, bei der die Aufrufer-Instanz bis zur Beendigung der Bearbeitung *warten* muss. Bei *asynchroner* Weitergabe der Verantwortung entsteht ein weiterer *logischer Kontrollfluss*[27]. Die asynchrone Weitergabe von Verantwortung per Prozeduraufruf ist jedoch mittels der in Abschnitt 4.3.1 beschrieben Methodik so lösbar, dass die Benutzung synchroner oder asynchroner Semantik unter Beibehaltung der gleichen uniformen Schnittstelle nicht nur möglich ist, sondern die Entscheidung darüber im Extremfall sogar individuell zur Laufzeit getroffen werden kann.

Man sollte sich vor Augen halten, dass die Frage nach einer parallel oder sequentiell übertragenen Verantwortung prinzipiell von der Aufteilungsstrategie in Module unabhängig ist und daher als eine Frage der konkreten Implementierungsstrategie gesehen werden kann.

Universelle Generizität relativ zu den oben genannten synchronen Mechanismen lässt sich direkt durch einheitliche Verwendung von synchronem RPC auf allen Ebenen des Systems erzielen, da dieser die Semantik der anderen Mechanismen als Spezialfall mit umfasst. Dies würde jedoch die Performanz eines lokalen Rechners in unakzeptabler Weise verschlechtern.

Zur Erzielung universeller Generizität unabhängig von verwendeten Kommunikationsmechanisem sehe ich folgende zwei Vorgehensweisen, die sich teilweise überschneiden:

1. Man wähle eine einheitliche und bequeme syntaktische Repräsentation, beispielsweise die Prozeduraufruf-Syntax der verwendeten Programmiersprache. Die *Bindung* von Instanzen dieser Konstrukte an einen der obigen Mechanismen erfolgt dann entweder zur Übersetzungszeit des jeweiligen Moduls, oder falls möglich auch zur Laufzeit bei der Instantiierung eines Moduls (die Unterscheidung zwischen indirekten / direkten / Inline-Aufrufen ist mit heute gängiger Standard-Technik[28] nicht ohne weiteres zur Laufzeit möglich). Dies ist im Wesentlichen eine Erweiterung der bereits in [Str78] dargestellten Methode.
2. Man verwendet indirekte / direkte / inline-Prozeduraufrufe als relativ billige Standard-Basismechanismen und implementiert „teure“ Mechanismen wie RPC in speziellen Bausteinen. Dies hat den Vorteil der jederzeitigen Erweiterbarkeit um weitere Kommunikationsmechanismen. Die Existenz der Kommunikations-Bausteine kann durch instanzorientierte Sichten ([ST04b] bzw. Kapitel 7) versteckt werden, so dass der Eindruck entsteht, man hätte einen kommunizierenden Draht vor sich.

Im Athomux-Prototyp wurde die Alternative 2 bevorzugt.

2.3.3. Abgrenzung von Verantwortung durch Kapselung

Im Software-Engineering ist lange bekannt, dass die Prüfbarkeit, Wartbarkeit und viele andere Eigenschaften von Modulen verbessert werden, wenn die Schnittstellen möglichst „dünn“ sind und möglichst wenig Information über die innere Struktur der Implementierung preisgeben (Prinzip der *Verbergung von Information*, vgl. [Par72]).

Dies bedeutet zum einen, dass Schnittstellen stets offengelegt werden müssen, wobei die Schnittstelle zwischen Bausteinen eine von der Architektur vorgebene Form haben muss, an die sich die Implementierer von Bausteinen halten müssen. Alles andere ist als *lokale Variablen* eines Bausteins zu betrachten. Die interne Realisierung von Bausteinen darf jedoch bzw soll möglichst andere Baustein-Instanzen als lokale Variablen benutzen (wobei wiederum keine „schwarzen Schnittstellen“ eingeführt werden dürfen), so dass sich insgesamt eine baumartige *Lokalitäts-Hierarchie* ergibt, die von der hierarchischen Struktur der äußerlichen Baustein-Verdrahtungen unabhängig ist. Die Verantwortung für den Einsatz und den Betrieb von lokalen Baustein-Instanzen liegt beim Implementierer eines Bausteins. Die Tatsache der Benutzung von lokalen Baustein-Instanzen ist dabei von außen nicht sichtbar.

2.4. Zerlegungs- und Rekombinationsstrategien

2.4.1. Zerlegung der zu lösenden Aufgaben in möglichst wenige universelle Elementarteile

Bei der Problemanalyse propagiere ich die Anwendung einer einfachen Methode: es ist zu fragen, wie generische Funktionalität (vorzugsweise in Form von universeller Generizität) in möglichst wenige, aber universelle Elementarfunktionalität zerspalten werden kann. Aus dieser Elementarfunktionalität ergeben sich dann die *Elementaroperationen* als Umsetzung in ein von-Neumann-Maschinenmodell, die im Pseudo-Code einer imperativen Programmiersprache notiert werden.

[27] Es muß nicht unbedingt ein tatsächlich neuer Kontrollfluss im physischen Sinne entstehen: Bei der Implementierung von Bausteinen als Server-Prozesse (vgl. [Han70, Hoa78]) bleibt die Anzahl physischer Kontrollflüsse im Regelfall konstant. Die im Nachrichtensystem gepufferte Nachricht stellt im logischen Sinne einen logischen Kontrollfluss dar, der von einem physischen Kontrollfluss simuliert wird, sobald die Nachricht bearbeitet wird. Solche Systeme sind für den Fall gebaut, dass mehr logische als physische Kontrollflüsse auftreten können, und beschränken daher die theoretisch mögliche maximale Parallelität auf künstliche Weise. Falls jedoch genügend physische Kontrollflüsse vorhanden wären, um alle eingehenden Nachrichten eines Bausteins sofort bearbeiten zu können, wäre die Pufferung von Nachrichten überflüssig. In der Praxis ist die Bevorratung von Kontrollflüssen mit hohem Overhead verbunden; wenn man deswegen die Bevorratung von Kontrollflüssen durch dynamische Erzeugung und Destruktion von Kontrollflüssen (vgl. historisches Beispiel [Opl65]) ersetzt, landet man wieder beim allgemeinen asynchronen Modell, dessen Parallelitätsgrad theoretisch nicht beschränkt ist.

[28] Mittels dynamischer Code-Generierung, wie sie z.B. bei Java bei der Umsetzung von Byte-Code in Zielplattform-Maschinencode eingesetzt wird, kann man prinzipiell auch die Bindung von Prozeduraufrufen zu allen genannten Mechanismen zur Laufzeit durchführen. Es wäre ein interessantes Forschungsprojekt, dies nicht nur aus dem Quelltext oder aus Zwischencode-Darstellungen heraus, sondern auch aus bereits vollständig compiliertem Code heraus (der ggf. Zusatzinformationen enthalten müsste) zu bewerkstelligen.

2.4.2. Orthogonalität und Rekombinierbarkeit von Elementaroperationen

Im Idealfall sollten die Elementaroperationen die Eigenschaft der *Orthogonalität* besitzen, d.h. keine der Elementaroperationen sollte sich durch Kombination anderer Elementaroperationen auf einfache Weise simulieren lassen.

Eine Zerspaltung in orthogonale Elementaroperationen kann dazu führen, dass sehr kleine, nach üblichen Maßstäben „triviale“ Elementaroperationen entstehen, die in der Praxis selten oder nie als allein stehende Operationen genutzt werden, sondern häufig nur in Kombination mit anderen Elementaroperationen. Entwerfer haben oft die konkrete programmiersprachliche Repräsentation einer Elementaroperation als Prozeduraufruf oder RPC vor Augen und schrecken daher oft vor dem Overhead einer solchen „trivialen“ Zerlegung zurück. Das Problem des Overheads lässt sich jedoch weitgehend vermeiden, wenn man die *systematische Rekombination* von Elementaroperationen als eigenes Grundkonzept einführt.

Wenn die Rekombination orthogonaler Elementaroperationen bei der Bearbeiter-Instanz der Prozedur- oder RPC-Aufrufe erfolgt, ergibt sich nur ein geringer bis verschwindender Overhead im Vergleich zur direkten Implementierung jeglicher Kombination. Der wesentliche Vorteil der Zerlegung besteht darin, dass im Regelfall nur der Programmcode für die wenigen Elementaroperationen implementiert werden muss, nicht dagegen für alle vorkommenden Kombinationen, die oft wesentlich zahlreicher ausfallen.

2.5. Das Problem der „Eier legenden Wollmilchsau“

Größere Softwaresysteme, die über längere Zeit benutzt und erweitert wurden oder die für sehr große Anwendungsfelder und -bandbreiten ausgelegt wurden, zeigen oft ein Phänomen auf, das salopp mit „Featuritis“ bezeichnet wird. Solche Alleskönner haben manchmal Schwierigkeiten, ganz einfache grundlegende Aufgaben auf einfache und effiziente Weise zu erledigen. Die Softwarestruktur ist im Vergleich zu einfachen Programmen, die nur die Grundaufgaben abdecken, oder im Vergleich zu Spezialisten meist deutlich aufgebläht (Overhead) und schwerer zu durchschauen (Phänomen des softwaretechnischen *Wildwuchses*).

Um dieses Problem handhabbar zu machen, schlage ich die Verwendung unterschiedlicher *Modelle* vor.

Modelle, die sich nicht gegenseitig beeinflussen, sind *orthogonal* zueinander und können gleichzeitig von einer Baustein- oder Nest-Instanz implementiert bzw. verwendet. werden. Mehrere orthogonale Modelle können gleichzeitig durch dieselbe Schnittstelle verwirklicht werden.

Zwischen nicht-orthogonalen Modellen sollte möglichst eine hierarchische Inklusions-Beziehung in dem Sinne herrschen, dass man ein Modell als Spezialfall eines anderen Modells betrachten kann; oft ist das weniger mächtige Modell auch einfacher. Derart zusammen hängende Modelle werden eine *Modell-Familie* genannt. Mitglieder verschiedener Familien müssen stets orthogonal zueinander sein. Als Beispiel für eine Modell-Familie werden in Abschnitt 3.2 verschiedene Zugriffs-Modelle auf Nester wie `singleuser` oder `multiuser` behandelt werden.

Verschiedene Modelle einer Familie müssen sich nicht unbedingt durch verschiedene Schnittstellen oder -Varianten unterscheiden; die Unterschiede können beispielsweise auch in der Semantik, in Zusicherungen, oder in nicht-funktionalen Eigenschaften liegen. Falls verschiedene Schnittstellen benutzt werden oder sogar erforderlich sind, sollte versucht werden, wenigstens noch das Prinzip der Erweiterungs-Generizität einzuhalten, wenn schon die Prinzipien der universellen und kompositorischen Generizität nicht mehr ausschließlich zum Zuge kommen können.

Eine Nest- oder Baustein-Implementierung braucht nicht unbedingt alle möglichen Modelle einer Familie zu unterstützen. So kann man beispielsweise mit der Implementierung der einfacheren Modelle beginnen und diese erst später und bei Bedarf auf die komplizierten Modelle erweitern oder für die komplizierteren Modelle alternative Implementierungen vornehmen (deren höheren Kosten müssen dann von den Verwendern der einfacheren Modell-Variante nicht bezahlt werden).

Welche Modelle einer Familie von einer Nest- oder Baustein-Implementierung im Einzelfall unterstützt werden, wird als *Kompetenz* (*competence*) dieser Implementierungs-Instanz bezeichnet. Demgegenüber steht das *Verhalten* (*habit*) einer konkreten Aufrufer-Instanz; damit werden die Anforderungen an die Kompetenzen der Bearbeiter-Instanz bezeichnet.

Die Kompetenzen einer Bearbeiter-Instanz müssen zum Verhalten einer Aufrufer-Instanz *kompatibel* sein. Die Kompetenzen und das Verhalten von Nest- und Baustein-Instanzen werden je Baustein-Art in Form von *Attributen* angegeben. Die Kompatibilität aller beteiligten Familien kann dann bei der Verdrahtung konkreter Baustein-Instanzen automatisch getestet werden, wobei inkompatible Verdrahtungen zurückgewiesen werden.

Attribute können entweder *statisch* oder *dynamisch* sein: statische Attribute eines Baustein-Typs ändern sich nie (sie hängen nur vom Baustein-Typ oder der gewählten Implementierung ab), dynamische Attribute können potentiell von der konkreten Instantiierung und/oder von der Verdrahtung mit anderen Bausteinen abhängen, ändern sich jedoch nicht während der Lebensdauer einer Baustein-Instanz. Werte, die sich während der Lebensdauer einer Baustein-Instanz ändern können, stellen keine Attribute dar, sondern gehören zum *Zustand* der Baustein-Instanz.

Attribute werden insbesondere zur Unterscheidung verschiedener Modelle eingesetzt und in `Schreibmaschinenschrift` gesetzt.

2.6. Automatismen

Die Grundidee der Automatisierung wird seit Jahrhunderten erfolgreich zur Reduktion von Aufwand und Kosten eingesetzt.

Im Kontext der hier vorgestellten Architektur bedeutet Automatisierung, dass vorzugsweise von deskriptiven Methoden Gebrauch gemacht wird, um einen Mechanismus selbsttätig auszulösen, der immer wiederkehrende Vorgänge selbsttätig bearbeitet.

Die Implementierung vieler Arten von Automatismen ist

eine Frage von konkreten Strategien, die in speziellen Bausteinen (`strategy_*`) lokalisiert werden sollten (vgl. Abschnitt 4.2.2).

Zur Erstellung von Bausteinen sind weitere Automatismen von großem Nutzen, die über den Funktionsumfang üblicher Werkzeuge wie Compiler oder Debugger hinaus gehen sollten. Viele mit den hier vorgestellen (Schnittstellen-)Mechanismen zusammen hängende Konstruktionsvorgänge lassen sich automatisieren.

Die in Abschnitt 2.3.2 vorgestelle späte Bindung an konkrete Aufrufmechanismen kann beispielsweise auf folgende Weise automatisiert werden:

Der Programmierer gibt ein statisches Baustein-Attribut an, mit dem er sein Denk-Modell deklariert, mit dessen Hilfe er den Programmcode entwickelt hat. Das Attribut kann folgende Werte annehmen:

`code_nolock` Der Programmierer tut so, als gäbe es nur einen einzigen Kontrollfluss, der den Baustein betreten dürfte[29]; er kümmert sich also überhaupt nicht um Parallelität. Dies hat zur Folge, dass aus Sicherheitsgründen niemals ein weiterer Kontrollfluss den Baustein betreten darf, selbst wenn der bereits eingetretene Kontrollfluss eine lange dauernde blockierende Operation aufruft.

`code_monitor` Wie vorher, nur ist der Programmierer sich immerhin der Tatsache bewusst, dass mehrere Kontrollflüsse vorkommen können. Jeder Aufruf einer möglicherweise blockierenden Operation (z.B. `wait`, siehe Abschnitt 3.3.2) führt automatisch zu einer Entblockierung des Zugriffsschutzes gegenüber anderen Kontrollflüssen. Der Programmierer ist für die Beachtung der damit verbundenen Effekte verantwortlich; insbesondere ist ihm bewusst, dass kritische Abschnitte genau an der Stelle von Operationsaufrufen aufgehoben werden (analog zum Monitor-Konzept).

`code_reentrant` Der Programmierer ist sich dessen bewusst, dass mehrere Kontrollflüsse den Baustein asynchron betreten können. Er ist selbst für die Sicherung kritischer Abschnitte und für das Setzen von Locks verantwortlich.

Ein zu syntaktischer Analyse fähiger und die Schnittstellen-Konventionen kennender Quelltext-Präprozessor extrahiert diesen Attribut-Wert aus dem Quelltext und generiert bei Bedarf (je nach eingestelltem Aufruf-Mechanismus) automatisch Lock-Operationen zur Klammerung kritischer Abschnitte, Fallunterscheidungs-Kontrollstrukturen zum Demultiplexen eingehender (L)RPC-Aufrufe, und so weiter. Auf diese Weise lässt sich jedes Programmiermodell mit jedem Schnittstellen-Mechanismus kombinieren[30]; bei Verwendung von statischen oder Inline-Prozeduraufrufen werden durch den Präprozessor mehrere Quelltexte verschiedener Bausteine zu einem einzigen Kombi-Baustein fest zusammengeschweisst[31]. Bei geeigneter sorgfältiger Konstruktion und Verwendung gut optimierender Compiler lässt sich durch Inline-Prozeduraufrufe jeglicher Schnittstellen-Aufwand fast vollständig eliminieren; dies ist insbesondere zum Anschluss von Prüf- und Sicherheits-Bausteinen oder kleineren Trivial-Bausteinen nützlich.

2.7. Zugriffsrechte und Schutzmechanismen

2.7.1. Grundlegende Betrachtung: Mandate

Die Rechteverwaltung in Betriebssystemen basiert bei den meisten praktisch eingesetzten Modellen [Lan81] auf *Rechte-Relationen*, die *Subjekte* und *Objekte* mit einander in Beziehung setzen; typischerweise lassen sich derartige Relationen zwischen Subjekten und Objekten tabellarisch darstellen.

Ich möchte für die Rechteverwaltung einen Ansatz vorstellen, der sich bei geeigneter Interpretation ebenfalls als Subjekt-Objekt-Schema auffassen lässt, dabei jedoch beliebige Dinge als Subjekte bzw Objekte auffassen kann.

Zunächst ist zu fragen, *was* zu schützen ist. Hierauf sind mehrere Antworten möglich. Eine möglichst allgemeine Antwort lautet, dass es sich um *Informationen* handelt, die zu schützen sind. Dies umfasst die klassischen Objekte (die auf jeden Fall Informationsträger darstellen), bedeutet aber nach meiner Ansicht etwas mehr. In einem Betriebssystem gibt es Informationen, die nicht unbedingt in Form zugreifbarer Datenobjekte vorliegen müssen. Typische Beispiele hierfür sind die Menge aller möglichen Schlüsselwerte eines geheimen kryptographischen Schlüssels, oder die Rechte anderer Subjekte (die nicht unbedingt explizit repräsentiert zu sein brauchen), oder die Kompetenzen oder das Verhalten anderer Subjekte, oder andere Subjekte schlechthin; Subjekte können in anderem *Kontext* wiederum Objekte darstellen. Informationen über Objekte lassen sich ggf. auch durch *Schlussfolgern* oder *Ableiten* erhalten. Ich schlage deshalb vor, statt der Begriffe Subjekt und Objekt andere Begriffe zu verwenden, die durch die folgenden Überlegungen begründet sind:

Der *Zweck* eines Betriebssystems ist die *Ausführung von Operationen*; dies geschieht in der hier vorgestellten Architektur (vgl. Abschnitt 3.2 und Kapitel 6) durch *Bearbeiter-Instanzen* im Auftrag von *Aufrufer-Instanzen*. Da eine Aufrufer-Instanz wiederum im Auftrag mehrerer anderer Aufrufer-Instanzen handeln kann, ergeben sich zwei Fragestellungen:

1. Wer soll natürlicherweise das Subjekt darstellen, das einen bestimmten Auftrag veranlasst?

[29] Auch im `singleuser`-Modell (Abschnitt 3.2) können asynchrone Kontrollflüsse durch `notify_*`-Operationen (Kapitel 5) entstehen.

[30] Falls man `code_nolock` mit (L)RPC kombiniert, braucht man selbstverständlich keine Lock-Operationen einzufügen, da es dann ja nur einen einzigen (automatisch eingefügten und bei der Initialisierung gestarteten) Kontrollfluss gibt; durch einheitliche Verwendung dieser Konfiguration bei allen Bausteinen erhält man das klassische CSP-Modell, das zu Zwecken der Fehlersuche und -Eingrenzung deutliche Vorteile bringt, ansonsten aber die potentielle Parallelität unnötig einschränkt. Die Kombination von `code_reentrant` mit (L)RCP stellt insofern eine gewisse Verschwendung dar.

[31] Hierbei sind (interne) Bezeichner (bzw. zu internen Bezeichnern werdende ursprünglich öffentliche Bezeichner) mit einem Versions-Kennzeichen zu verändern, um die mehrfache Instantiierung desselben Baustein-Typs in einem Kombi-Baustein möglich zu machen. Details werden in dieser Arbeit nicht behandelt.

2. Wer soll natürlicherweise das Objekt darstellen, das den Auftrag ausführen soll?

Die erste Frage ist auf verschiedene Weisen beantwortbar. Daher schlage ich die Einführung eines anderen Begriffes statt „Subjekt“ vor, nämlich das *Mandat* (engl. mandate). Operationen geschehen grundsätzlich aufgrund eines Mandates; im allgemeinen kann dabei eine Instanz (bzw ein Subjekt in bisheriger Terminologie) mehrere verschiedene Mandate *wahrnehmen*; auch kann es vorkommen, dass verschiedene Instanzen aufgrund des gleichen Mandats handeln. Beispiele hierfür finden sich in der realen Welt im Rechtswesen: ein und derselbe Rechtsanwalt kann gleichzeitig verschiedene Mandate für verschiedene Mandanten wahrnehmen. Ein Mandant kann verschiedene Mandate gleichzeitig an verschiedene Rechtsanwälte oder an den gleichen Rechtsanwalt vergeben. Er kann dasselbe Mandat aber auch an mehrere Rechtsanwälte gleichzeitig vergeben, beispielsweise bei Anwalts-Gemeinschaften oder Kanzleien. Mandate können *vertreten* oder *weitergereicht* werden. Im Rechtswesen können auch *Untermandate* vergeben und aufgeteilt werden; von der letzteren Möglichkeit habe ich hier wegen der damit verbundenen Komplexität und Folgeeffekte vorläufig Abstand genommen[32]. Die Menge aller möglichen Mandate sollte durch einen abstrakten Datentyp mit ausreichend großem Wertevorrat[33] dargestellt werden. Mandate haben keine festliegende Interpretation, sondern werden lediglich zwischen Baustein-Instanzen weiter gereicht. Alle Operationen auf Mandaten wie z.B. ihre Erzeugung gehören somit zu den Strategien, die der Implementierer eines Bausteins selbstständig bestimmen kann. Mandate stellen somit eine weitere Hilfsabstraktion dar, die jedoch in die Nest-Abstraktion eingebettet ist.

Die zweite Frage ist im Kontext der hier vorgestellten Architektur relativ schnell zu beantworten: ein Objekt ist auf jeden Fall die Bearbeiter-Instanz, an die der zu bearbeitende Auftrag gerichtet wird. Diese Nest-Instanz zerfällt jedoch nicht nur in Unterobjekte wie z.B. einzeln adressierbare Bytes, sondern stellt auch wegen der eingangs aufgeführten Problematik nicht alle Informationen dar, die nach Möglichkeit geschützt werden sollten. Die folgende Begriffsbildung kommt diesem Ziel etwas näher:

Zu schützen ist die *Ausführung* von Operationen. Die Menge aller ausführbaren Operationen wird *Operationenmenge* genannt; dies sind alle möglichen Kombinationen von Operations-Bezeichnern mit ihren Eingabe-Parameter-Werten, oder in anderen Worten die Vereinigung aller kartesischen Produkte aller Operationsnamen mit ihren Parameter-Wertemengen (egal wie viele es sein mögen). Auch wenn man die Menge aller möglichen Parameter-Versorgungen als endlich betrachtet (indem man beispielsweise die Menge aller vorkommenden Zeichenketten-Parameter beschränkt), ist die Operationenmenge i.a. sehr groß. Eine dazu äquivalente Darstellung ist die Beschreibung als Menge aller möglichen Bitstrings, die in einem generischen Operations-Nest (vgl. Kapitel 6) auftreten können.

Eine *Schutzmenge* ist eine Teilmenge der Operationenmenge eines gegebenen Systems. Schutzmengen sind als Mengen aller (im Sinne irgend eines ausserhalb definierten Zulässigkeits-Begriffes) zulässigen Operationen zu interpretieren.

2.7.2. Spezifikation von Schutzrechten

Der Begriff der Schutzmenge vereinfacht die Abstraktion irgendwelcher Schutz- und Zulässigkeitsbegriffe auf mathematisch sehr einfach fassbare und hochgradig flexible Weise[34]. Da Schutzmengen i.a. zu groß für eine tabellarische Darstellung sind, braucht man irgendwelche Spezifikationsmechanismen, die sie beschreiben. Der Zweck eines Spezifikationsmechanismus besteht einerseits darin, eine Schutzmenge für menschliche Leser verstehbar und nachvollziehbar komprimiert darzustellen, andererseits das Enthaltens-Problem in einer Schutzmenge effizient auf Rechnern auswertbar zu machen.

Als Spezifikationsmechanismen kommen sehr viele konkrete Realisierungen (z.B. passend eingeschränkte Prädikatenlogiken oder andere Kalküle) in Betracht, deren Diskussion den Rahmen dieser Arbeit sprengen würde. Es sind ohne weiteres auch militärische Schutzmodelle (vgl. [Lan81]) oder mehrere voneinander unabhängige Schutzmodelle gleichzeitig einsetzbar.

2.7.3. Prüfung von Schutzrechten

Die Prüfung von Schutzrechten kann in speziellen Prüf-Bausteinen `check_*` erfolgen, die sich prinzipiell an beliebigen Stellen einer Baustein-Hierarchie einfügen lassen. Dort weisen sie die im Sinne des jeweiligen Schutzmodells unzulässigen Operationen zurück. Falls man in speziellen Anwendungsbereichen wie z.B. Echtzeit-Steuerungen kein Schutzmodell benötigt, kann man auf die Instantiierung von `check_*`-Bausteinen vollkommen verzichten und damit jeglichen Overhead einsparen.

Die konkrete Realisierung eines Schutzmechanismus ist eine Frage der Strategie, an welchen Stellen einer Baustein-Hierarchie welche Arten von Prüfungen auf Basis welcher Mandate durchgeführt werden sollen. Zu beachten ist, dass zur Laufzeit beliebige Baustein-Instanzen neu instantiiert und neue Verdrahtungen erzeugt werden können, mit denen ein festes Schutzkonzept u.U. umgangen werden kann. Beim Einsatz kompositorischer Generizität steckt ein Teil der im System vorhandenen Informationen in der Verdrahtung der Bausteine, die hochgradig flexibel und dynamisch änderbar ist. Zur Lösung dieses Problems müssen die Instantiierungs-Operationen von `control`-Instanzen (siehe Abschnitt 4.2.1) in das jeweilige Schutzkonzept mit einbezogen und überwacht werden. Zur mathematischen Beschreibung der dynamischen Eigenschaften von Baustein-Netzwerken eignet sich eventuell die Menge aller mög-

[32] Ob und in welchen Fällen Untermandate vorteilhafte Beschreibungen darstellen, dürfte eine interessante zu untersuchende Frage sein. Strukturell sind Untermandate zwar ähnlich zu Gruppen von Subjekten, die Erzeugung von Untermandaten kann jedoch rein dynamisch zur Laufzeit geschehen, die wiederum in weitere Untermandate zersplittert werden könnten. Ob und wie weit solche Modelle vorhersagbares und intuitiv für Menschen leicht begreifbares Verhalten zeigen, muss noch untersucht werden.

[33] Nach heutigen Maßstäben sind hierzu mindestens 64 Bit reservierter Platz erforderlich.

[34] Das klassische Konzept der Rechte-Tabellen läßt sich als Spezialfall einer Schutzmenge darstellen: man bilde die Menge aller Tripel (`op`,`subject`,`object`), die durch eine Subjekt-Objekt-Zuordnungstabelle beschrieben wird, und fülle gegebenenfalls durch die Tablle nicht beschriebene weitere Parameter-Komponenten der Operationen mit der Menge aller möglichen Werte auf.

lichen Baustein-Konfigurationen, die durch `control` erzeugbar sind.

2.7.4. Sicherstellung von Schutzrechten

In der Literatur über Schutzmechanismen in Betriebssystemen werden vor allem zwei Mechanismen zur Sicherstellung eines Schutzkonzepts verwendet:

1. Sicherstellung durch Zugriffsbeschränkung mittels eines Typkonzepts einer höheren Programmiersprache und/oder durch vertrauenswürdige Compiler [B+95]

2. Sicherstellung durch Hardware, inbesondere durch die MMU

Typkonzepte von Programmiersprachen haben zwar den geringsten Laufzeit-Overhead, bieten jedoch i.a. nicht das Maß an Sicherheit gegen Umgehung durch böswillige Angriffe, wie dies MMU-Hardware ermöglicht, indem sie Zugriffe auf fremde Schutzbereiche physikalisch unterbindet.

Beide Mechanismen lassen sich in der hier vorgestellten Architektur einsetzen, wobei Mechanismus 1 durchaus Vorteile bei der Erhöhung der Zuverlässigkeit eines Systems gegen versehentliche Fehlfunktionen mit geringem Overhead besitzt; wirkliche Sicherheit gegen bösartige Angriffe kann jedoch nur der Mechanismus 2 bieten.

Die Realisierung eines Schutzmodells ist daher von der (dynamischen) Einteilung des Betriebssystems in Schutzbereiche abhängig (siehe Abschnitt 4.2). Im einen Extremfall kann das gesamte System in einem einzigen Schutzbereich ablaufen, im anderen Extremfall kann jede einzelne Baustein-Instanz in einem eigenen Schutzbereich ablaufen; es sind beliebige Zwischenstufen möglich. Sinnvollerweise sollten Zugriffsrechte vor allem an den Grenzen zwischen Schutzbereichen geprüft werden, sofern dieser Aufwand in Kauf genommen werden soll.

2.8. Performanz-Fragen

Nicht nur in der Betriebssystem-Literatur wird der Performanz eines Systems ein hoher Stellenwert eingeräumt. Das Ziel hoher Performanz kann gelegentlich mit anderen Zielen in Widerspruch stehen. Durch konsequente Anwendung der Trennung zwischen Mechanismen und Strategien lässt sich dieser Zielkonflikt jedoch in vielen Fällen vermeiden, wenn man beim Entwurf der Mechanismen auf deren performante Implementierbarkeit achtet (die tatsächlich erzielte Performanz hängt jedoch u.a. von den Kenntnissen, Fähigkeiten und Methoden des Implementierers ab[35]).

2.8.1. Performanz von Elementaroperationen

In Abschnitt 2.4.2 wurde die systematische Rekombination von Elementaroperationen methodisch motiviert. Die Performanz stellt ein weiteres Motiv dar.

Aufrufer benutzen Elementaroperationen häufig in *Bündeln*, d.h. in sukzessiven Folgen, die einem bestimmten Programmier-Muster folgen. Gebündelte Aufrufe von Elementaroperationen enthalten i.A. triviale *Muster* von Programmlogik: nur bei Erfolg der ersten Operation wird die nächste aufgerufen, wobei Ergebnis-Parameter der ersten Operation an die zweite weiter gegeben werden. Es kommen viele verschiedene Muster in Betracht.

Die Idee besteht nun darin, die Implementierung häufig vorkommender Muster (siehe z.B. Abschnitt 3.3.8) von der Implementierung der Elementaroperationen zu separieren. Hierfür eignen sich dedizierte Bausteine `pattern_*`, die sich mit beliebigen anderen Bausteinen kombinieren lassen. Dies bewirkt praktisch eine Schnittstellen-Erweiterung, die jedoch keine neue Funktionalität einführt, sondern lediglich der Bequemlichkeit und teilweise auch der Performanz-Verbesserung dient. Insbesondere bei Verwendung von RPC-Mechanismen in `remote`-Bausteinen sorgt die gebündelte Übertragung einer einzigen Operation über ein Netzwerk für die Einsparung von Nachrichten und Latenzzeit, wenn sie erst beim Server in Elementaroperationen aufgespalten und interpretiert wird. Damit dies auch in größeren Baustein-Hierarchien nutzbar wird, sollten relativ einfache Bausteine wie `selector` oder `union`, die fast gar nichts tun außer Operationen mit geringen Modifikationen weiterzureichen, gebündelte Muster als passend modifizierte Bündel an die nächste Instanz weiterreichen. Dies ist jedoch keine absolute Pflicht[36].

Zur Umsetzung dieser Idee sollte ein Baustein in seinen Verhaltens-Attributen die Muster angeben, die er verwenden möchte. Die Instantiierungs- und Verdrahtungs-Logik (`control` und `strategy_*`, siehe Abschnitt 4.2) kann dann die passenden Muster-Bausteine automatisch an geeigneten Stellen einfügen und dazwischen schalten, so dass die Verwendbarkeit der Muster in jedem Fall sicher gestellt ist, ohne sich darum sorgen zu müssen. Falls der untergeordnete Baustein in seinen Kompetenz-Attributen angibt, mit einem bestimmten Muster umgehen zu können, wird dieser Zwischenschritt ggf. ausgelassen. In günstigen Fällen ergibt sich dadurch eine Kette von bündelungsfähigen Verdrahtungen, die sich auch über Speicher-Hierarchien hinweg erstreckt und so die Performanz deutlich verbessert.

Eine weitere, in Athomux [Ath] favorisierte Methode besteht darin, anstelle nicht eigens implementierter Kombi-Operationen eine Default-Version vom Präprozessor einsetzen zu lassen, die eine wohldefinierte Standard-Semantik durch sequentiellen Aufruf der jeweiligen Elementaroperationen implementiert. Dadurch werden Kombi-Operationen überall mit einheitlicher Semantik verfügbar; die ausdrückliche Implementierung erfolgt im Regelfall lediglich aus Performanz-Gründen.

2.8.2. Zero-Copy-Architektur

Frühere IO-Schnittstellen wie die von Unix [RT74] oder [Che87] besaßen eine implizite Kopier-Semantik. Neuere Entwicklungen wie [BS96, PDZ99, PDZ00] vermeiden das Herstellen von Kopien in möglichst vielen Situationen. Es

[35] In neuerer Zeit haben Performanz-Fragen ein erhöhtes Interesse in der Literatur gefunden. Trotz des Bemühens um Vergleichbarkeit verschiedener Architekturen und Vorschläge ist diese nicht immer gegeben; das Stichwort des Vergleiches zwischen Äpfeln und Birnen findet sich z.B. in [Lie95b].

[36] Missachtung kann jedoch teilweise erhebliche Einbußen bei der Performanz nach sich ziehen. Wenn es dagegen auf schnelle Implementierung neuer Funktionalität ankommt, ermöglicht das Weglassen von Bündeln sogenanntes „rapid prototyping“.

ist bekannt, dass dies dramatischen Einfluss auf die Performanz haben kann.

Der in Kapitel 3 vorgestellte Entwurf definiert daher die IO-Funktionalitiät der Nest-Schnittstelle mit Hilfe von Referenz-Semantik; die Übergabe von physischem Speicher über Schnittstellen hinweg erfordert dies ohnehin. Damit ist kopierfreie Übergabe von Daten über beliebig große Baustein-Hierarchien hinweg möglich. Falls in besonderen Fällen eine Kopier-Semantik gewünscht wird oder notwendig ist, kann sie leicht als interne Strategie durch einen Baustein implementiert werden.

2.8.3. Das Background-IO-Konzept

Durchsatz-begrenzende *Flaschenhälse* treten oft in komplexen Netzwerken von Baustein-Verdrahtungen oder in verteilten Systemen auf. Wenn ein Falschenhals auf physikalischen Bedingungen beruht (z.B. bei Festplatten-IO), lässt er sich prinzipiell nicht entfernen (außer durch Kauf schnellerer und teurerer Hardware). Man kann jedoch Methoden entwicklen, um mit derartigen Flaschenhälsen besser leben zu können, d.h. zu versuchen, „das beste draus zu machen“.

IO-Prioritäten dienen diesem Zweck. Mittels IO-Prioritäten lassen sich wichtige von unwichtigen IO-Operationen trennen. Ich schlage die Verwendung folgender IO-Prioritässtufen vor:

`prio_background` Der IO-Auftrag kann irgendwann ausgeführt werden, beispielsweise wenn ansonsten ungenutzte Übertragungs-Bandbreite frei geworden ist. Die Wartezeit bis zur Ausführung darf beliebig lang sein; es werden keinerlei Fairness-Bedingungen garantiert oder erwartet.

`prio_normal` Der IO-Auftrag muss prinzipiell gleichberechtigt zu anderen Aufträgen bearbeitet werden, die ebenfalls `prio_normal` haben. Der Auftrag muss nach endlicher Zeit erledigt sein (kein Verhungern). Permutationen der Abarbeitungs-Reihenfolge sind im Rahmen dieser Bedingungen erlaubt.

`prio_urgent` Der IO-Auftrag ist anderen vorzuziehen. Die größere Wichtigkeit hat eine logische Begründung (beispielsweise Sicherung wichtiger Meta-Daten).

Die Idee besteht darin, bei Verstopfung eines IO-Flaschenhalses Aufträge mit `prio_background` *vollkommen* zu verdrängen. Bei geeigneter Realisierung lässt sich dies so gestalten, dass Hintergrund-Aufträge die normalen IO-Aktivitäten überhaupt nicht stören.

Bei Festplatten kann man dies folgendermaßen realisieren: erst werden alle Aufträge mit `prio_urgent` und `prio_normal` abgearbeitet. Danach folgt eine spekulative Wartepause in der Größenordnung von 10ms bis 100ms, nach deren Ablauf erst mit dem Abarbeiten der Hintergrund-Operationen begonnen wird. Falls während dieser Wartepause erneute Operationen mit `prio_urgent` oder `prio_normal` eintreffen, werden *keine* Hintergrund-Operationen gestartet, sondern die höher priorisierten sofort vorgezogen; danach beginnt die Wartepause von vorn. Der Sinn dieser Wartepause besteht darin, den mechanisch bewegten Schreib-Lesekopf nicht zu verstellen, falls eine Anwendung ihre IO-Anforderungen zwar in Gruppen (Bursts) stellt, zwischen den Anforderungen aber sehr kurze Lücken (z.B. wegen kurzer Rechenzyklen) einlegt. Ohne die Wartepause könnte der mechanische Arm der Festplatte während der sehr kleinen Lücke zwischen normal-priorisierten Burst-Operationen (wenige Mikro- oder Nanosekunden) auf die Position der Hintergrund-Operation verstellt werden, was bei heutigen Festplatten im Mittel etwa 5ms bis 10ms dauert, und hernach wieder auf die Position des Burst-Prozesses zurück gestellt werden. Da das mechanische Verstellen um mehrere Größenordnungen langsamer geht als das sequentielle Lesen von Burst-Datenblöcken, würde dieses andauernde mechanische Hin-und-Herstellen zu einem Thrashing-Effekt mit einem Zusammenbruch des Daten-Durchsatzes führen. Die Wartepause löst dieses Problem durch eine Spekulation auf das Eintreffen weiterer wichtiger Operationen. Wenn sich diese Spekulation nach einer relativ langen Zeit von einigen Vielfachen der mittleren Zugriffszeit der Festplatte nicht erfüllt hat, dann ist es wegen der bekannten empirisch beobachteten zeitlichen Ungleichverteilung der IO-Anforderungen sehr unwahrscheinlich, dass ausgerechnet dann weitere IO-Operationen mit normaler Priorität eintreffen, während der Kopf für die Hintergrund-Aufträge verstellt wird. Falls dieser Fall trotzdem einmal eintreten sollte, dann verschlechtert er den ohnehin sehr geringen Durchsatz der normal-priorisierten Aufträge nicht mehr wesentlich.

Wenn man das Konzept eines den Normalbetrieb praktisch nicht störenden Hintergrund-IO-Betriebes in die Nest-Schnittstelle aufnimmt, ergeben sich neuartige Realisierungen für das Problem der Persistenz über Speicherlücken hinweg. So schadet es beispielsweise in einem Buffer-Cache nicht, wenn man jegliche geänderten Puffer *sofort* nach Bekanntwerden der Änderung (Wechsel in den „dirty“-Zustand des Puffers) einen IO-Auftrag zum Abspeichern mit Hintergrund-Priorität generiert. Falls der Hintergrund-Auftrag sehr lange nicht ausgeführt wird, dann entsteht der gleiche Effekt wie beim konventionellen Puffern ohne IO-Auftrag. Werden Hintergrund-Aufträge auf „heißen“ (d.h. oft zugegriffenen) Seiten ausgeführt, dann führt dies in der Summe zwar zu einer höheren Belastung des IO-Kanals mit Aufträgen, diese füllen jedoch nur die ansonsten ungenutzte Rest-Bandbreite des Kanals, die ansonsten „verschwendet“ worden wäre.

Im Endeffekt entsteht dadurch eine zeitnahe Sicherung transienten Speichers auf persistente Hintergrund-Medien, die sich automatisch an die *aktuell verfügbare* Bandbreite adaptiert. Bei hoher verfügbarer Bandbreite werden spekulative Sicherungs-Vorgänge ausgeführt, die zu einer besseren Sicherheit gegen unvorhergesehene Ereignisse wie Stromausfall führen.

Die Hintergrund-Priorität lässt sich weiterhin auch zum spekulativen Vor-Laden von Caches etc. nutzen (read-ahead). Spekulatives Ladevorgänge können wegen der Hintergrund-Priorität jederzeit von wichtigeren Vordergrund-Aktivitäten verdrängt werden. Die Realisierung solcher Preloading-Strategien kann beispielsweise durch zwischengeschaltete Verhaltens-Beobachter-Bausteine erfolgen, die das IO-Verhalten einer Anwendung

beobachten und analysieren und daraus Schlüsse über das erwartete zukünftige Verhalten der Anwendung ziehen.

In Abschnitt 3.3.1 wird näher beschrieben, wie sich die IO-Priorität bereits erteilter Aufträge nachträglich erhöhen lässt. Diese dynamische Änderungsmöglichkeit ist wichtig, da richtig spekulierte Anforderungen jederzeit zu „echten“ Anforderungen einer Anwendung werden können.

3. Nester

Die Abstraktion des *Nestes* stellt ein konkretes *Speichermodell* dar. Betriebssystem-Konstrukteure sind traditionell gewohnt, dass ihnen die Hardware eines Rechners oder eines Rechner-Typs ein bestimmtes Speichermodell vorschreibt oder zumindest in relativ engen Grenzen sehr nahe legt. Diesem Zwang versuche ich an einigen Stellen so zu entkommen, dass dadurch keine grundlegende Verschlechterung der Performanz-Eigenschaften der Hardware ausgelöst wird.

Eine *Nest-Instanz* ist ein *logischer Adressraum*, der von der Adresse 0 bis zu einer Maximalgröße (mit mindestens 64Bit[1]) laufen kann. Mit logischen Adressen darf Adress-Arithmetik getrieben werden; es werden *Bytes*[2] adressiert.

Ein Nest dient vor allem zur Abbildung von logischen Byte-Adressen auf *physische* Byte-Adressen von *Datenblöcken*. Damit soll u.a. das bekannte Konzept eines virtuellen Benutzer-Adressraums nachgebildet werden; dies ist jedoch prinzipiell unabhängig davon, ob MMU-Hardware in einem Rechner vorhanden ist oder nicht. Ein physischer Datenblock hat eine begrenzte Länge, innerhalb deren er sich durch Maschinenbefehle des Prozessors ansprechen lässt und innerhalb deren Adress-Arithmetik auf Basis von physischen Byte-Adressen getrieben werden darf.

Nester können prinzipiell sowohl mittels Hardware als auch mittels Software, oder durch eine Kombination von beidem dargestellt werden. Die Realisierung in Software erfolgt vorzugsweise in Form einer *abstrakten Schnittstelle*.

3.1. Arten von Nestern

Es wird zwischen *statischem* und *dynamischem* Nest unterschieden: während sich ein statisches Nest ähnlich wie ein Unix-Device verhält und auch nur im Zusammenhang mit Peripheriegeräten o.ä. vorkommen sollte, basiert das restliche System vorzugsweise auf dynamischen Nestern.

Ein dynamisches Nest kann im Unterschied zu einem konventionellen File abfragbare und veränderbare Löcher im Definitionsbereich der partiellen Abbildung von logischen Adressen auf physische Adressen von Datenblöcken enthalten, ähnlich wie ein Sparse File unter Unix (vgl. auch [Fot61, Lie95a]), jedoch wird diese Eigenschaft vom gesamten Betriebssystem auf allen Ebenen außer Low-Level-Gerätetreibern unterstützt und auch intensiv genutzt.

[1] Noch größere logische Adressräume, etwa mit 128 Bit oder mehr, haben durchaus praktische Anwendungen: das bekannte Segmentierungs-Modell (vgl. Multics [Org72] oder die Speichermodelle von Intel-Prozessoren) lässt sich durch einen linearen logischen Adressraum darstellen, indem man ein (Segmentselektor,Offset)-Paar bildet, das insgesamt als logische Adresse eines (löchrigen) linearen Adressraums interpretiert wird. Wenn der Segmentselektor beispielsweise 64 Bit und der Offset ebenfalls 64 Bit umfasst, dann reicht ein logischer Adressraum von 128 Bit für eine *äquivalente* Darstellung des Segmentierungs-Modells durch den linearen Adressraum (vgl. auch das Thunk-Modell von Microsoft beim Übergang vom 16Bit-MSDOS-Speichermodell auf 32 Bit).

[2] In früheren Zeiten war der Begriff Byte mehrdeutig, da er auch andere Bitfolgen als das Oktett bezeichnen konnte. Der Begriff ist inzwischen standardisiert und bezeichnet genau 8 Bit.

Als neuartige Elementaroperation kommt die *transparente Verschiebung* von logischen Adressbereichen hinzu. Eine Verschiebung bewirkt, dass ein Teil des Definitionsbereiches der partiellen Abbildung von logischen Adressen auf Adressen von physischen Datenblöcken so verschoben wird, dass dieselben Datenblöcke anschließend unter einem Adress-Offset (im Vergleich zu den vorigen Adressen) im logischen Adressbereich erreichbar sind. Im allgemeinen können durch Verschiebe-Operationen Löcher im Definitionsbereich entstehen und/oder geschlossen werden, eventuell können am Ziel der Verschiebung auch Datenblöcke „verloren gehen“, d.h. sie werden nicht mehr vom Nest adressiert (können aber i.a. noch solange weiter existieren, solange noch dynamische Referenzen darauf bestehen).

In konventioneller Terminologie bedeutet eine Verschiebeoperation, dass je nach Vorzeichen des Verschiebe-Offsets und Länge des verschobenen Bereichs eine Insert- oder Delete- oder Move-Operation in einer Datei, in einer Datenbank, oder allgemein gesprochen in einem Nest durchgeführt wird. Ich verwende daher nur noch die Begriffe Nest und Verschiebe-Operation, um diese Sachverhalte zu charakterisieren.

Die Verschiebe-Operation sollte im Idealfall *transparent* erfolgen. Damit ist gemeint, dass nur diejenigen Teile des Gesamtsystems von einer Verschiebung Kenntnis erhalten, die unbedingt davon betroffen sind; für die anderen Teile ergeben sich durch die Verschiebung keine Änderungen.

Hierzu ein Beispiel (siehe auch Abschnitt 4.1.4): aus dem oberen Ende eines Original-Nestes sei ein Teil-Nest ausgeschnitten worden. Die Ausschneide-Operation bewirkt, dass die zugrunde liegenden Datenblöcke sowohl im Original-Nest als auch im ausgeschnittenen Teil-Nest auf Anforderung erscheinen, in letzterem jedoch unter neuen logischen Adressen (im Regelfall bei 0 oder einer anderen festen Adresse beginnend). Nun finde eine Verschiebung im Original-Nest statt, die das gesamte ausgeschnittene Teil-Nest mit umfasse, der gesamte Ausschnitt also mitverschoben werde. Transparenz bedeutet in diesem Beispiel, dass sich an den Adressen des ausgeschnittenen Teil-Nestes nichts ändert. Benutzer des Teil-Nestes merken gar nichts davon, dass ihr logischer Gastgeber-Adressraum „hinter ihrem Rücken“ verschoben wurde.

3.2. Zugriffs-Modelle

Auf einer Nest-Instanz lassen sich *Operationen* ausführen, darunter *elementare Grundoperationen*. *Operations-Aufrufe* dürfen grundsätzlich parallel durch mehrere *Aufrufer-Instanzen* (wer auch immer das sein mag) bzw. *asynchron* an eine *Bearbeiter-Instanz* (i.d.R. die Implementierung der Nest-Operationen) erfolgen. Die Bearbeiter-Instanz bewirkt für alle Operations-Aufrufe jeweils zugehörige *Operations-Ausführungen,* deren ausgelösten *Effekte* die Ausführung anderer Operationen beeinflussen können. Die Beeinflussung wird durch zwei grundlegende Modelle

beschrieben:

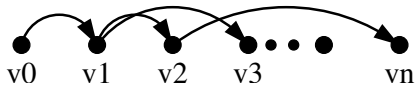
`linearize_global` Durch Operations-Ausführung entsteht eine sequentielle Folge von *Nest-Zuständen* auf derselben Nest-Instanz. Es ist Aufgabe einer Implementierung von Nest-Operationen, für die Einhaltung dieser so genannten *globalen Operations-Linearisierungs-Eigenschaft*[3] (zur Grundidee vgl. [HW90][4]) zu sorgen. Verschiedene Nest-Instanzen sind grundsätzlich voneinander unabhängig, soweit nicht durch Bausteine Abhängigkeiten eingeführt werden.

`linearize_local` Für alle Operations-Aufrufe, die sich auf eine gegenseitig überschneidende Teilmenge der logischen Adressen beziehen, wird eine sequentielle Folge von Zuständen definiert bzw. hergestellt; Operations-Ausführungen auf verschiedenen logischen Adressbereichen stehen nicht notwendigerweise in einer Ordnungsrelation zueinander (auch *Kommutierbarkeit*[5] genannt). Damit wird i.A. nur eine Halbordnung zwischen den Operations-Ausführungen auf derselben Nest-Instanz garantiert (*lokale Linearisierung*[6]).

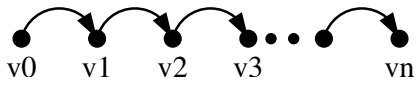
Globale und lokale Linearisierung unterscheiden sich demnach nur in der Granularität, mit der die Zustände bezeichnet werden; weitere Modelle werden in Kapitel 8 behandelt. Nur wenn lokal linearisierte Operationen mit unterschiedlich großen Adressbereichen auftreten, bei denen ein größerer Adressbereich mehrere kleinere überlappt, entstehen zwangsläufig halbordnungsartige Querbezüge zwischen ansonsten unabhängigen Strängen von „Objekt“-Zuständen.

Eine weitere Verfeinerung der Semantik von Nestern mittels Zusätzen wie z.B. Bausteine zur Implementierung verschiedener Arten von Serialisierbarkeit [Vid87] ist möglich und gehört zu den Strategien, die mittels Bausteinen implementiert werden können. Das Modell der lokalen Linearisierung ist ziemlich allgemein gehalten, um verschiedene Konsistenz- oder Korrektheitsbegriffe als Spezialisierung ableiten zu können.

Die Forderung nach (irgend)einer sequentiellen Folge von Nest- oder Adressbereichs-Zuständen (unabhängig von der Granularität) lässt sich durch eine *operationale Semantik* (siehe [Pag81]) beschreiben. Diese legt nur fest, welche Operation welche Effekte auf einem Nest bzw. Adressbereich bewirkt, d.h. wie aus einem Zustand v_i ein Zustand v_j wird. Die Zustände v_i und v_j werden auch als *Versionen* (vgl. [VGH93]) bezeichnet. Eine Operations-Ausführung impliziert zwischen den Versionen v_i und v_j die Relation der *direkten funktionalen Abhängigkeit* (vgl. [VGH93]). Im allgemeinen gilt $j > i$, d.h. die direkte funktionale Abhängigkeit ist eine Halbordnung:



Die (oft wünschenswerte) Eigenschaft der *Determinanz*[7] ist dann gegeben, wenn stets $j = i+1$ gilt, d.h. die funktionale Abhängigkeit stellt eine Totalordnung dar:



Die Folge der (globalen oder lokalen) Operations-Ausführungen hängt per Konvention mit der Folge der Nest-Zustände in einer 1:1-Beziehung zusammen (d.h. jede Operation bewirkt genau eine Version des Nest-Zustands, die ggf. gleich zu einer früheren Version ausfallen kann), auch wenn die Determinanz nicht gegeben sein sollte.

Ein Zustand v_i einer Nest-Instanz besteht aus einer *Adressabbildungs-Funktion* a_i, die logische Adressen auf physische Adressen abbildet, und einer *Datenabbildungs-Funktion* d_i, die physische Adressen auf Bytes abbildet. Wir schreiben $v_i = (a_i, d_i)$ und bringen damit zum Ausdruck, dass die Komposition der beiden Funktionen a_i und d_i den Zustand v_i vollständig charakterisiert. Die Operationen lassen sich in folgende *Operations-Arten* einteilen:

invariant Eine invariante Operation ändert nichts am Zustand des Nestes, d.h. es gilt $v_j = v_i$.

datenmodifizierend Es gilt $a_j = a_i$ und $d_j \neq d_i$. Eine datenmodifizierende Operation ersetzt den Inhalt physischer Datenblöcke durch einen

[3] Zum besseren Verständnis kann man hilfsweise den Begriff der „Folge von Operations-Aufrufen“ einführen. Dann ist die Folge der Operations-Ausführungen eine Permutation der Folge der Operations-Aufrufe, die folgende weitere Bedingung erfüllt: wenn man die beiden Folgen zu einer gemeinsamen Folge vereinigt, dann liegt jeder ursprüngliche Operations-Aufruf vor seiner zugehörigen Operations-Ausführung. Dieses Modell ist jedoch nur als Hilfskonstrukt zum besseren Verständnis zu verwenden. Realiter braucht man den Begriff der Folge von Operations-Aufrufen nicht einzuführen, sondern es genügt, von Operations-Aufrufen schlechthin zu sprechen, ohne dass diese in eine Folge gebracht werden müssen. Der Grund hierfür ist nicht, weil das Herstellen dieser Folge insbesondere in verteilten Systemen gewisse bekannte Schwierigkeiten macht (vgl. [AW94]), sondern weil man es zur Beschreibung der Semantik eines rein asynchronen Modells nicht benötigt.

[4] Im Unterschied zum Konsistenzmodell der Linearisierbarkeit wird hier nicht eine spezielle Speicher-Semantik unterstellt, denen die Lese- und Schreiboperationen unbedingt gehorchen müssen. Weiter unten werden beispielsweise Lese-Operationen behandelt, die ausdrücklich die Auslieferung veralteter Versionen zulassen und daher im konventionellen Modell der Linearisierbarkeit nicht enthalten sind.

[5] Im allgemeinen ist die Disjunktheit der betroffenen Adressbereiche für die Kommutation von Operations-Ausführungen hinreichend, aber nicht notwendig. So genannte „semantische Modelle“ versuchen, die Auswirkungen normalerweise nicht kommutierbarer Operationen nachträglich so zu interpretieren oder zu korrigieren, dass im Sinne irgendwelcher semantischer Anforderungen oder Bedingungen ein „korrektes“ Ergebnis herauskommt. Ein derartiger Korrektheitsbegriff muß nicht notwendigerweise in allen möglichen Ausführungsfolgen den gleichen Endzustand erreichen. In Abgrenzung zum Kommutierbarkeits-Begriff möchte ich dies *Quasi-Kommutierbarkeit* nennen.

[6] Der Begriff *lokal* bezieht sich hierbei auf eine Eigenschaft des Nestes, nämlich seinen Adressbereich, und nicht auf eine bestimmte physische Verteilung auf Rechnerknoten oder dergleichen.

[7] Dieser Begriff soll zur Unterscheidung von den Begriffen Determiniertheit und Determinismus dienen. Ich verwende den Begriff Nichtdeterminismus nicht, da sein Gegenteil, der Determinismus, in einem asynchronen Modell nicht auftritt (bzw. nur als Spezialfall auftritt). Determiniertheit ist auch nicht gegeben, weil i.a. die Reihenfolge der Ausführung mehrerer paralleler Operationen nicht fest liegt. Determinanz ist ein Begriff, der eine Zwischenstufe zwischen Determinismus und Nichtdeterminismus ausdrücken soll: es entsteht zwar eine „deterministische“ Folge der voneinander funktional abhängigen Zustände, doch diese Entstehung geschieht auf nichtdeterministische Weise aus Operations-Aufrufen, für die keine Ordnung vorausgesetzt wird.

anderen Inhalt in einem zusammen hängenden Bereich des logischen Adressraums, der *Modifikations-Bereich* genannt wird; ausserhalb des Modifikations-Bereiches ist der Effekt der Operation invariant. Datenmodifizierende Operationen können sowohl auf statischen als auch auf dynamischen Nestern ausgeführt werden.

adressmodifizierend Es gilt $d_j = d_i$ und $a_j \neq a_i$. Eine adressmodifizierende Operation bewirkt, dass neue Inhalte (Werte) physischer Datenblöcke an bisher undefinierten Stellen erscheinen (`clear`) oder unter neuen logischen Adressen erreichbar sind (`move`) oder auf undefiniert gesetzt werden (`delete`). Adressmodifizierende Operationen setzen ein dynamisches Nest voraus.

Was mit den Zuständen geschieht, bzw. wofür sie benutzt werden, ist eine eigenständige Frage, die sich auf verschiedene Weisen beantworten lässt. Ich propagiere folgende Modelle:

`singleuser` Es ist nur eine einzige Aufrufer-Instanz vorhanden, die synchrone Operations-Aufrufe auf der Nest-Instanz ausführen kann. Die Nest-Instanz *sichert* dem Aufrufer die Determinanz *zu*. Der Aufrufer hat nur Zugriff auf die jeweils *letzte* Version v_n der Zustände der Nest-Instanz, bzw. Änderungen erfolgen nur dort (Single-Copy-Eigenschaft).

`multiuser` Es dürfen mehrere Aufrufer-Instanzen vorhanden sein. Diesen wird die Eigenschaft der Determinanz zugesichert. Im Vergleich zum Singleuser-Modell erhält eine einzelne Aufrufer-Instanz trotz Beauftragung mit einem einzigen Operations-Aufruf nicht mehr die Zusicherung, in welcher Reihenfolge die Ausführung gegenüber den Aufträgen anderer Instanzen oder eigenen asychronen Aufrufen geschieht.

`multiversion` Dieses Modell wird in Kapitel 8 behandelt und zerfällt in mehrere Untermodelle. Es dürfen mehrere Aufrufer-Instanzen vorhanden sein. Diese haben im allgemeinsten Untermodell *potentiellen Zugriff* auf alle Adress-Versionen a_i und alle Daten-Versionen d_j einer Nest-Instanz, womit ein Zugriff auf beliebige *Pseudo-Zustände* $v'_{ij} = (a_i, d_j)$ möglich ist, bei denen $i = j$ nicht unbedingt gelten muss (was jedoch in einigen Untermodellen gefordert werden kann). Damit wird nur noch die Zusicherung gegeben, dass die Operations-Ausführung auf *irgend einer gültigen* (also nicht auf einer undefinierten) Version der Adress- und Datenabbildung statt findet, die jedoch auch älter sein kann.

Die Modelle `singleuser` und `multiuser` erfüllen per Konstruktion die Eigenschaft der Determinanz. Im Folgenden werden wir uns auf diese beiden Modelle beschränken und `multiversion` in Kapitel 8 gesondert betrachten.

Die Eigenschaft der *Korrektheit* wird durch die hier beschriebenen Nest-Modelle nicht betrachtet[8]; die damit verbundenen Probleme sind auf anderer Ebene zu lösen. Beispielsweise muss nicht unbedingt die Semantik von Halbleiter- oder Plattenspeichern implementiert werden, bei denen eine Schreiboperation die vorherige Version ersetzt und nicht mehr zugänglich macht; virtuelle Dateninhalte im Stile des `/proc`-Dateisystems sind nicht ausgeschlossen. Dies ermöglicht konkreten Baustein-Implementierungen, verschiedene Modelle von Korrektheit auf Basis der allgemeineren Abstraktion Nest zu erfüllen. Jede Aufrufer-Instanz hat die *Verpflichtung*, selbst[9] für die Erfüllung weiterer von ihr selbst gegebenen Zusicherungen, insbesondere die Korrektheit (im von ihr selbst definierten Sinne) zu sorgen. Die Nest-Abstraktion stellt dafür eine universelle abstrakte Schnittstelle zur Verfügung.

Eine Nest-Implementierung braucht nicht unbedingt alle drei Modelle zu unterstützen. Welche Modelle im Einzelfall unterstützt werden, gehört zu den Kompetenzen (vgl. Abschnitt 2.5) der Nest-Implementierung. Wie bereits gesagt, müssen die Kompetenzen der Nest-Implementierung zum Verhalten einer Aufrufer-Instanz kompatibel sein.

Die Tabelle 3.1 zeigt die möglichen Kompatibilitäten. Die Tripel bedeuten hierbei, wie viele verschiedene Aufrufer-Instanzen mit Verhalten `singleuser`, `multiuser` und `multiversion` jeweils an einer Konfiguration mit einer Nest-Implementierung der betreffenden Kompetenz teilnehmen dürfen. Während sich `singleuser` mit keinem anderen Aufrufer-Verhalten verträgt, sind `multi*` beliebig untereinander mischbar[10], sofern die Nest-Implementierung von ihren Kompetenzen her mitspielen kann.

3.3. Elementaroperationen auf statischen und dynamischen Nestern

Der hier vorgestellte Satz von Elementaroperationen auf Nestern ist als Beispiel zu verstehen, wie man die Nest-Schnittstelle gestalten kann. Die Beschreibung ist informell und soll die Kernidee erklären; es handelt sich nicht um eine vollständige Spezifikation aller Details.

Der Entwurf von IO-Schnittstellen wird in der Praxis durch diverse Umstände verkompliziert, die nach meinem Dafürhalten für einen Großteil der Implementierungs-Komplexität realer Betriebssysteme mit verantwortlich sind. Durch Berücksichtigung dieser Umstände bereits in

[8] Die einzelnen Elementaroperationen müssen natürlich trotzdem die „Korrektheit“ oder besser die „Validität“ auf ihrer jeweiligen Abstraktionsebene bzw. Zugriffs-Granularität erfüllen.

[9] Die Abstraktion der Nester gibt jedenfalls keine Zusicherungen über „Korrektheit“, sondern höchstens über die (Nicht-)Kommutierbarkeit von Elementaroperationen. Den Aufrufern wird überlassen, welche Semantik sie implementieren wollen; dabei dürfen und sollen sie sich jedoch auf die Zusicherungen der Bearbeiter-Implementierung abstützen.

[10] In Kapitel 8 werden zwar die Schnittstellen der Elementaroperationen gegenüber hier erweitert, so dass vordergründig betrachtet die Schnittstellen zwischen `multiuser` und `multiversion` nicht zusammen passen. Dieses Problem lässt sich jedoch z.B. sehr leicht durch triviale Anpassungs-Bausteine lösen, da `multiuser` von der abstrakten Mächtigkeit her ein Untermodell von `multiversion` ist.

		Verhalten		
		singleuser	multiuser	multiversion
Kompetenz	singleuser	$(1,0,0)$	–	–
	multiuser	$(1,0,0)$	$(0,n,0)$	–
	multiversion	$(1,0,0)$	$(0,n,m)$	$(0,n,m)$

Tabelle 3.1.: Kompatibilitäten von Kompetenzen und Verhalten

den Schnittstellen und im *gesamten System über alle Ebenen hinweg*[11], sowie durch die Unterstützung so genannter „fortgeschrittener IO-Konzepte“ wie asychroner IO und Nonblocking-IO mittels universeller Generizität lässt sich diese Komplexität reduzieren[12]. Daher enthält bereits die Schnittstelle für statische Nester alle im System vorkommenden IO-Konzepte.

Ich habe versucht, im folgenden Entwurf auch die Anforderungen von Datenbanken so weit zu berücksichtigen, dass hoffentlich die Grundfunktionalität eines Datenbank-Systems weitgehend abgedeckt werden kann[13].

Sperrmechanismen dienen dazu, die *möglichen Ausführungsfolgen* von Operationen einzuschränken. Als Sperrmechanismen sind bei statischen und dynamischen Nestern die elementaren Grundoperationen `lock` und `unlock` vorgesehen. Neben diesen werden keine weiteren Sperrmechanismen im gesamten Betriebssystem benötigt[14].

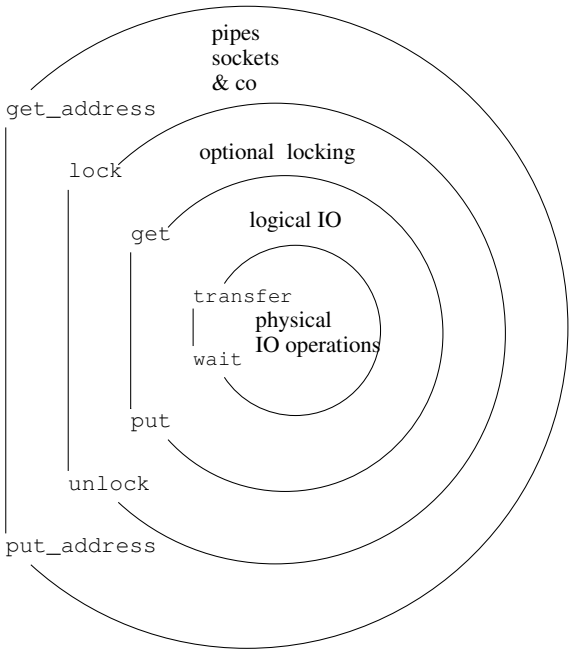
Ein statisches Nest lässt sich als Einschränkung eines dynamischen Nestes verstehen, bei dem die Verschiebe-Operation fehlt und keine Löcher im Definitionsbereich vorkommen, sowie in den meisten Fällen (etwa bei Festplatten-Partitionen) keine dynamische Änderung der Größe erfolgt. Allerdings wird diese abstrakte Sicht nicht vollkommen von der Realität aktueller Peripheriegeräte geteilt: die Granularität von Datentransfers ist bei aktueller Peripherie auf Blöcke fester Länge, etwa 512 Byte bei Festplatten oder 2048 Byte bei CD-Laufwerken, begrenzt. Darüber hinaus dürfen Datentransfers nur an solchen Adressen statt finden, die durch die Granularität ohne Rest teilbar sind. Diese Einschränkungen werden durch ein Attribut `transfer_size` modelliert, das jeder Nest-Instanz zugeordnet ist. Jeder Verwender eines Nestes sollte es abfragen und bei seinen IO-Aufträgen berücksichtigen.

Bei statischen und dynamischen Nestern sorgen die Elementaroperation dafür, dass die logischen Adressen (die relativ zum jeweiligen Nest gelten) auf *physische Adressen* von transienten Datenblöcken übersetzt werden. Die Zuständigkeit für die Verwaltung der physischen Adressen kann unterschiedlich geregelt sein. Hierfür werden zwei Betriebsarten unterstützt, die prinzipiell auch gemischt nutzbar sind: *logischer IO* und *physischer IO*.

Beim logischen IO geschieht die Verwaltung (Platzreservierung und -Freigabe) der transienten Datenblöcke im physischen Hauptspeicher durch die Operation `get`, während beim physischen IO dafür der Aufrufer selbst verantwortlich ist. Physischer IO sollte idealerweise nur auf den unteren Schichten von Gerätetreibern benutzt werden, das restliche System benutzt normalerweise logischen IO.

Die Elementaroperationen wickeln zusammen nicht nur sämtlichen asynchronen IO ab, sondern erlauben auch den transparenten Einsatz eines Buffer-Cache ohne Änderung der Schnittstelle. Wie wir später sehen werden, führen sie auch die Speicherverwaltungs-Funktionalität des Systems durch, einschließlich des Paging von virtuellen Benutzer-Speicherseiten. Aus der folgenden Beschreibung des asynchronen IO-Modelles ergibt sich die Realisierung von synchronem IO und Nonblocking-IO auf triviale Weise, so dass ich mir deren Ausführung sparen werde.

Das folgende Bild enthält eine Übersicht über die Elementarooperationen auf statischen Nestern:



Wie zu sehen ist, werden die Elementaroperationen immer

[11] Neuere Buffer-Caches wie beispielsweise [PDZ99] vereinheitlichen zwar das Caching in einem System auf ähnliche Weise wie hier, führen jedoch ihre eigene Schnittstelle ein. Im Gegensatz dazu verlangt das Prinzip der universellen Generizität (Abschnitt 2.1.4), dass die Nest-Schnittstelle nicht verändert wird, wenn irgendwo Caching eingeführt oder nicht mehr benutzt wird.

[12] In der historischen Entwicklung erfolgreicher Betriebssystem-Kerne wie der UNIX-Familie wurden immer wieder so genannte „Balkone“ an die vorhandenen Konzepte angebaut, um mit geänderten und erweiterten Anforderungen Schritt halten zu können. Balkone widersprechen der Forderung nach möglichst geringer Redundanz eines Entwurfs. Die hier vorgeschlagenen Schnittstellen-Konzepte und Abstraktionen versuchen das Ideal von universeller Generizität anzunähern, um auch zukünftige unvorhersehbare Anforderungen mit etwas Glück ohne Balkone nachrüsten zu können.

[13] Es sollte eventuell noch eine weitere Elementaroperation `search` zur Modellierung universeller Suchoperationen eingeführt werden. Dies wird im Rahmen dieser Arbeit über Betriebssysteme nicht weiter verfolgt. Falls fortgeschrittene Konzepte der Implementierung von Datenbank-Systemen [HR01] nicht ausreichend berücksichtigt sein sollten, dann hoffe ich, dass sich diese Mechanismen ebenfalls in die vorgestellte Architektur mit geringem Aufwand integrieren lassen.

[14] Dies gilt auch für die bekannte Multiprozessor-Problematik (SMP) und die bisher durch Unterbrechungs-Sperren gelösten Synchronisationsprobleme, die durch geeignet implementierte Elementaroperationen `lock` und `unlock` lösbar sind, sofern man Unterbrechungen als vollwertige Kontrollflüsse implementiert (vgl. [KE95]).

paarweise mit einer entgegengesetzten oder aufhebenden (antagonistischen) Operation kombiniert. Verschiedene Arten von IO oder sonstige Zugriffs-Arten auf Nester ergeben sich durch ein Schichtenmodell, das hier kurz von innen nach außen skizziert werden soll:

Im einfachsten Fall wird physischer IO durch die Operation `transfer` betrieben. Die Funktionalität synchronenen IOs entsteht durch anschließendes Warten auf die Beendigung mittels `wait`. Bei asynchronem IO kann auch auf den Aufruf von `wait` verzichtet werden. Bei logischem IO wird die Abbildung von logischen auf physische Adressen von der Operation `get` übernommen. Durch den antagonistischen Aufruf von `put` wird die Ressourcen-Verwaltung auf ähnliche Weise wie bei einem konventionellen Buffer-Cache gelöst (z.B. interne Verwendung von Referenzzählern oder andere Methoden). Damit ein Ausgang eines Bausteins mit mehreren Eingängen anderer Bausteine parallel verdrahtet werden darf (`multiuser`-Kompetenz), wird der *wechselseitige Ausschluss* (didaktisch hervorragende Erklärung in [Lag78]; Problembeschreibung und Lösung bereits in [Bri64], siehe außerdem [Dij65]) bei parallelem Zugriff durch die Operationen `lock` und `unlock` bereit gestellt. Zur Nachbildung von Pipes u.ä. sowie zur Speicherverwaltung existieren weitere Elementaroperationen `get_address` und `put_address`, mit denen das Problem der atomaren Reservierung von Adressbereichen insbesondere im Falle mehrerer paralleler Zugreifer lösbar ist.

3.3.1. `transfer`

Diese Elementaroperation sorgt dafür, dass eine *Aktualisierungs-Operation* in eine der beiden möglichen Richtungen zwischen logischer Adresse im Nest und physischem Datenblock beauftragt wird (ohne auf die Ausführung dieser Operation zu warten). Eine derartige Aktualisierungs-Operation wird synonym *IO-Operation* oder *IO-Auftrag* genannt; anstelle des Begriffs IO-Operations-Ausführung wird synonym auch der Begriff *Beendigung* des IO-Auftrags verwendet. Verschiedene IO-Aufträge sind prinzipiell voneinander unabhängig. Die Schnittstelle sieht folgendermaßen aus:

```
transfer(nest,mandate,log_addr,len,
            phys_addr,direction,io_prio,
                        depend) → success
```

Der Parameter `direction` kann einen der folgenden Werte annehmen:

`read` Die Aktualisierung erfolgt von der logischen Adresse im Nest in Richtung zum physischen Datenblock. Bei erfolgreicher Beendigung des IO-Auftrages wird zugesichert, dass sich die neueste Version an der physischen Adresse befunden hat.

`write` Die Aktualisierung erfolgt in Richtung von der physischen Datenblock-Adresse an die logische Adresse auf dem Hintergrund-Medium. Mit Durchführung des IO-Auftrages entsteht eine geänderte neueste Version des Nest-Zustandes.

`stop` Ein eventuell bereits wartender IO-Auftrag wird wieder aus der Warteschlange der *wartenden* Aufträge entfernt, sofern dies möglich ist; bei bereits in Ausführung befindlichen IO-Aufträgen ist dies im Regelfall nicht mehr möglich. Die Folge ist, dass die von den anderen `direction`-Arten gegebenen Zusicherungen nicht mehr eingehalten werden müssen.

Der IO-Auftrag erfolgt mit der Prioritätsangabe `io_prio`, die in Abschnitt 2.8.3 ausführlich erklärt wird. Im Regelfall werden Aufträge mit einer höheren IO-Priorität vor solchen mit geringerer Priorität ausgeführt; Ausnahmen sind möglich, wenn dadurch der Gesamtdurchsatz des IO-Systems gesteigert wird. Der Parameter `depend` erlaubt die Spezifikation einer Halbordnung zwischen IO-Aufträgen, was insbesondere zur Implementierung absturzsicherer atomarer Operationen bzw. Transaktionen benötigt wird. Es wird eine Liste derjenigen logischen Adressen angegeben, deren zugehörige IO-Aufträge (sofern sie existieren) auf jeden Fall beendet sein müssen, bevor der aktuelle IO-Auftrag ausgeführt (physikalisch gestartet) werden darf.

Falls `transfer` auf einen bereits vorhandenen IO-Auftrag trifft, sollte ein doppelter Eintrag von gleich lautenden Aufträgen vermieden werden. Dadurch wird eine Überflutung des IO-Systems durch unnötige IO-Aufträge selbst bei massenhaftem Einsatz von Background-IO (vgl. Abschnitt 2.8.3) verhindert. Ggf. wird die IO-Priorität eines bereits wartenden Auftrages auf das neue Niveau erhöht, so dass in Zukunft mit einer zügigeren Bearbeitung zu rechnen sein wird.

Im boolschen Ergebnis-Parameter `success` wird zurück gemeldet, ob ein erfolgreiches Erzeugen des IO-Auftrages möglich war.

3.3.2. `wait`

Diese Elementaroperation implementiert mittelfristiges Warten. Die Schnittstelle sieht folgendermaßen aus:

```
wait(nest,mandate,log_addr,len,
                                  prio) →
                                (success)
```

Es wird solange gewartet, bis alle IO-Aufträge, die zum Zeitpunkt des Aufrufs im angegebenen beliebig großen Adressbereich gelegen haben und höchstens die IO-Priorität `prio` hatten[15], abgearbeitet worden sind. Falls kein solcher Auftrag existiert, wird nicht gewartet.

3.3.3. `get`

Ein Buffer-Cache verwaltet eine *transiente interne Zuordnung* zwischen logischen und physischen Adressen. Diese Zuordnung ist *transient*, d.h. sie gilt längstens für den Zeitraum, in dem das Betriebssystem läuft, und kann wieder aufgelöst werden, sobald keine dynamischen Referenzen mehr auf den physischen Datenblock bestehen. Trotzdem

[15] Wenn man auch auf nachträglich durch parallele Aufrufer hinzugekommene Aufträge warten würde, wäre Verhungern möglich. Einfache und effiziente Implementierungen, die Verhungern nicht vermeiden, könnten u.U. in unkritischen Anwendungen dennoch zugelassen werden.

sollte die Zuordnung nach Möglichkeit für längere Zeit bestehen, als der Aufrufer unbedingt verlangt. Ohne die Möglichkeit zur Aufbewahrung einer zeitlich länger dauernden Zuordnung zwischen logischen und physischen Adressen wäre eine *Caching-Funktionalität* nicht oder nur schlecht erfüllbar.

Die Schnittstelle umfasst folgende Parameter:

```
get(nest,mandate,log_addr,len,mode)→
 (phys_addr,len,min_version,max_version)
```

Die übergebene Adresse `log_addr` muss zur erfolgreichen Durchführung durch `transfer_size` teilbar sein, die Länge muss ein Vielfaches davon darstellen. Bei erfolgreicher Durchführung wird die physische Adresse `phys_addr` zurückgeliefert. Die zurückgelieferte Länge darf kürzer als die angeforderte Länge sein (beispielsweise wenn das unterliegende IO-System keine Transfers mit der gewünschten Länge am Stück ausführen kann), sie muss aber durch `transfer_size` teilbar sein.

Die Bearbeiter-Instanz sucht auf atomare Weise in ihren internen Zuordnungs-Strukturen (z.B. schnelle Hash-Tabellen), ob bereits eine Zuordnung mit einem physischen Datenblock an der gewünschten logischen Adresse besteht. Falls ja, dann wird diese Adresse und die ggf. gekürzte Länge zurückgeliefert. Falls noch keine Zuordnung besteht, wird Platz für einen neuen zusammenhängenden Datenblock der Länge `len` allokiert, oder falls dies nicht möglich ist, ein möglichst großer Datenblock bis zur Länge `len` (in Vielfachen von `transfer_size`). Schließlich werden die Adresse und Länge in die Zuordnungs-Strukturen eingetragen und zurückgeliefert.

Die Ergebnis-Parameter `min_version` und `max_version` melden die Aktualität des gefundenen Datenblocks nach der folgenden Systematik:

`undefined` Der Inhalt des Datenblocks an der zurückgelieferten physischen Adresse ist undefiniert.

`newest` Die Nest-Implementierung sichert zu, dass die neueste Version des Nest-Zustandes im Datenblock ausgeliefert wurde, die zum Zeitpunkt der Operations-Ausführung gültig war. Sofern keine Sperrmechanismen die Ausführung weiterer Operationen verhindern, wird nicht garantiert, dass in der Zwischenzeit zwischen der Operations-Ausführung und der Inspektion durch die Aufrufer-Instanz keine weiteren Operations-Ausführungen stattgefunden haben, die den Nest-Zustand inzwischen verändert haben.

Zwischen diesen Werten gilt die Relation `undefined` $<$ `newest`. Weiterhin gilt `min_version` $\leq$ `max_version`. Der Wert von `min_version` bedeutet, dass sich die Nest-Implementierung dessen sicher ist, dass der Datenblock mindestens die angegebene Aktualität besitzt. Umgekehrt bedeutet `max_version`, dass die Nest-Implementierung sicher ist, dass die Version den angegebenen Aktualitätsgrad nicht überschreitet. Wenn die Nest-Implementierung beispielsweise überhaupt nichts über die Aktualität weiß, dann ist `min_version` = `undefined` und `max_version` = `newest`. Nur wenn `min_version` = `max_version` = `newest` gilt, ist der Datenblock aktuell. Aus `min_version` = `max_version` = `undefined` kann gefolgert werden, dass entweder kein IO-Auftrag (mehr) vorhanden ist, oder dass sein Effekt durch `transfer` im `stop`-Modus wieder aufgehoben wurde, oder dass die `transfer`-Operation technisch nicht erfolgreich war (Schreib- oder Lesefehler).

Physische Adressen von Datenblöcken stellen *Aliase* dar, die ab `multiuser`-Kompetenz von mehreren Aufrufern gemeinsam benutzt werden können. Es wird allerdings nicht garantiert, dass verschiedene Aufrufer-Instanzen beim `get` an der gleichen logischen Adresse dieselbe physische Adresse zurückgeliefert bekommen[16].

Der boolsche Parameter `mode` gibt an, ob der Aufrufer den Datenblock anschließend zu Schreibzwecken benutzen darf. Bei einigen Implementierungs-Paradigmen (vgl. Abschnitt 2.7) wird diese Information genutzt, um Verstöße gegen das Reglement zu erkennen und ggf. zu sanktionieren. Ab `multiuser`-Kompetenz sollte intern gemerkt werden, wieviele bzw. welche Aufrufer den `mode`-Parameter gesetzt hatten (siehe Prüfung der Zulässigkeit in Abschnitt 3.3.6).

Falls mehrere Aufrufer den `mode`-Parameter gesetzt haben, dürfen diese prinzipiell unabhängig voneinander eigene Änderungsoperationen auf dem physischen Datenblock ausführen, was zu Interferenzen führen kann[17], wenn sie keine weiteren Maßnahmen ergreifen (z.B. durch Setzen von Locks[18]).

Es sind mehrere verschiedene Implementierungs-Semantiken dieser Schnittstelle möglich:

Referenzzähler-Semantik Kennzeichen der Referenzzähler-Semantik ist, dass nur gemerkt wird, *wie viele* Benutzer gerade einen Datenblock benutzen, aber nicht *welche* Benutzer dies konkret sind. Die technische Durchführung kann nach folgendem Schema erfolgen:

Jedem zurückgelieferten physischen Datenblock wrd ab `multiuser`-Kompetenz ein *Referenzzähler* zugeordnet, der die Anzahl der auf ihn existierenden Referenzen verwaltet. Ein Aufruf von `get`, der einen bereits vorhandenen Datenblock auffindet, inkrementiert diesen Referenzzähler. Falls ein neuer Datenblock allokiert wurde, dann steht der Referenzzähler regelmäßig auf 1. Die logische Freigabe des physischen Datenblocks erfolgt durch die später beschriebene Operation `put`, die den Referenzzähler wieder dekrementiert und daher immer paarweise zu `get` aufgerufen

[16] Dies ist in Verteilten Systemen im allgemeinen unmöglich. Umgekehrt wird auch nicht garantiert, dass *kein* Alias entsteht.

[17] Interferenzen brauchen nicht generell zu entstehen; beispielsweise könnten externe Absprachen zwischen mehreren Schreibern bestehen, dass diese nur auf zueinander disjunkte Bereiche eines physischen Datenblocks ändernd zugreifen; ein typischer Anwendungsfall dafür ist z.B. ein Nachrichtenpuffer zur wechselseitigen Publikation irgendwelcher Status-Informationen, die keinen Einfluss auf die Korrektheit haben, sondern z.B. zur Lastbalancierung genutzt werden. Die hier vorgestellte Architektur möchte die Lösung von Interferenz-Problemen durch Baustein-Implementierungen *ermöglichen* und *unterstützen*, aber nicht erzwingen, insbesondere denjenigen Implementierern keine Hemmschuhe in den Weg legen, die „wissen was sie tun“ und die nicht für Performanz-Overhead durch höherwertige Zusicherungen bezahlen möchten. Im übrigen lassen sich höherwertige Zusicherungen jederzeit durch geeignete Bausteine nachrüsten.

[18] Interferenzen sind auch zwischen Nur-Lesern und einem Schreiber möglich. Die Nest-Implementierung übernimmt dafür keine Verantwortung. In Abschnitt 3.3.5 wird beschrieben, wie Baustein-Implementierungen durch Locks für die Vermeidung von Interferenzen sorgen können.

werden muss (gleiche Anzahl von Aufrufen mit genau denselben Adressen und Längen `log_addr` und `len`).

Eigentümer-Verwaltungs-Semantik Hier wird gemerkt, unter welchem Mandat (siehe Abschnitt 5.1) welche Puffer gerade in Verwendung sind. Im Unterschied zu Referenzzählern darf ein Datenblock auch mehrmals nichtpaarweise zu `put` unter dem gleichen Mandat angefordert werden. Die Freigabe durch `put` kann in einer anderen Granularität als durch `get` erfolgen; insbesondere ist die schlagartige Freigabe aller unter dem gleichen Mandat verwendeten Puffer durch eine einzige `put`-Operation möglich.

Diese Semantik besitzt zwar den Nachteil eines höheren Verwaltungsaufwandes im Vergleich zu Referenzzählern, dafür bringt sie jedoch hervorstechende Vorteile in größeren Systemen: falls irgend ein Baustein ein Fehlverhalten zeigt, kann anders als bei Referenzzählern gezielt reagiert werden, indem z.B. gezielt Ressourcen entzogen und ein Cleanup durchgeführt wird.

3.3.4. `put`

Diese Operation dient als Antagonist zu `get`. Bei Referenzzähler-Semantik wird dieser auf atomare Weise dekrementiert; bei Eigentümer-Verwaltungs-Semantik wird der Besitz an dem angegebenen Datenbereich für das betreffende Mandat aufgegeben. Die Schnittstelle lautet:

```
put(nest,mandate,log_addr,len,mode)
```

Bei Referenzzähler-Semantik: falls der Referenzzähler durch diese Operation zu 0 wird, dann *darf* die ursprünglich durch `get` hergestellte Zuordnung zwischen logischen und physischen Adressen wieder auf atomare Weise aufgelöst werden und der Speicherplatz für den Datenblock freigegeben werden. Dies *soll* jedoch möglichst nur bei Speichermangel geschehen. Nach dem letzten `put` haben Zugriffe auf die zugehörige physische Adresse keine Bedeutung mehr für die Aufrufer-Instanz.

Falls irgend eine Aufrufer-Instanz Zugriffe auf eine physische Adresse macht, die keine Bedeutung (mehr) für sie hat, dann gilt dies als *Fehler*. Falls Fehler nicht abgefangen und verhindert werden konnten (vgl. Abschnitt 2.7), dann kann als Folge daraus ein Teilsystem in einen undefinierten Zustand geraten. Jegliche Instanzen haben daher die Pflicht, Fehler zu vermeiden.

Durch den Parameter `mode` wird der interne Referenzzähler für die Anzahl der Schreiber verwaltet; Aufrufer haben die Pflicht, diesen Parameter paarweise übereinstimmend zum `mode`-Parameter von `get` zu verwenden (Zuwiderhandlungen gelten als Fehler).

Bei Eigentümer-Verwaltungs-Semanik dürfen auch nicht-paarweise Aufrufe von `put` stattfinden, die nicht unbedingt auf logische Adressen verweisen müssen, die auch tatsächlich unter dem betreffenden Mandat angefordert waren. Es wird jedoch zugesichert, dass alle vom logischen Adressbereich betroffenen physischen Datenadressen keine Bedeutung mehr haben.

3.3.5. `lock` und `unlock`

Sperren bzw. *Locks* (Beschreibung von Read-Write-Locks bereits in [Bri64]) bewirken, dass die möglichen Ausführungs-Folgen von Operationen eingeschränkt werden. Die abstrake Schnittstelle ermöglicht mindestens zwei verschiedene Semantiken: *advisory Locks* und die hier vorgeschlagenen *hard Locks*.

- advisory Locks (vgl. [Ste97]) sperren nur gegenüber anderen anderen advirsory Locks; wenn ein Aufrufer keine Sperren setzt, dann findet keine Synchronisation statt. Jeder Aufrufer ist also voll für das korrekte Setzen der Sperren verantwortlich; eine einzige Aufrufer-Instanz mit Fehlverhalten kann jede andere korrekte Aufrufer-Instanz durch ihr Fehlverhalten stören.

- hard Locks[19]: diese hier vorgeschlagenen Sperren wirken nicht nur gegenüber anderen Sperren, sondern auch gegenüber einigen anderen Elementaroperationen. Ist eine Sperre erst einmal erfolgreich gesetzt worden, dann *garantiert* die jeweils beauftragte Ausführungs-Instanz (Nest-Implementierung) ihre Einhaltung gegenüber der Aufrufer-Instanz, und zwar auch gegenüber anderen Aufrufern von Operationen, die die Korrektheit verletzen *könnten*. Dies ist nicht gleichbedeutend mit *mandatory Locking* (vgl. [Ste97]): Aufrufer-Instanzen werden nicht gezwungen, *hard Locks* zu verwenden. Sie müssen also für die Realisierung irgendwelcher Korrektheits-Begriffe grundsätzlich selbst Sorge tragen, indem sie die möglichen Ausführungsfolgen von Operationen mit den zulässigen Ausführungsfolgen ihres jeweiligen Korrektheits-Begriffes *verträglich* machen. Dies belastet zwar die Baustein-Implementierungen mit der Pflicht, für diese Verträglichkeit selbst zu sorgen, ermöglicht dafür jedoch wesentlich flexiblere Korrektheits-Modelle.

Alle hier vorgeschlagenen Sperren sollen neben der Grundfunktionalität eines Betriebssystems auch wichtige Bereiche der Funktionalität von Datenbanken und Transaktionen (siehe z.B. [Dat95, EN95]) abdecken. Das hier vorgestellte Modell ist als *Vorschlag* zu verstehen, der an manchen Stellen verbessert werden kann (beispielsweise durch Einführung intentionaler Locking-Arten [EN95]).

Die Elementaroperationen `lock` und `unlock` sperren bzw entsperren *Bereiche des logischen Adressraums* gegenüber anderen Aufrufer-Instanzen der `lock`-Operation bzw. gegenüber anderen Mandats-Inhabern und ggf. auch gegenüber `get`; gehaltene Sperren gelten grundsätzlich als der aufrufenden Instanz bzw den in Abschnitt 2.7 beschriebenen Mandaten zugeordnet. Die gesperrten Bereiche müssen nicht notwendigerweise im Definitionsbereich

[19] Bei Betriebssystem-Schnittstellen wird zwischen den Locking-Arten *mandatory* und *advisory* (vgl. [Ste97]) unterschieden. Die hier vorgeschlagenen hard Locks gehören zu keiner dieser beiden Arten. Es wird zwar ähnlich wie beim advisory locking niemand gezwungen, Locks zu verwenden; falls jedoch eine Aufrufer-Instanz einen Lock erhält, dann wird seine Einhaltung stets zugesichert, und zwar auch dann, wenn sich andere Instanzen „nicht an die Regeln halten“. Im Gegensatz zum advisory locking wird eine Umgehung des Locks durch nichtkonformes Verhalten anderer Instanzen verhindert. Hard Locks stellen also eine Zwischenstufe zwischen advisory und mandatory Locks dar.

von dynamischen Nestern aktuell vorkommen („Phantom-Locks“); es findet also keine Synchronisation auf Daten-Objekten statt, sondern eine auf *logischen Adressbereichen*. `lock`-Operationen, die auf gegenseitig nicht überlappenden Adressbereichen ausgeführt werden, sind voneinander unabhängig und kommutieren miteinander. Locks sind nicht rekursiv, sondern *akkumulierend* innerhalb derselben Lock-Art. Es schadet nicht, wenn bereits erhaltene Lock-Bereiche unter dem gleichen Mandat nochmals angefordert werden, bewirkt aber auch nichts. Die Rückgabe mittels `unlock` braucht nicht den gleichen Adressbereich zu betreffen als der vorherige `lock`. Damit ist es möglich, einen ursprünglich am Stück liegenden gesperrten Bereich in kleinere Teilstücke mit dazwischen liegenden freigegebenen Bereichen umzuwandeln. Umgekehrt darf ohne Schaden ein größerer Bereich mittels `unlock` freigegeben werden als vorher mittels `lock` gesperrt wurde; falls man z.B. `unlock` mit dem gesamten Adressraum als Parameter aufruft, dann werden sämtliche von der aufrufenden Instanz bzw dem Mandat gehaltenen Locks atomar auf einen Schlag freigegeben (*strikter* Zweiphasen-Commit). Die Parameter lauten:

```
lock(nest,mandate,log_address,len,
     try_address,try_len,kind,action) →
           (locked_address,locked_len)
unlock(nest,mandate,log_address,len)
```

Der *Sperr-Adressbereich* wird durch die Parameter `log_address` und `len` vorgegeben. Die Parameter `try_address` und `try_len` beschreiben einen *Versuchs-Adressbereich*, der einen Oberbereich des Sperr-Adressbereichs darstellen muss. Falls der Versuchs-Adressbereich echt größer als der Sperrbereich ist, dann braucht er von der Nest-Implementierung nur soweit im *Ergebnis-Sperrbereich* `locked_address` und `locked_len` berücksichtigt zu werden, wie dies ohne *zusätzliches* Warten möglich ist. Nähere Erklärungen für das damit mögliche *spekulative Locking* finden sich in Abschnitt 5.3. Der Ergebnis-Sperrbereich umfasst entweder mindestens den Sperr-Adressbereich (d.h. er liegt zwischen dem Sperr- und dem Versuchs-Bereich), oder er hat die Länge 0. Im letzteren Fall war ein Setzen des Locks nicht möglich (s.u.). Durch den Parameter `kind` werden neben den bekannten Arten *Read-Lock* und *Write-Lock*[20] noch weiter verfeinerte Lock-Arten unterschieden. Er hat einen der folgenden Werte:

`read` Der Ergebnis-Sperrbereich ist anschließend vor *Überschreiben* der d_i-Komponente durch andere Aufrufer-Instanzen von `lock` und von `get` jeweils im `write`-Modus *geschützt*[21].

`write` Der Ergebnis-Sperrbereich ist anschließend zum Überschreiben der d_i-Komponente für den Aufrufer *exklusiv reserviert*.

`upgrade` Wie `write`, jedoch wird ein bereits gesetzter Read-Lock zu einem Write-Lock *atomar aufgewertet* (sofern es möglich ist); dieser Modus wird bei einigen Transaktions-Modellen (insbesondere pessimistische Modelle) benötigt und sollte ausserhalb dieser Anwendung nur mit Vorsicht eingesetzt werden[22].

`fix` Der Ergebnis-Sperrbereich ist anschließend gegen Änderungen der a_i-Komponente durch andere Aufrufer-Instanzen geschützt.

`reorg` Der Ergebnis-Sperrbereich ist anschließend zum Ändern der a_i-Komponente für den Aufrufer exklusiv reserviert.

Die Arten `fix` und `reorg` entsprechen in ihrer Semantik den konventionellen Arten `read` und `write`, sie beziehen sich jedoch auf Modifikationen der Adressbereiche (z.B. durch `move`).

Der Parameter `action` beschreibt, wie die `lock`-Operation mit dem gewünschten Lock umgehen soll. Er hat einen der folgenden Werte:

`ask` Es wird lediglich nachgesehen, ob das Setzen eines entsprechendes Locks ohne zu blockieren möglich gewesen wäre. Der zurück gelieferte Ergebnis-Sperrbereich kann an Wettrennen mit anderen Aufrufern von Lock teilnehmen und ist entsprechend vorsichtig zu behandeln.

`try` Der Lock wird atomar gesetzt, wenn es sofort ohne zu blockieren möglich ist. Ansonsten wird die Länge 0 zurück geliefert.

`wait` Der Lock wird gesetzt, auch wenn dazu erst auf die Freigabe durch andere Aufrufer gewartet werden muss. Die Rücklieferung von `locked_len` = 0 ist trotzdem möglich, insbesondere wenn ein Deadlock vorliegt. Das Ergebnis muss daher in jedem Fall vom Aufrufer überprüft werden.

Die Einschränkung der möglichen Ausführungsfolgen hängt von den Kompetenzen der Nest-Implementierung ab. Bei `singleuser` brauchen Locks nicht implementiert zu werden; `multiversion` wird in Kapitel 8 behandelt. Bei `multiuser` sieht die Kompatibilitätstabelle der Operationen folgendermaßen aus:

[20] Vgl. Semantik von File-Locking in neueren Unix-Abkömmlingen [Vah96], die sowohl die Semantik von Mutex-Semaphoren als auch von read/write-Locks aus der Datenbank-Welt als Spezialfall umfasst.

[21] Im Fachgebiet der Datenbanken wird diese Eigenschaft als *repeatable read* bezeichnet.

[22] Falls mehrere Instanzen jeweils einen erfolgreichen Read-Lock auf derselben Version halten, dann kann nur eine einzige von ihnen diesen erfolgreich in einen Write-Lock umwandeln, da ansonsten der Schutz gegen Modifikation durch andere Instanzen (*repeatable read*) verloren gehen müsste. Bei klassischen Transaktionen stört dies nicht grundlegend, da die anderen Transaktionen, die dies ebenfalls versuchen, einfach mittels Rollback abgebrochen werden. Ausserhalb von Transaktionen besteht jedoch das Problem, dass ein Update auf die neueste Version nicht durch Transaktions-Abbruch erfolgen kann, sondern „von Hand“ erledigt werden muss (i.d.R. durch Freigabe des inzwischen veralteten Read-Locks, erneutem Lock und `transfer` im `read`-Modus). Wer diesen Modus einsetzt, muss also damit rechnen, dass er nicht immer funktioniert, weil es genau genommen eine *Spekulation* darstellt, bei der darauf spekuliert wird, dass niemand ausser der eigenen Instanz eine Modifikation der gesperrten Daten vornehmen wird. Diese Spekulation lässt sich sicherlich auch ausserhalb von Transaktionen vorteilhaft zur Erhöhung der potentiellen Parallelität einsetzen; dies wird jedoch durch erhöhte Komplexität der Programm-Logik erkauft.

	gr	gw	lr	lw	lu
gr	+	+	+	−	∗
gw	+	+	−	∗	−
lr	+	−	+	−	∗
lw	−	∗	−	−	−
lu	∗	−	∗	−	−

Der erste Buchstabe der Überschrifts-Bezeichnung bezeichnet die Operation g=`get` oder l=`lock`. Der zweite Buchstabe bezeichnet die Lock-Art r=`read`, w=`write` oder u=`upgrade`. Im Falle von `get` bezieht sich der zweite Buchstabe auf den `mode`-Parameter. Es sind die logischen Adress-Zuordnungen sämtlicher gehaltenen Datenblöcke in Betracht zu ziehen, die sich mit dem angeforderten `lock`-Bereich überschneiden.

In der Tabelle bedeutet +, dass die Operation ohne zu warten durchläuft, wenn sie auf die jeweils andere stößt, während − bedeutet, dass gewartet werden muss, bis die andere Operation aufgehoben wurde (d.h. es findet eine Einschränkung der möglichen Ausführungsfolgen statt). Das Zeichen ∗ bedeutet, dass es davon abhängt, ob die andere Operation vom gleichen Aufrufer-Mandat initiiert wurde; falls ja, dann wird nicht gewartet (andernfalls würde ja sofort ein Deadlock entstehen); falls nein, dann muss gewartet werden. An denjenigen Stellen der Tabelle, wo ein − steht, darf die andere Operation, wenn sie von der aktuellen verschieden ist, nicht unter dem selben Aufrufer-Mandat ausgeführt worden sein[23].

Die Lock-Arten `fix` und `reorg` sollen gegen die Effekte aller datenmodifizierenden sowie der später beschriebenen adressmodifizierenden Operationen `move`, `clear` und `delete` schützen:

	o	m	lf	lg
o	+	+	+	−
m	+	+	−	∗
lf	+	−	+	−
lg	−	∗	−	−

In der Überschrift bedeutet o=`other` eine datenmodifiziernden, m=`move` oder `clear` oder `delete` eine adressmodifizierende Operation. Der Endbuchstabe f bedeutet `fix`, g bedeutet `reorg`. Die Adressänderungs-Sperren funktionieren analog zu den klassischen Read-Write-Locks, nur beziehen sie sich ausschließlich auf die a_i-Komponenten der Zustände.

Die hier vorgeschlagenen Tabellen sind möglicherweise in einigen Situationen nicht optimal; andere Varianten sollten in zukünftigen Forschungen evaluiert werden.

Die Realisierung von `lock` und `unlock` ist intern so zu gestalten, dass auf eine Trennung in Spinlocks und kontrollflusswechselnde Locks verzichtet wird (vgl. [K+91, LA93]), und fernerhin eine Benutzung durch Unterbrechungen (interrupts) möglich ist (vgl. [KE95]).

3.3.6. Freiwillige Selbstkontrolle des Verhaltens

Eine Nest-Implementierung garantiert durch die hier propagierten hard Locks wettrenn-freies Verhalten und Schutz gegen Eingriffe anderer Instanzen, und zwar auch dann, wenn andere sich nicht an Locking-Konventionen halten. Diese Garantie nützt jedoch einer Aufrufer-Instanz mit `multi*`-Verhalten nur dann etwas, wenn sie die Locks auch tatsächlich benutzt – wer das „vergisst“, der ist selbst an möglichen Folgen schuld. Bei einem komplexeren System sind jedoch Fehler, die von fehlenden oder falsch gesetzten Locks verursacht werden, extrem schwer reproduzierbar und zu finden. Jede Baustein-Implementierung sollte daher ihr Verhalten durch eine freiwillige Selbstkontrolle mittels folgender Tabelle prüfen:

	gr	gw	lr	lw	lu	lf	lg
gr			¬	¬			
gw			¬	¬	¬		
lr	!	¬		¬	!		
lw	!	!	¬				
lu	!	!	¬				
lf	&	&	&	&	&		¬
lg	&	&	&	&	&	¬	

Hierbei bedeutet g=`get` und l=`lock`. Der letzte Buchstabe hat die gleiche Bedeutung wie bei den vorigen Kompatibilitäts-Tabellen. Die Zeilen bezeichnen die erste Operation, die Spalten die zweite Operation. Beide Operationen beziehen sich auf dasselbe Mandat, unter dem sie ausgeführt werden sollen. Die Tabelle ist daher nicht symmetrisch wie die vorherigen Tabellen, sondern zeigt die Wichtigkeit der Reihenfolge der Operationen. In der Tabelle bedeutet !, dass die zweite Operation nur dann aufgerufen werden darf, wenn die erste bereits ausgeführt wurde (und noch nicht durch eine Gegenoperation aufgehoben wurde). Wenn in einer Spalte mehrere ! auftauchen, dann reicht es, wenn eine der ersten Operationen vorher ausgeführt wurde. Das Zeichen & bedeutet, dass die betreffende erste Operation zusätzlich ausgeführt worden sein muss, wobei eine der mit & markierten Operationen ausreichend ist. Das Zeichen ¬ bedeutet, dass die erste Operation auf gar keinen Fall vorher aufgerufen worden sein darf, bzw. dass sie inzwischen wieder aufgehoben worden sein muss.

Obwohl diese Verhaltens-Selbstkontrolle freiwillig ist, wird sie de facto als Standard für das Verhalten aller Bausteine betrachtet; Ausnahmen sind z.B. zur Inspektion irgendwelcher Zustände oder zu Statistik-Zwecken zulässig, sollten aber begründet werden (z.B. damit, dass der betreffende Vorgang sowieso inhärent wettrenn-gefährdet ist). Die Selbstkontrolle kann z.B. durch interne Verwendung eines Prüf-Bausteins erfolgen, der alle Operations-Aufrufe mitliest und bei Verletzung der Tabelle Warnungen oder Fehlermeldungen ausgibt, ggf. auch zu härteren Sanktionen greift. Da die Information über gehaltene Locks auf jeden Fall in der Bearbeiter-Instanz der Locks geführt werden muss, bietet sich diese ebenfalls als Ort zur performanten Durchführung der an sie delegierten Kontrolle an. Falls die Aufgabenstellung hohe Anforderungen an die Performanz stellt, kann die Selbstkontrolle zu Lasten erhöhter Fehleranfälligkeit auch ausgeschaltet werden.

3.3.7. `get_maxlen` und `set_maxlen`

`get_maxlen(nest)` → `len`
`set_maxlen(nest,len)` → `success`

[23] Andernfalls stellt es einen (ggf. erkennbaren und per Fehlercode sanktionierbaren) Fehler dar.

Damit lässt sich die maximale Länge (maximale benutzbare Adresse plus 1) eines Nestes abfragen bzw. setzen. Bei einigen (eher selten vorkommenden) Geräten lässt sich diese Länge verändern (z.B. bei Bandlaufwerken durch Schreiben einer EOF-Marke oder dergleichen); bei den meisten Geräten liegt die Länge jedoch physikalisch fest.

Größere Bedeutung hat `set_maxlen` bei dynamischen Nestern, wobei sowohl Verkleinerungen als auch Vergrößerungen die Semantik von `delete` in den betroffenen Bereichen ergeben (siehe Abschnitt 3.4.3).

3.3.8. Stream-IO, Pipes, Sockets und Speicherverwaltung

Die bisher beschriebenen Elementaroperationen lösen die Probleme des portionsweisen Zugriffs, des asynchronen IO über Speicherhierarchien hinweg, und der damit verbundenen Speicherverwaltung. Dies funktioniert jedoch nur dann, wenn die logischen Adressen *bekannt sind*, an denen IO geschehen soll. Bei rein sequentiellem IO, wie er z.B. bei sequentiellen Streams, Pipes oder Sockets auftritt, benutzen konventionelle Schnittstellen keine Adressen in der Schnittstelle.

Nester lassen sich zur Nachbildung von Pipes und Sockets als Ringpuffer[24] (mit durch `get_maxlen` beschriebener Länge) ausführen. Es werden logische Adressen verwendet, die jedoch nicht vom Aufrufer, sondern von der Bearbeiter-Instanz der Nest-Operationen vergeben und verwaltet werden. Damit werden weitere Nutzungsarten und Kombinationen möglich[25], insbesondere der *Lookahead* auf noch nicht endgültig konsumierte Daten.

Das Problem der *Atomarität der Reservierung* tritt besonders bei Pipes, Sockets und regulären Dateien im Append-Modus auf. Parallele Write-Operationen stören sich nicht gegenseitig und überschreiben sich nicht, sondern führen letztlich stets zu *irgend einer* serialisierten Folge von Schreiboperationen (auch wenn die konkrete Reihenfolge i.a. nicht determiniert ist). Ebenso ist beim parallelem Lesen aus Pipes und Sockets garantiert, dass die Daten bei irgend einem der Leser genau einmal ankommen werden (auch wenn der konkrete Empfänger wiederum nicht determiniert ist). Auf Nester übertragen, bedeutet dies, dass keine Wettrennen zwischen verschiedenen Aufrufer-Instanzen bei der *Inspektion von Änderungen* in der Addressabbildung statt finden dürfen. Wenn man andererseits den Lookahead auf noch nicht endgültig konsumierte Daten ermöglichen will, dann muss man solche Wettrennen wiederum *ermöglichen* (es ist dann Aufgabe der Konsumenten, mit den damit verbundenen Effekten korrekt umzugehen).

Zur Lösung der Atomarität des Reservierungs-Problems schlage ich vor, die Operationen `get_address` und `put_address` einzuführen. Die Schnittstelle sieht folgendermaßen aus:

```
get_address(nest,min_len,max_len,
  where,lock,action) → (log_address,len)
```

Die beiden Längen-Parameter kennzeichnen das Minimum und das Maximum eines Reservierungs-*Wunsches*. Der boolsche Parameter `where` gibt an, ob die Reservierung im bisher undefinierten Bereich der Adressabbildung statt finden soll (Write-Funktionalität) oder im definierten Bereich (Read-Funktionalität). Falls `action` = `wait` gesetzt wurde und eine Reservierung der Minimallänge nicht möglich ist, blockiert der Aufruf so lange, bis zumindest die Minimal-Reservierung möglich ist (langfristiges Warten[26]). Der boolsche Parameter `lock` bestimmt, ob mehrere Aufrufer parallel jeweils einen reservierten Adressbereich erhalten dürfen, oder ob weitere Bewerber bis zum nachfolgenden `put_address` warten müssen.

Als Ergebnis wird die Adresse des nunmehr reservierten Bereiches zurück geliefert, sowie seine tatsächliche Länge, die zwischen der Minimal- und Maximal-Anforderung liegen kann (in Vielfachen der `transfer_size`); bei Fehlschlagen wird 0 zurück geliefert. Die Nest-Implementierung sichert jeder Aufrufer-Instanz die Atomarität der Reservierung zu, indem parallele Aufrufe von `get_address` in jedem Falle disjunkte Adressbereiche ausliefern. Der Aufrufer besitzt anschließend Kenntnis über einen nur für ihn exklusiv reservierten Adressbereich, mit deren Hilfe er Datenblöcke lesen oder schreiben, und/oder Änderungen an der Addressabbildung mittels `clear` oder `delete` (siehe Abschnitte 3.4.2 ff.) ausführen kann[27].

```
put_address(nest,log_address,len,where)
```

Hebt die Reservierung wieder auf. Jeder Aufrufer von `get_address` hat die Pflicht, einen einmal erhaltenen Adressbereich irgendwann wieder mittels `put_address` freizugeben; der Zeitpunkt und die Reihenfolge bleibt dabei prinzipiell offen.

Die Standard-Pipe-Funktionalität wird durch folgenden Ablauf erzielt:

Leser: der Aufrufer führt als erstes `get_address` aus, dann `get` im erhaltenen Adressbereich; falls die Aktualität der Daten noch nicht gegeben sein sollte, folgt `transfer` im `read`-Modus und `wait`. Nach der Verarbeitung der Daten folgen `delete`, `put` und `put_address`.

[24] Ein so genanntes „Umschlagen“ (wraparound) des zirkulären Puffers lässt sich auch durch Einschieben einer `move`-Operation vermeiden, sobald der Platz am Ende des Puffers nicht mehr reicht.

[25] Getrennte Adressbekanntgabe ermöglicht u.a. eine einfache Lösung für ein Problem, das insbesondere bei Unix zu einer Aufblähung der Systemschnittstellen geführt hat: wenn ein Konsument unter Unix wissen möchte, ob Daten an einem IO-Kanal anliegen, kann er dies mit Hilfe der Systemaufrufe `select` bzw `poll` erfahren; i.a. kann er jedoch nicht die Größe der Daten erfahren; eine Ausnahme ist lediglich die Socket-Schnittstelle, die Lookahead auf Datenpakete unter gewissen Umständen ermöglicht. Diese historisch später nachgerüstete Funktionalität zeigt ein offenbar vorhandenes Bedürfnis auf.

Man kann unter Unix nicht ohne weiteres Daten „auf Verdacht“ oder mehrmals inspizieren (Lookahead), um dann aufgrund ihres Inhaltes zu entscheiden, ob man sie wirklich „konsumieren“ oder zur Verarbeitung durch eine andere Instanz im Puffer belassen will. Dass eine solche Funktionalität auch auf regulären Dateien und Pipes gebraucht wird, ist daran zu sehen, dass sie in der Standard-Benutzerbibliothek `libc` teilweise angeboten wird. Wenn diese Funktionalität nicht erst dort simuliert würde, sondern bereits im Kern integriert wäre, dann würden gewisse durch die doppelte Pufferung im Kern und Benutzer-Adressraum bedingte Anomalien nicht auftreten. Dazu gehört beispielsweise, dass ein Lookahead nicht zwischen verschiedenen Prozessen funktioniert, die aus der gleichen Pipe lesen wollen.

[26] Eventuell sollte man hier noch einen `timeout`-Parameter einführen.

[27] Die atomare Reservierung führt dies jedenfalls nicht selbsttätig aus. Es ist Sache des Aufrufers, wofür er den reservierten Bereich benutzt.

Schreiber: der Aufrufer führt als erstes `get_address`, dann `clear` und `get` aus. Nach der Ablage der Daten folgen `transfer` im `write`-Modus, `put` und `put_address`.

Falls ein Schreiber oder Leser die Daten nicht endgültig konsumieren oder bereit stellen will, kann er auf `delete` bzw `clear` auch verzichten, bzw. die jeweilige Adressraum-Operation durch eine Kompensations-Operation vor dem `put_address` wieder aufheben. Ob und wie viele Daten tatsächlich konsumiert oder bereit gestellt wurden, wird ausschließlich durch Änderungs-Operationen des Definitionsbereiches angezeigt, die von der Reservierung auch nach unten abweichen dürfen. Der Standard-Fall einer vollen Ausnutzung sollte sich durch das Konzept der systematischen Rekombination von Elementaroperationen (vgl. Abschnitt 2.4) effizient lösen lassen.

Wenn der Parameter `lock` nicht gesetzt ist, dann kann die tatsächliche Belegung des Adressraums durch `clear` bzw die Freigabe durch `delete` auch in einer anderen Granularität oder Reihenfolge stattfinden, als die Reihenfolge der Adress-Reservierung vorgab. Es kann sogar passieren, dass gar kein lückenlos zusammen hängender definierter Adressbereich entsteht, wenn einige der Aufrufer nach `get_address` auf die entsprechende Manipulation des Adressraums verzichten oder sie wieder rückgängig machen. Dadurch geht u.U. die Reihenfolge-Beziehung der Datenpakete verloren, die bei der Pipe-Funktionalität notwendig ist, anderen Anwendungen aber die Möglichkeit einer *out-of-order*-Behandlung bietet[28].

Wenn man die `transfer`-Operationen weglässt, erhält man eine nichtpersistente Semantik. Damit kann man die Funktionalität eines Speicher-*Heaps* nachbilden. Daher eignet sich diese Schnittstelle auch für jegliche Speicherverwaltungen, darunter auch Verwalter für interne Objekte fester Größe[29].

3.4. Elementaroperationen auf dynamischen Nestern

Dynamische Nester enthalten alle Operationen eines statischen Nestes sowie zusätzlich die folgenden adressmodifizierenden Operationen. Sie haben mit der Lückenhaftigkeit des Definitionsbereich der Speicherabbildung und mit der Verschiebeoperation zu tun.

Auf Löchern im Definitionsbereich eines Nestes kann nicht gelesen oder geschrieben werden. Dies unterscheidet Nester von der Semantik von Sparse Files unter Unix. Falls letztere Semantik gewünscht wird, ist diese relativ einfach durch einen entsprechenden Anpassungsbaustein implementierbar.

3.4.1. `get_map`

Diese Operation liefert Informationen über die definierten Bereiche eines dynamischen Nestes. Konzeptionell ist dies eine Liste von Paaren, wobei jedes Paar die logische Startadresse eines definierten Bereichs und seine Länge repräsentiert. Repräsentiert wird diese Liste durch ein Array von Paaren, das nach den Startadressen aufsteigend sortiert gehalten wird. Zur effizienten Suche nach Adressen kann daher beispielsweise binäre Suche verwendet werden.

Da ein solches Array in Extremfällen sehr lang werden kann, wird es von `get_map` nicht direkt zurück geliefert, sondern statt dessen wird ein Verweis auf eine weitere Nest-Instanz zurück geliefert, die dieses Array enthält, und von der nur gelesen werden darf.

`get_map(nest)` $\rightarrow$ `nest`

Die von `get_map` zurück gelieferte Nest-Instanz wird *adjungiertes Nest*[30] genannt. Sie enthält den Definitionsbereich des Original-Nestes in der oben beschriebenen Repräsentation. Adjungierte Nester haben ihrerseits keine Löcher, d.h. ihr adjungiertes Nest hat nur noch eine triviale Struktur mit einem einzigen Eintrag[31]

Die im adjungierten Nest enthaltene Liste hat eine besondere Bedeutung im Falle von Geräten mit dynamischen Block- oder Sektorgrößen (insbesondere Bandlaufwerke, bei denen jeder Datenblock eine individuelle Größe haben kann, oder sonstige zeichenorientierte Geräte, bei denen es auf die Länge von einzelnen Transfers ankommt), bei Pipes und bei Netzwerk-Sockets, bei denen die Paketgröße eines einzelnen Original-Eintrags vom Empfänger ermittelbar sein *muss*[32]. In all diesen Fällen repräsentiert das adjungierte Nest dicht aneinander liegende Adressbereiche (ohne dazwischen liegende Lücken). Damit kann jeder Verwender selbst entscheiden, ob er einen Unix-typischen Zugriff auf das Nest unter Missachtung der ursprünglichen Paket- bzw. Datensatzgrenzen machen will, oder ob er zuvor die Paket- bzw. Datensatzgrenzen im adjungierten Nest nachsehen und entsprechend berücksichtigen will.

Ein dichtes Aneinanderliegen von definierten Bereichen ist auch bei persistenten Nestern möglich. Die Benutzer werden jedoch gebeten, derlei in der Praxis möglichst zu meiden, da es zu Verständigungsproblemen mit den Datei-Paradigmen anderer aktueller Betriebssysteme kom-

[28] Beispielsweise zur Verwaltung von IO-Aufträgen für Festplatten-Gerätetreiber, wo Umordnungen der Reihenfolge zur Durchsatzsteigerung eingesetzt werden.

[29] so genannte Objekt-Caches [Vah96] lassen sich sehr einfach dadurch realisieren, dass man die Objekt-Größe als `transfer_size` benutzt.

[30] Diese Lösung hat den Vorteil, dass die interne Verwaltung des Definitionsbereiches durch lokale Baustein-Instanzen geschehen kann, die das adjungierte Nest direkt verwalten. Erweiterungen und Löschungen im Definitionsbereich lassen sich direkt durch `move`-Operationen auf dem adjungierten Nest realisieren. Ansonsten dürfen Implementierungen von Nestern ihren Definitionsbereich auch auf beliebige andere Weise intern realisieren, sofern sie ein virtuelles adjungiertes Nest zur Verfügung stellen, das seinen Inhalt auf Anforderung dynamisch generiert.

[31] Mit `get_map`(s) sei das zu einem beliebigen Nest s gehörige adjungierte Nest notiert. Die n-fache Komposition der `get_map`-Funktion mit sich selbst sei mit `get_map`^{n} notiert. Dann gilt für $n > 2$: get_map$^n(s)$ = get_map$^{n+1}(s)$. Das dreifach adjungierte Nest beschreibt sich also selbst.

[32] Das von BSD stammende Socket-Konzept und seine Schnittstelle ist heute in praktisch allen bedeutenden Betriebssystemen realisiert, auch in den Betriebssystemen von Microsoft, in allen modernen Unix-Varianten, und ist selbst in neueren Versionen von IBM-Großrechner-Betriebssystemen nachgerüstet worden. Da Sockets je nach Modus und Umständen das Lesen von Paketen sowohl mit Beachtung der beim Schreiben vorgegebenen Paketgrenzen, als auch nach dem Unix-Paradigma mit aufgelösten Paketgrenzen als unabhängige Folge von Teilblöcken unterstützen und verlangen, kommen ernsthafte Betriebssystem-Entwürfe im Endeffekt nicht darum herum, beide Paradigmen zu unterstützen. Wenn man schon beide Paradigmen unterstützen *muss*, dann sollte dies auf einheitliche Weise, bei möglichst vielen Arten von Nestern und auf möglichst allen Ebenen geschehen.

men könnte[33].

Konzeptuell gesehen ist ein Nest somit weit mehr als ein File unter Unix, da es untrennbar mit seinem Definitionsbereich verknüpft ist, der bei dem hier vorgeschlagenen Entwurf in einem adjungierten Nest repräsentiert wird.

Bei der Repräsentation eines Unix- oder Windows-Files in einem Nest enthält das zugehörige adjungierte Nest nur einen einzigen Eintrag mit der Startadresse 0 und der Länge des Files[34]. Die mittels `get_maxlen` und `set_maxlen` zugreifbare Maximaladresse hat hierbei nur die Funktion einer Sicherheitsschranke ähnlich einer Quota, die nichts über den tatsächlichen Platzbedarf aussagt und ohne weiteres auch Werte wie z.B. $2^{63} - 1$ annehmen kann.

3.4.2. `clear`

`clear(nest,log_address,len,mode)` → `success`

Damit wird der Definitionsbereich eines Nestes an der angegebenen Stelle und Länge erweitert, soweit sich vorher dort eine Lücke befand. Falls anschließend in diesem Bereich des Adressraumes gelesen wird, erscheinen Null-Byte-Datenblöcke.

Dies gilt auch dann, wenn vorher bereits Datenblöcke dem betroffenen Bereich ganz oder teilweise zugeordnet waren. In diesem Fall bleiben bereits mittels `get` ausgelieferte Datenblöcke weiterhin verwendbar, allerdings werden später abgesetzte oder bereits gepufferte `transfer`-Operationen auf diesen Blöcken nicht mehr tatsächlich ausgeführt. Nach endgültiger Freigabe durch `put` werden derartige *verwaiste* (*orphane*) Datenblöcke sofort zum Speicher-Recycling verwendet.

Falls `get` einen Block mit Vielfachen der `transfer_size` ausgeliefert hat, der von einem nachfolgenden `clear` eventuell nur zu einem Teil betroffen ist, muss die interne Logik der Nest-Implementierung sicherstellen, dass bei nachfolgenden Operationen (insbesondere `put` und `transfer`) der nicht betroffene Teil dieses Blocks wie ein normaler Block, der betroffene Teil dagegen als verwaist behandelt wird.

Falls `clear` einen bereits definierten Bereich berührt oder mit ihm ganz oder teilweise überlappt, dann findet standardmäßig eine Erweiterung bzw. Verschmelzung der Definitionsbereiche im adjungierten Nest statt. Ggf. werden Löcher auch vollständig aufgefüllt und benachbarte Bereiche mit einander zu einem größeren Bereich verschmolzen.

Durch Setzen des boolschen Parameters `mode` lässt sich dieses Standard-Verhalten so abändern, dass genau an den Grenzen des `clear`-Bereiches Einträge im adjungierten Nest so erzeugt bzw. verändert werden, dass der Bereich als ein einziges Datenpaket mit ggf. lückenlosem Anschluss an (ggf. erst dadurch entstandene) benachbarte Bereiche erscheint. Falls vorher im `clear`-Bereich mehrere kleinere definierte Bereiche gelegen waren, dann werden sie ebenfalls „platt gemacht“.

[33] Dieser Mechanismus kann auch dazu dienen, File-Paradigmen aus anderen (meist älteren) Betriebssystemen nachzubilden, die nicht das Unix-Paradigma vom File als Folge von Bytes verfolgen.

[34] In diesem Fall gilt bereits für $n > 1$: get_map$^n(s)$ = get_map$^{n+1}(s)$.

3.4.3. `delete`

`delete(nest,log_address,len)` → `success`

Diese Operation bewirkt, dass an der angegebenen Stelle und Länge ein Loch im Definitionsbereich entsteht bzw. bereits vorhandene Löcher ggf. erweitert werden. Falls betroffene Datenblöcke bereits vorher mittels `get` ganz oder teilweise in Verwendung genommen worden waren, dann werden diese bei den IO-Operationen und beim Speicher-Recycling analog zu `clear` als verwaist behandelt.

3.4.4. `move`

`move(nest,log_address,len,offset)` → `success`

Ein durch die angegebene Adresse und Länge bezeichneter Bereich des Adressraums wird um den Offset (positiv oder negativ) verschoben. Der Inhalt der verschobenen Datenblöcke wird hierbei nicht verändert, lediglich die Zuordnung von logischen zu physischen Adressen wird verändert. Bereits mittels `get` ausgelieferte Blöcke, die vollständig im Quellbereich der Verschiebung liegen, bleiben samt ihren physischen Adressen weiterhin gültig, jedoch wird ihre interne transiente Zuordnung gemäß der Verschiebung angepasst. Beim späteren `put` werden sie so behandelt, als hätten sie schon immer an der neuen Adresse gelegen. Quell- und Zielbereich der Verschiebung dürfen sich unabhängig von der Verschieberichtung überlappen[35]; gegenüber `fix`- und `reorg`-Locks sind grundsätzlich beide Bereiche als relevant zu betrachten, auch wenn sie sich nicht überlappen. Im frei gewordenen Quellbereich erscheint ein Loch. Falls sich im Zielbereich bereits definierte Bereiche befinden, werden diese „verdeckt“; bereits durch `get` ausgelieferte Datenblöcke werden analog zu `delete` als verwaist behandelt. Eventuelle im Quellbereich vorhandene Löcher werden mitverschoben; die Freigabe von bereits vorhandenen Datenblöcken im Zielgebiet erfolgt selbst dann, wenn ausschließlich ein Loch dorthin verschoben wurde[36].

Falls vor dem `move` bereits Blöcke mittels `get` angefordert worden waren, die vollständig im Quellbereich lagen, dann werden sie anschließend so behandelt, als hätten sie schon bei der Anforderung im Zielbereich gelegen. Bei Blöcken mit einer vielfachen Länge der `transfer_size`, die von der Verschiebung nur teilweise betroffen sind, muss die interne Verwaltungslogik die interne transiente Zuordnung aufspalten und getrennt behandeln, so dass die einzelnen Teile bei folgenden Operationen wie z.B. `put` oder `transfer` so behandelt werden, als wären sie ursprünglich nicht am Stück angefordert worden.

Eine effiziente Realisierung von `move` ist möglich und wird in Abschnitt 4.1.3 (Baustein `map_simple`) näher detailliert.

[35] Eine Variante von `move` könnte den überdeckten Bereich des Zielgebietes in den gleichgroßen freigewordenen Bereich des Quellgebietes in einer einzigen atomaren Operation verschieben, so dass im Endeffekt eine Art „Rotationsoperation“ ohne jeglichen Verlust von Daten ausgeführt wird. Damit läßt sich insbesondere eine Vertauschung von Adressbereichen realisieren. Im Moment sehe ich für diese Variante nur geringe praktische Notwendigkeit, jedoch bildet diese Variante von `move` zusammen mit `clear` und `delete` eine Algebra von Operationen auf Adressräumen mit sehr schönen mathematischen Eigenschaften.

[36] Würde man ∞ als zulässige Quell- und Ziel-Angaben für die Verschiebung einführen, könnte man damit nebenbei auch die Semantik von `clear` und `delete` erfüllen.

3.4.5. `get_meta`

`get_meta(nest)` → `nest`

Zur Verwaltung von Information über den Zustand einer Nest-Instanz kann dieser eine Hilfs-Nest-Instanz zugeordnet sein, die *Meta-Nest* genannt wird[37]. Falls das Meta-Nest nicht existiert, wird ein NULL-Kennzeichen zurück geliefert.

Meta-Nester sind dazu gedacht, um Informationen *über* ein Nest bereit zu stellen, analog zu Datei-Attributen oder Inode-Informationen (siehe [Bac86]), jedoch nicht ausschließlich auf diese Verwendungszwecke beschränkt. Meta-Nester werden später in Abschnitt 6.1 eine besondere Rolle bei der Einführung von Typsystemen spielen, ebenso in Abschnitt 4.2 bei der Auto-Instantiierung von Bausteinen und in Abschnitt 4.1.5 bei den Lokalitätseigenschaften von Zugriffen auf Verzeichnis-Hierarchien.

Ein Meta-Nest sollte möglichst nur einen einzigen definierten Bereich ohne Löcher enthalten, der idealerweise nur relativ wenig Platz beanspruchen sollte (d.h. er sollte nicht zur Speicherung großer Datenmengen missbraucht werden), um ein performantes Zugriffsverhalten sicherzustellen. Wegen der beabsichtigten Kleinheit sollte ein Meta-Nest möglichst eine kleine `transfer_size`, ggf bis hinab zu 1, unterstützen.

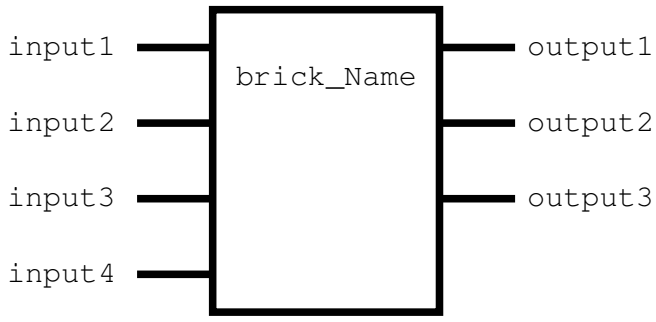
Um in einem Meta-Nest verschiedene Arten von Meta-Informationen unterzubringen, müssen diese durch geeignete Konventionen voneinander unterschieden werden. Der Inhalt eines Meta-Nestes muss also ebenfalls *interpretiert* werden (vgl. Abschnitt 6.1). Eine Darstellung dieser Interpretations-Vorschriften kann wiederum in einem Meta-Meta-Nest erfolgen. Eine rekursive Zuordnung von Meta-Nestern zu Meta-Nestern muss allerdings irgendwann terminieren; wo dies zu erfolgen hat, und wozu die Inhalte von Meta-Nestern im einzelnen dienen sollen, ist ein *freier Parameter* der hier vorgestellten Architektur.

Bei einem persistent gehaltenen Original-Nest muss das zugehörige Meta-Nest, sofern es existiert, ebenfalls persistent gehalten werden. Falls absturzsichere atomare Transaktionen implementiert werden, muss das Meta-Nest bezüglich dieser Semantik wie ein Teil des Original-Nestes behandelt werden.

Meta-Nester lassen sich grundsätzlich auch zur Repräsentation statischer und dynamischer Attribute benutzen; dies führt zu einer Vereinfachung der zu implementierenden Konzepte, macht jedoch die Existenz (virtueller) Meta-Nester vor der Instantiierung des Haupt-Nestes erforderlich.

[37] Wenn man die Zuordnung mehrerer unterschiedlicher Meta-Nester erlaubt, sind beispielsweise Simulationen von „reflektiven“ Betriebssystem-Architekturen [Yok92], die zwischen verschiedenen Meta-Informationen wie Klassen und Reflektoren durch verschiedene Instanzen unterscheiden, einfacher möglich.

4. Bausteine



Eine Baustein-Instanz ist ein Objekt, das eine beliebige, eventuell auch zur Laufzeit sich ändernde Anzahl von Ein- und Ausgängen besitzt, die wiederum jeweils Instanzen von Nestern darstellen.

Baustein-Instanzen werden ähnlich wie in der Elektro- und Digitaltechnik als Kästchen mit linksseitigen Eingängen und rechtsseitigen Ausgängen gezeichnet (vgl. Function Block Diagram in [IEC]). Als Verdrahtungsregel gilt, dass Eingänge nur mit den Ausgängen anderer Bausteine verknüpft werden dürfen und umgekehrt, wobei ein Eingang nur mit einem einzigen Ausgang, ein Ausgang hingegen i.A. mit mehreren Eingängen verknüpft werden darf.

Ein Ausgang stellt eine Nest-Instanz anderen Bausteinen zur Verfügung, wobei deren Eingänge als *Konsumenten* der vom Ausgang angebotenen *Dienstleistungen* anzusehen sind. Wir haben damit ein System von *Produzenten* und von *Konsumenten* im *logischen* Sinne. Eine Verdrahtungs-Leitung dient dem *Transport* von Dienstleistungen. Darüber hinaus repräsentiert sie eine *Hierarchie-Beziehung*[1] (vgl. Abschnitt 7.1) zwischen einer *vorgeschalteten* Baustein-Instanz (Produzent der Dienstleistung) und einer *nachgeschalteten* (Konsument der Dienstleistung). Ein wesentliches Merkmal der Hierarchie-Beziehung ist, dass sie zwischen *Instanzen* gilt (damit unterscheidet sie sich z.B. von der Objektorientierung, wo Hierarchie-Beziehungen auf *Klassen*-Ebene betrachtet werden; vgl. Abschnitt 6.2.2 und Kapitel 7).

Per Konvention definieren wir als „Stromrichtung“ einer Verdrahtungs-Leitung die Richtung vom Produzenten zum Konsumenten; dies hat jedoch nichts mit möglichen Datenfluss-Richtungen zu tun, denn die Stromrichtung ist zwar gleichzeitig die logische Datenfluss-Richtung beim Lesen, doch die logische Schreib-Datenflussrichtung läuft dem entgegen (was am Anfang zu Verwirrung führen kann, ebenso die baustein-interne Weitergabe von Operations-Aufrufen, die intern meistens von den Ausgängen her zu den Eingängen verläuft). Eine Leitung stellt alle Nest-Operationen, die am Ausgang eines Bausteins zur Verfügung gestellt werden, einem oder mehreren Konsumenten zur Verfügung. Da dies auch die Operationen `get_map` und `get_meta` umfasst, wird auf diese Weise das zugehörige adjungierte Nest und das Meta-Nest verfügbar gemacht.

Sinn der Bausteine ist, verschiedene Transformationen sowohl des Adressbereiches als auch eventuell des Inhaltes von Nestern, gelegentlich auch von Meta-Nestern oder von Operations-Nestern (siehe Kapitel 6) oder von Nest-Attributen wie `transfer_size` oder von (Zugriffs-)Modellen durchzuführen. Durch Kombination der Bausteine zu komplexen Netzwerken ergeben sich Transformationsmöglichkeiten, die mit herkömmlichen Betriebssystem-Architekturen nur sehr schwer realisierbar sind.

Bausteine besitzen mindestens eine Instantiierungs- und Konstruktor-Operation (vgl. Abschnitt 4.2.1), mit der sich neue Instanzen des jeweiligen Baustein-Typs erzeugen lassen, wobei die Parameter von Konstruktor-Operationen i.d.R. baustein-spezifisch sind. Die Destruktor-Operation hat hingegen eine einheitliche Schnittstelle. Einige Baustein-Typen haben darüber hinaus weitere spezifische Operationen, mit denen sich ihr Verhalten (etwa die Anzahl der Ein- und Ausgänge) zur Laufzeit steuern lässt. Zur nachträglichen Instantiierung von Ein- und Ausgängen kann zwischen der Instantiierung von Bausteinen und derjenigen von Ein- und Ausgängen unterschieden werden.

Eine automatisierte Überprüfung der Kompatibilität der Kompetenzen von Ausgängen mit dem Verhalten von Eingängen (vgl. Abschnitt 2.5) sollte bei der Verdrahtungs-Operation stattfinden; beispielsweise sollte das Zusammenpassen des `transfer_size`-Attributs überprüft werden. Weitere Details dazu in Abschnitt 4.2. Viele der nachfolgend vorgestellten Bausteine lassen sich in verschiedenen Ausbaustufen[2] und mit verschiedenen Kompetenzen und Verhalten implementieren.

4.1. Beispiel-Baustein-Arten

Beim Entwurf von Bausteinen sollte darauf geachtet werden, dass möglichst wenig Redundanz zur Funktionalität bereits vorhandener Baustein-Arten auftritt. Sinn der Zerlegung in Bausteine ist, die in einem Betriebssystem insgesamt zu lösenden Aufgaben in möglichst viele, kleine, voneinander möglichst unabhängige (orthogonale) Teile und Zuständigkeiten aufzuspalten.

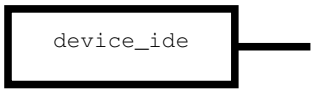
Ich habe mich bemüht, bei der hier als beispielhaft zu verstehenden Zerlegung möglichst nur solche Aufgaben zu stellen, die auch in anderen aktuellen Betriebssystem-Entwürfen (einschließlich Netzwerk-Betriebssysteme) oder in Datenbanken irgendwo auftauchen und dort mit teils hohem Aufwand ad hoc gelöst werden. Damit möchte ich aufzeigen, dass eine Komponenten-Zerlegung auf Basis der Abstraktionen Nest und Baustein den Gesamtaufwand

[1] Dies hat Ähnlichkeit mit einem Schichtenmodell. Ein reinrassiges Schichtenmodell läßt sich z.B. durch eine Kette von Baustein-Instanzen nachbilden. Im Unterschied zu Schichtenmodellen werden auch *nebenläufige Hierarchien* und graphenartige Strukturen unterstützt, die von einer Baumstruktur abweichen. Im Allgemeinen sind jedoch keine zyklischen Verdrahtungen erlaubt.

[2] Wenn ein neu entwickelter Baustein im `singleuser`-Modus stabil läuft, dann kann er schrittweise bis zur `multiversion`-Kompetenz bzw. -Verhalten erweitert werden.

der zu implementierenden Aufgaben mindern kann, da bereits die Kombination von wenigen wieder verwendbaren Baustein-Arten eine mächtige Funktionalität erzeugt.

4.1.1. device_*



Diese Bausteine haben im Regelfall keinen Eingang[3] und genau einen Ausgang, womit sie den Inhalt eines Gerätes (beispielsweise device_ide für IDE-Festplatten) als statisches Nest den etwaigen Konsumenten zur Verfügung stellen. Im Sinne der Verdrahtungslogik der Bausteine stellen sie Produzenten oder auch „Datenquellen“ dar.

Eine prinzipielle Unterscheidung zwischen Block- und Character-Devices wie bei Unix ist nicht notwendig, da der Entwurf der Nest-Operationen beide Paradigmen und die darin vorkommenden Betriebsarten auf uniforme Weise unterstützt; der Unterschied wird lediglich im Wert des transfer_size-Attributs angezeigt.

Normalerweise dienen Geräte-Treiber zum Transfer von Datenblöcken auf exklusiv genutzte Peripheriegeräte und brauchen daher nur die Operationen statischer Nester und nur singleuser-Kompetenzen[4] zu implementieren.

Eine mindestens multiuser-fähige Sonderform ist device_mem, die transienten Speicher zur Verfügung stellt, indem sie die Operationen get_address- und put_address implementiert. Um Speicher mit verschiedenen transfer_size-Werten verwalten zu können, bilden mehrere Instanzen von device_mem eine mit einander verdrahtete Hierarchie, bei der sich Verwalter kleiner Granularität den Speicher über einen Eingang bei Verwaltern mit größerer Granularität „ausleihen“ (vgl. Abschnitt 5.1). Der „Urverwalter“ des physischen Hauptspeichers besitzt den gesamten Speicher beim Start des Betriebssystems und hat daher keinen Eingang[5].

Eine weitere multiuser-fähige Sonderform ist device_ramdisk, die eine limitierte Persistenz implementiert, die sich nicht über Stromausfälle hinweg erstrecken muss[6].

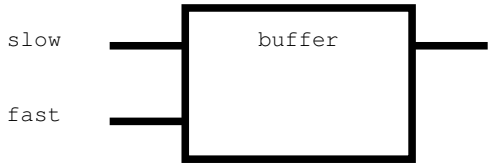
[3] Dies ist eine konzeptuell vereinfachende Darstellung. Konkrete Hardware besitzt oft eine innere Hierarchie, etwa Peripherie-Busse wie PCI oder Geräte-Busse wie SCSI. Zur Realisierung deren (Unter-)Treiber eignen sich prinzipiell ebenfalls die Abstraktionen Nest und Baustein, so dass der „eigentliche“ Geräte-Treiber auch einen oder mehrere Eingänge haben kann, mit denen er an die weitere Unter-Treiber-Infrastruktur angeschlossen wird. Weiterhin lassen sich Hardware-Konzepte wie Memory-Mapped-IO, Grafikkarten-Framebuffer und dergleichen leicht und effizient durch die Abstraktion der Nester darstellen.

[4] Die Einführung multiuser-fähiger Treiber könnte auf dem Gebiet der Storage Area Networks (SAN) Vorteile bringen, insbesondere die Last besser verteilen.

[5] Einige Hardware-Architekturen wie z.B. die IBM z-Serie erlauben den Austausch von Speichermodulen zur Laufzeit. Dies lässt sich dadurch modellieren, dass der Urverwalter die Anzahl seiner Ressourcen ändert. Dazu muss er notfalls bereits vergebene Ressoucen wieder zurückfordern; siehe Kapitel 5.

[6] Eine mögliche Verklemmung des Gesamtsystems durch im Normalbetrieb nicht zurückforderbare (quasi-persistente) Datenblöcke muss vermieden werden. Eine einfache Lösung dafür ist die Forderung, dass die Summe der Maximallängen aller device_ramdisk-Instanzen den verfügbaren Hauptspeicher nicht überschreiten darf. Alternativ dazu kann auch das Verfahren von Habermann [Hab69] angewandt werden, mit dem eine Überbuchung der vorhandenen Hauptspeicher-Ressourcen prinzipiell möglich ist (jedoch wegen des damit verbundenen Wartens unakzeptable Auswirkungen auf das von Benutzern erwartete Verhalten von interaktiven Anwendungen oder von Realzeit-Anwendungen haben kann).

4.1.2. buffer



Ein buffer-Baustein sorgt für die Entkoppelung von Aktivitäten zwischen Eingang und Ausgang (in beide Richtungen). Er eignet sich zur Adaption des *zeitlichen Zugriffsverhaltens* zwischen langsamen und schnellen Baustein-Instanzen; im Idealfall sollte er die einzige Stelle im Gesamtsystem darstellen, die die Probleme der so genannten „Speicherlücke“ zu lösen hat.

Ein häufiger Anwendungsfall ist die Nachschaltung hinter ein device_*. Dazu sind oft nur die Operationen statischer Nester erforderlich, und der slow-Eingang braucht nur singleuser-Verhalten zu zeigen; auch der Ausgang braucht nur singleuser-Kompetenz bereitzustellen, da im häufigsten Anwendungsfall nur eine einzige map_*-Instanz nachgeschaltet wird. Es gibt aber auch Anwendungen, bei denen mindestens multiuser-Verhalten und -Kompetenz benötigt wird, insbesondere die Nachschaltung hinter einen remote-Baustein (siehe Abschnitt 4.1.12).

Bei der Implementierung des multiuser-Verhaltens tritt das in der Literatur bekannte Problem der *Cache-Kohärenz* (vgl. [AB86, LH89, HP95]) mit einer vorgeschalteten Instanz auf. Durch die in der Nest-Schnittstelle vorgesehenen notify_*-Operationen (vgl. Kapitel 5) ist dies vergleichsweise leicht lösbar.

Ein Baustein-Typ ist *statuslos*, wenn er zu jedem Zeitpunkt, in dem sich kein logischer Kontrollfluss in ihm aufhält, destruiert und anschließend erneut konstruiert werden kann, ohne dass dadurch ein geändertes Verhalten von außen sichtbar wird.

Ein Baustein-Typ ist *pseudo-statuslos*, wenn er jederzeit durch Aufruf einer weiteren speziellen Elementaroperation die Eigenschaft der Statuslosigkeit erreichen kann. Ein pseudo-statusloser Baustein darf also einen temporären Status enthalten, der auch zwischen verschiedenen Elementaroperations-Aufrufen lokal gespeichert werden darf; dieser Status muss jedoch jederzeit auf Anforderung in eins der Eingangs-Nester vollständig ausgelagert werden können.

buffer kann auf folgende Weise (pseudo-)statuslos implementiert werden: der mit slow bezeichnete Eingang dient zur Entkoppelung der Zugriffs-Häufigkeiten und -Anzahlen. Das mit fast bezeichnete Eingangs-Nest enthält den *gesamten Status* des Puffers; dazu gehört neben den Inhalten der zu puffernden Datenblöcke auch die transiente Zuordnung zwischen logischen und physischen Adressen und der Aktualitäts-Status der Puffer-Datenblöcke.

Damit dies zu guter Performanz führt, muss fast sehr schnellen Zugriff bieten. Die Idee besteht darin, an diesem Eingang wahlweise ein device_ramdisk anzuschließen, oder irgend ein anderes relativ schnelles Ge-

rät; bei der Pufferung von Zugriffen auf extrem langsame Bandlaufwerke[7] kann dies beispielsweise auch ein Nest sein, das auf Festplatte vorgehalten wird. Letztlich macht `buffer` nichts anderes, als das *zeitliche Zugriffsverhalten* des `fast`-Eingangs an den Ausgang weiter zu reichen. Die am `fast`-Eingang angeschlossene Instanz wird mit geringeren Datenmengen belastet als beim `slow`-Eingang vorhanden sind; bei Speichermangel darf sie Speicher-Rückforderungs-Anträge (vgl. Kapitel 5) stellen.

Die explizite Benutzung von `device_ramdisk` als `fast`-Eingang hat den weiteren Vorteil, dass die Probleme der Speicher-Rückforderung, sowie der im Gesamtsystem gemischt verwendeten unterschiedlichen `transfer_size`-Blockgrößen ausschließlich dort zu lösen sind (Lokalitäts-Prinzip).

Wenn man `buffer`-Bausteine als die einzige Stelle im Gesamtsystem ansieht, die die Entkoppelungs-Problematik des *zeitlichen Zugriffsverhaltens* löst, und zwar *sehr effizient* löst, dann kann man einen Schritt weiter gehen: man kann fast alle anderen Bausteine ebenfalls (pseudo-)statuslos implementieren. (Pseudo-)Statuslosigkeit ermöglicht eine deutliche Reduktion der inneren Komplexität vieler Bausteine im Vergleich zu konventionellen Implementierungen, weil die Verantwortung zur korrekten Aufbewahrung der internen Zustandsinformation an einen untergeordneten Baustein *delegiert* wird; daher ist Statuslosigkeit hochgradig erstrebenswert. Sie setzt allerdings voraus, dass Zugriffe über eine dermaßen entkoppelte Schnittstelle *fast nichts kosten*. Durch standardmäßige Benutzung von Prozeduraufrufen als Schnittstellen-Mechanismus und durch die Zero-Copy-Architektur (vgl. Abschnitt 2.8.2) wird dies angenähert.

Aufgrund von absehbaren Fortschritten in der Hardware-Entwicklung ist zu erwarten, dass zukünftige Rechner ganz andere interne Speicher-Hierarchien besitzen werden, als sie heute üblich sind, so dass `buffer`-Instanzen an anderen Stellen als heute vorhersehbar erforderlich werden. Das Baustein-Konzept ermöglicht eine flexible Anpassung an geänderte Rahmenbedingungen.

4.1.3. `map_*`

Aufgabe dieses Baustein-Typs ist, ein statisches Nest in ein dynamisches umzuwandeln. Es gibt im Regelfall nur einen Eingang[8] (der meist nur `singleuser`-Verhalten[9] zu implementieren braucht) und einen Ausgang, der im Falle von Netzwerk-Betriebssystemen mindestens `multiuser`-Kompetenz bereit stellen sollte; die Herstellung dieser Kompetenz kann aber auch an eine nachgeschaltete oder interne `adaptor_*`-Instanz (siehe Abschnitt 4.1.8) delegiert werden.

Es lassen sich verschiedene Arten von `map_*`-Bausteinen realisieren, die ihre Aufgabe mit jeweils anderen internen Realisierungsverfahren lösen und an verschiedene Last- und Benutzungsmodelle angepasst sind. Als Beispiele werden nun zwei mögliche Realisierungen beschrieben, `map_simple` und `map_simple_delta`.

Die Umwandlungs-Aufgabe wird speziell bei `map_simple` folgendermaßen erledigt: es wird angenommen, dass die Maximalgrößen des Eingangs- und Ausgangs-Nestes annähernd gleich sind, und dass der Konsument am Ausgang nur Anforderungen in Blockgrößen stellt, wie sie von der MMU für die virtuelle Speicherverwaltung gestellt werden (typischerweise 4KByte oder 8KByte). Das `transfer_size`-Attribut des Ausgangs-Nestes wird also auf eine derartige Blockgröße gesetzt. Dies bedeutet, dass alle Operationen, insbesondere auch die `move`-Operation, nur in Vielfachen dieser Transfergröße erfolgen dürfen. Damit liegt eine effiziente Realisierung bereits auf der Hand: man benutze eine Tabelle, die eine Abbildung der Blocknummern bzw -Adressen des Ausgangs-Nestes auf diejenigen des Eingangs-Nestes realisiert. Diese Tabelle ist größenordnungsmäßig etwa um den Faktor 1000 kleiner als die Größe des umzuwandelnden Nestes, verursacht also einen Platz-Overhead von etwa einem Promille, der sinnvollerweise am Anfang des Eingangs-Nestes vorreserviert wird. Alle Operations-Aufrufe, die etwas mit den virtuellen Adressen zu tun haben (`get` und ggf. `lock` / `unlock`) und die vom Ausgang her ankommen, werden nach Übersetzung durch diese Tabelle an den Eingang durchgereicht. Die `move`-Operation wird durch eine Verschiebung innerhalb der Tabelle realisiert, die aus den bereits dargestellten Gründen etwa um den Faktor 1000 weniger kostet, als wenn man `move` durch eine Serie von Blocktransfer- und Kopieroperationen im statischen Eingangs-Nest realisiert hätte (was z.B. ein Baustein mit dem Namen `map_braindead` besorgen könnte). Diese Eigenschaft lässt sich dazu ausnutzen, um den Aufwand weiter zu senken: wenn man die Tabelle wiederum in einem eigenen privaten Nest realisiert, kann dieses wiederum mit Hilfe einer beliebigen anderen `map_*`-Instanz verwaltet werden, die lediglich die `transfer_size`-Granularität der Tabellenelemente unterstützen sollte (beispielsweise `map_simple_delta`, `map_braindead`, oder eine Kombination aus `map_simple` mit einem nachgeschalteten `adaptor_*`, der in Abschnitt 4.1.8 beschrieben wird). Bei günstiger Wahl und Kombination der Bausteine zu mehrstufigen Kaskaden lassen sich damit die Kosten insbesondere bei sehr großen Nestern nochmals um einige Zehnerpotenzen senken.

Eine Adressübersetzung verursacht nach längerem Betrieb mit vielen `move`-, `clear`- und `delete`-Operationen ein Phänomen, das in der Literatur über Dateisysteme als *Fragmentierung* (vgl. z.B. [MJLF84, McK96]) bekannt ist.

[7] Früher galt die Grundregel: je mehr Kapazität ein Peripheriegerät bereitstellt, desto langsamer ist es. Zum Zeitpunkt der Niederschrift dieser Arbeit haben billige als Massenprodukte hergestellte Festplatten die 100-GByte-Grenze durchbrochen und damit die gleiche Größenordnung wie Bandlaufwerke erreicht (vielleicht noch mit Ausnahme weniger sehr teurer High-End-Modelle). Auch die Medienkosten pro GByte nähern sich langsam derselben Größenordnung. Ähnliche Verschiebungen von Speicher-Hierarchien könnten sich auch in Zukunft ereignen.

[8] Andere Varianten mit mehreren Eingängen sind ebenfalls denkbar, insbesondere wenn der Haupt-Eingang trotz Hinzunahme einiger dynamischer Operationen ständig den gleichen Inhalt (oder einen angenäherten Inhalt) wie der Ausgang haben soll; zusätzliche Status-Information kann in diesem Fall in einem separaten Hilfseingang gehalten werden.

[9] Die Herstellung von `multiuser`-Eingängen ist möglich, aber je nach gewählter interner Struktur u.U. aufwendig. Auch bei Client-Server-Anwendungen ist es sinnvoll, ein statisches Nest erst mittels `map_*` in ein dynamisches umzuwandeln, bevor es z.B. mittels `remote` exportiert wird.

Dieses Phänomen beschreibt die Tatsache, dass eine hohe Lokalität von Zugriffen auf der Ausgangsseite des `map`-Bausteins nicht unbedingt in eine hohe Lokalität der Zugriffe auf der Eingangsseite übersetzt wird, da die Tabelle die Adressen ähnlich wie bei Hash-Verfahren kräftig durcheinander würfeln kann. Ein Konsument erwartet jedoch, dass Schreib- oder Leseaufträge, die in zusammenhängender und aufsteigender Adress-Reihenfolge gegeben werden, mit deutlich besserer Performanz ausgeführt werden als zufällig verteilte Zugriffe[10]. Zur Lösung dieses Problems stehen mehrere bekannte Verfahren zur Verfügung, die aus der Literatur über Dateisysteme analog übertragbar sind, beispielsweise intelligente Allokations-Strategien und Defragmentierungs-Läufe. Durch das Background-IO-Konzept (Abschnitt 2.8.3) wird insbesondere eine Defragmentierung im Hintergrund des laufenden Betriebes stark erleichtert, ohne die Abarbeitung der Benutzeraktivitäten merklich zu stören[11].

Wenn man davon ausgehen kann, dass eine Defragmentierung im Hintergrund dafür sorgt, dass die Adress-Übersetzung „nur wenig“ von einer Identitäts-Abbildung abweicht, dann lässt sie sich auch auf folgende Weise realisieren, die ich mit `map_simple_delta` bezeichne:

Statt einer Tabelle fester Größe wird eine Liste (bzw. ein als Ringpuffer ausgeführtes Array oder dergleichen) derjenigen bisher ausgeführten Operationen eingeführt, die etwas an den Adress-Zuordnungen ändern. Dadurch geht die Durchführung von `move`, `clear` und `delete` rasend schnell, weil nur ein Eintrag in die Liste gemacht werden muss, so dass sich im Endeffekt ein *Log* der durchzuführenden Operationen ergibt (falls im Log auch alle IO-Operationen aufgezeichnet werden, entsteht ein ähnlicher Effekt wie in Log-strukturierten Dateisystemen, vgl. [RO91]). Die Übersetzung einer Adresse kann nun grundsätzlich dadurch erfolgen, dass in dieser Log-Liste nachgesehen wird, ob irgendwelche Operationen eingetragen wurden, die etwas an einer gegebenen Adresse ändern; diese Änderungen werden dann in der richtigen Reihenfolge quasi virtuell nachvollzogen. Dieses einfache Verfahren führt natürlich nur dann zu akzeptabler Performanz, wenn die Liste möglichst kurz gehalten wird. Eine Möglichkeit zur Kürzung der Liste besteht darin, dass solche Operationen, die sich gegenseitig ganz oder teilweise aufheben bzw. die sich durch eine einzige Ersatz-Operation mit gleicher Semantik ersetzen lassen, aus der Liste entfernt und ggf. durch die Ersatz-Operation ersetzt werden (Prinzip der *Kompensation* von Operationen). Eine andere, in der Praxis auf Dauer unumgängliche Möglichkeit besteht darin, dass im Hintergrund Verschiebungen und Umordnungen durch Background-IO statt finden, durch die Einträge in der Liste überflüssig werden[12].

Zu erwähnen ist weiterhin, dass ein derartiges `map_simple_delta` auch kleinere `transfer_size`-Werte bis hinunter zu 1 unterstützen kann. Der Ausgang unterstützt automatisch die gleiche Granularität, wie sie vom Eingang vorgegeben wird, allerdings kann es vorkommen, dass eine `get`-Anforderung mit einem hohen Vielfachen der `transfer_size` nur mit einem geringeren Vielfachen möglich ist[13].

Ein `map_*`-Baustein muss auch die Operation `get_meta` sowie die Operationen auf dem Meta-Nest implementieren. Ein Meta-Nest sollte im Regelfall eine `transfer_size` von 1 unterstützen, was sich auch durch interne Verwendung eines `adaptor_*`-Bausteins (Abschnitt 4.1.8) erzielen lässt.

Wichtig ist ferner, dass `map_*`-Bausteine unbedingt die Eigenschaft der *Absturzfestigkeit* besitzen sollten. Mit diesem Begriff soll umschrieben werden, dass eine jederzeitige Trennung des Bausteins von seinem Eingang bzw. ein Verlorengehen von `transfer`-Operationen (z.B. bei einem plötzlichen Stromausfall, der zum Verlust der vorgeschalteten flüchtigen `buffer`-Informationen führt) nicht die Integrität der Adressübersetzung zerstören darf. Wenn man nicht auf die Performanz-Vorteile von Pufferung in flüchtigem Speicher verzichten will, muss man damit leben, dass der Zustand auf dem persistenten Hintergrundmedium ständig dem Zustand im flüchtigen Speicher hinterherhinkt, so dass bei unerwartet auftretenden Störungen Daten und damit Informationen verloren gehen. Bekannte Lösungen dieses Problems[14] stellen Log-basierte Schreibverfahren dar; entweder in Form separater Logs oder in Form so genannter Log-strukturierter Speicher. Zur Unterstützung dieser Verfahren wurde in Abschnitt 3.3.1 der Parameter `depend` eingeführt, mit dem sich die Reihenfolge von Schreiboperationen teilweise vorgeben lässt. Absturzfestigkeit lässt sich fernerhin als Teil von Transaktionen implementieren, falls man diese durch ein `multiversion`-Modell (Kapitel 8) in einer `map_*`-Variante implementiert. Eine weitere Möglichkeit stellt die Vorschaltung eines eigenen `powersafe`-Bausteins analog zu [dJ93] dar, der in isolierter Weise nur die Absturzfestigkeit als Strategie implementiert und keine Erweiterung statischer Nester auf dynamische vornimmt.

Die Funktionalität von `map_*` lässt sich sauch mit Dutzenden weiterer Verfahren zur Adress-Übersetzung herstellen, die eventuell Vorteile bei bestimmten Last-Charakteristiken bringen können. Zu nennen sind hier bei-

[10] Diese Erwartungshaltung ist bei genauer Betrachtung letztlich nur durch die Tatsache gerechtfertigt, dass die heutige Externspeicher-Technik, vor allem diejenige von Festplatten, schlechte Lokalitätseigenschaften aufgrund der mechanisch bewegten Teile besitzt (siehe auch die in der Literatur ausführlich diskutierte so genannte „Speicherlücke“). In absehbarer Zukunft werden hoch-kapazitive Externspeicher mit nicht nur deutlich höheren Datentransferraten und geringeren Latenzzeiten, sondern auch besseren Lokalitätseigenschaften zu konkurrenzfähigen Preisen verfügbar sein, beispielsweise holographische Speicher oder auf Magnet- oder Quanteneffekten beruhende hoch-kapazitive Halbleiterspeicher. Daher könnte die Diskussion zur Behebung des Fragmentierungs-Problems nur begrenzte Bedeutung haben.

[11] Ich erwarte, dass die Anwendung einfacher Verfahren wie die beständig angestrebte Herstellung einer Identitäts-Abbildung oder „Beinahe-Identitäts-Abbildung“ sehr gute Resultate liefern wird, *sofern dies im Hintergrund* mittels Background-IO geschieht.

[12] Unabhängig davon sollte `map_simple_delta` nur bei solchen Last-Charakteristiken angewandt werden, bei denen nur relativ selten Verschiebe-Operationen angefordert werden (was bei vielen Anwendungen durchaus erwartet werden kann), oder aber bei denen umgekehrt extrem häufige Verschiebungen, dafür aber nur relativ wenige Adress-Zugriffe vorkommen, so dass der Adressübersetzungs-Aufwand bei den amortisierten Kosten nur wenig ins Gewicht fällt.

[13] Dies ist in der Schnittstelle für Nester ausdrücklich so vorgesehen, und es ist die Aufgabe eines jeden Konsumenten, mit diesem Fall umgehen zu können (wobei er sich das Problem auch durch Vorschalten eines `adaptor_*` vom Hals schaffen kann).

[14] Eine weitere Lösung, nämlich die Konsistenzprüfung nach Abstürzen, halte ich wegen ihrer schlechten Reparatur-Eigenschaften und vor allem wegen ihrer Komplexität für unpraktikabel (als Standard-Mittel zur Behebung der Effekte von Abstürzen; in absoluten Notsituationen sieht es dagegen anders aus). Die Komplexität einer Konsistenzprüfung kann wegen der möglichen Zersplitterung eines Nestes in sehr viele kleine Flicken i.A. sehr groß werden.

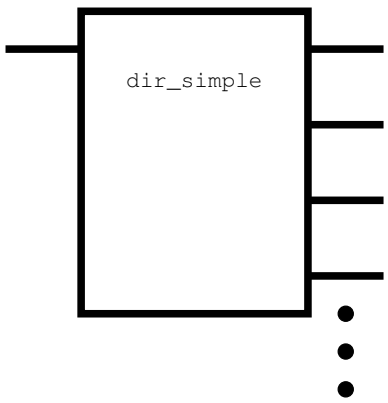
spielsweise Tries, balancierte Bäume, B-Bäume, Fuzzy Hashing [Sch99], und andere.

4.1.4. `selector`

Dieser Anpassungs-Baustein schneidet einen zusammenhängenden Teil-Adressraum aus seinem Eingangs-Adressraum aus und stellt ihn unverändert am Ausgang zur Verfügung, wobei die Adressen standardmäßig wieder bei 0 neu beginnen. Wird der ausgeschnittene Adressraum als Ganzes verschoben, dann wirkt sich das nicht auf den Gastgeber-Adressraum aus, ebenso umgekehrt bei Überstreichung einer `move`-Operation im Gastgeber-Adressraum über den gesamten ausgeschnittenen Bereich (Transparenz).

Die Implementierung ist relativ einfach, da außer einer uniformen Adressübersetzung und -Überprüfung nichts zu machen ist und alle Operationen vom Ausgang an den Eingang durchgereicht werden können. Es sind zwei grundlegende Varianten von Selektoren möglich, `selector_persistent` und `selector_tmp`, die sich darin unterscheiden, ob die Ausschnitts-Adressen und -Längen sowie deren Änderungen persistent festgehalten werden sollen oder nicht. Im ersteren Fall wird Hilfsinformation im Meta-Nest des Eingangs gehalten, mit der die Startadresse und die Länge relativ zum Gast-Adressraum auf Dauer festgehalten wird, so dass eine Destruktion des `selector` mit anschließender Reinstantiierung wieder den alten Zustand ergibt.

4.1.5. `dir_*`



Ein Baustein dieser Art benutzt das Eingangs-Nest als eine Art Sammel-Container, um mehrere von einander unabhängige Ausgangs-Nester daraus zu bilden. Die Ausgangs-Nester werden bei Bedarf durch die baustein-spezifische Zugriffsoperation `create` neu erstellt (die als Nest-Instantiierungs-Operation betrachtet werden kann); bereits früher erstellte werden durch `lookup` (oder ebenfalls durch `create`) erneut instantiiert, sofern sie nicht bereits instantiiert sind. Unterschieden werden die verschiedenen möglichen Ausgangs-Instanzen durch einen Suchschlüssel, der als ein zusammenhängender Datenblock übergeben wird. Zur Abfrage aller vorhandenen persistent gespeicherten Schlüsselwerte dient ein spezieller Ausgang, der Verzeichnis-Nest genannt wird, der im Regelfall eine `transfer_size` von 1 unterstützt, und von dem nur gelesen werden darf. Ein Verzeichnis-Nest stellt ein Nest mit lückenlos liegenden Datenpaketen dar, wobei jedes Paket genau einen der vorhandenen Suchschlüssel enthält. Die Schlüsselwerte können, müssen aber nicht nach irgend einem internen Kriterium sortiert gehalten werden. Löschungen von Schlüsselwerten und der zugehörigen Ausgangs-Nest-Instanzen lassen sich als baustein-spezifische Operation und/oder durch eine `delete`-Operation auf dem Verzeichnis-Nest realisieren. Hierbei ist die aus Unix bekannte Methodik zu bevorzugen, dass zwischen statischen und dynamischen Referenzen unterschieden werden sollte, und dass ein Ausgang und sein Speicherplatz erst dann tatsächlich freigegeben werden sollen, wenn keine dynamischen Referenzen (Drähte) mehr vorhanden sind.

Aus dieser Beschreibung dürfte klar geworden sein, dass damit ein Verzeichnis realisierbar ist, wie es in konventionellen Betriebssystem-Architekturen von *Dateisystemen* zur Verfügung gestellt wird. Ein `dir_*` stellt jedoch keine Verzeichnisbaum-Hierarchie zur Verfügung, sondern ähnelt eher dem flachen Index einer Datenbank. Dennoch lassen sich Verzeichnisbaum-Hierarchien sehr leicht herstellen: am Ausgang einer `dir_*`-Instanz braucht lediglich eine weitere `dir_*`-Instanz angeschlossen zu werden und so weiter. Auf diese Weise kann der Dateisystem-Baum (bzw. ein momentan instantiierter Teilbaum davon) direkt als Baumstruktur von Baustein-Instanzen mit der zugehörigen Verdrahtung dargestellt werden. Wenn man die in Abschnitt 4.2 vorgestellte Auto-Instantiierung von Bausteinen benutzt, dann braucht man sich als Benutzer nicht um die dynamische Herstellung dieser Baumstruktur zu kümmern.

Im Vergleich zur Funktionalität klassischer Dateisysteme ermöglicht dieses Konzept eine wesentlich flexiblere Instantiierung und Verdrahtung, da man beispielsweise

- für jedes Verzeichnis individuell verschiedene Baustein-Typen einsetzen kann, beispielsweise `dir_simple` für kleine Verzeichnisse, `dir_hash` für solche mit besonders schneller `lookup`-Funktionalität, oder `dir_btree` für solche mit besonders guten Lokalitätseigenschaften trotz riesiger Ausdehnung
- Datenkomprimierungs-, Datenverschlüsselungs- und sonstige Anpassungs-Bausteine wie `adaptor_*` und andere auf beliebigen Hierarchieebenen (automatisch) dazwischen schalten kann
- auf Konzepte wie feste Mounts und feste Mount-Tabellen verzichten kann, da dies von einem `redirect`-Baustein (vgl. Abschnitt 4.1.9) oder von der Strategie-Ebene (vgl. Abschnitt 4.2.2) übernommen werden kann
- ebenso auf Konzepte wie Loopback-Devices verzichten kann, indem lediglich ein `map_*` dazwischen geschaltet wird
- nahtlose Integration mit der Funktionalität von Datenbanken möglich ist: dazu zählt nicht nur der später vorgestellte `transaction`-Baustein, sondern auch die Möglichkeit, spezialisierte Bausteine wie beispielsweise `dir_fixed_keysize` einzusetzen, bei de-

nen eine Uniformität der Schlüssellängen zugunsten besserer Platzausnutzung erzwungen wird

- eine *virtuelle* Herstellung von Verzeichnisinhalten durchführen kann, beispielsweise mit einem Baustein `dir_proc` für die Funktionalität von `/proc`-Dateisystemen, oder `dir_join` zur Herstellung der Funktionalität der aus der Datenbank-Welt bekannten Join-Operation aus dem Daten-Inhalt mehrerer anderer Nester. In diesem Fall können `dir_*`-Varianten entstehen, die gar keinen oder mehrere Eingänge besitzen, was sich u.U. mit dem althergebrachten Konzept eines Dateisystem-Baums nicht auf intuitive Weise modellieren lässt.

Die grundlegende Funktionsweise von `dir_*`-Bausteinen möchte ich am Beispiel einer Realisierung von `dir_simple` erläutern. Es werden insgesamt drei verschiedene Regionen von zusammen hängenden Adressbereichen benutzt, um die Container-Funktionalität zu realisieren. Dies sind

- die Index-Region,
- die Meta-Region,
- die Daten-Region.

Diese Regionen werden innerhalb des Eingangs-Nestes und des Eingangs-Meta-Nestes angelegt, wobei die Daten-Region auf jeden Fall im Eingangs-Nest, die Meta-Region auf jeden Fall im Eingangs-Meta-Nest, und die Index-Region wahlweise in einem der beiden Nester liegen kann (sinnvollerweise sollte die Zuordnung von der Größe des Index abhängig gemacht werden).

`dir_simple` betrachtet diese Regionen jeweils als kompakt zusammen hängende Teil-Nester und benutzt die am Eingang zur Verfügung stehenden `move`-Operationen, um Einfügungen und Löschungen von Index-Werten durchzuführen, sowie die an den Ausgängen ankommenden `move`-Operationen an den Eingang weiter zu reichen. Dies bedeutet für die `get_maxlen`-Werte der Ausgänge, dass ihre Summe kleiner-gleich des `get_maxlen`-Wertes des Eingangs sein muss[15]. Die interne Realisierung von `dir_simple` braucht sich nicht um die Bereitstellung und Verschiebung von Platz zu kümmern, da diese Funktionalität bereits am Eingang verfügbar ist. Um die Funktionalität eines instantiierten Ausgangs zu erfüllen (mit Ausnahme von `set_maxlen`, das abgefangen werden muss), können interne Instanzen von `selector` verwendet werden.

Die Reihenfolge von Einträgen ist bei `dir_simple` in allen Regionen gleich: wenn beispielsweise ein Eintrag in der Index-Region ganz am Anfang bei der ersten Position eingefügt wird, dann wird in der Meta-Region und in der Daten-Region ebenso verfahren und dort jeweils Platz im Adressraum geschaffen. In diesem Fall, oder wenn sich beispielsweise die Größe eines Ausgangs-Nestes durch `set_maxlen` so stark ändert, dass Überschneidungen drohen, wird der gesamte restliche Adressraum in einem Rutsch verschoben; wegen der Transparenz-Eigenschaft der intern verwendeten `selector`-Bausteine merken die bereits instantiierten Ausgänge nichts davon.

Der Index-Bereich wird bei `dir_simple` direkt im gleichen Format abgespeichert, wie sie der Verzeichnis-Nest-Ausgang verlangt. Andere `dir_*`-Arten können selbstverständlich davon abweichen.

Die Ausgänge eines `dir_*`-Bausteins sollten mindestens `multiuser`-Kompetenz bereitstellen. Hierfür gibt es mehrere Möglichkeiten: die interne Realisierung könnte z.B. nur durch `singleuser`-Verhalten erfolgen (was den Entwurf ein klein wenig vereinfacht) und die `multiuser`-Kompetenz an den Ausgängen durch eine `adaptor_*`-Instanz zur Verfügung stellen. Für die Einsatzgebiete von Netzwerk-Betriebssystemen ist jedoch durchgehendes `multiuser`-Verhalten vorteilhaft. Dies ist bei `dir_simple` relativ einfach zu erfüllen, da die oben beschriebene Realisierung *(pseudo-)statuslos* erfolgen kann. Bei (pseudo-)statuslosen Realisierungen ist die Hinzunahme von `multiuser`- oder `multiversion`-Verhalten einfach, da lediglich alle potentiell konfliktträchtigen Operationen durch `lock` / `unlock`-Paare zu klammern sind. Die einzige konfliktträchtige Operation bei den Ausgangs-Nestern ist `set_maxlen`, die bei vernünftigem Entwurf nur selten aufgerufen werden sollte; alle anderen Operationen außer Modifikationen in der Index-Region brauchen keine zusätzlichen Klammern und laufen unverändert oder nur wenig verändert durch.

Diese recht einfache Realisierung hat sehr gute Lokalitätseigenschaften: das Lokalitäts-Verhalten eines Ausgangs wird direkt an den Container beim Eingang weitergereicht. Falls kleine Verzeichnisse so realisiert sind, dass die Index-Region im Meta-Nest des Eingangs liegt, dann enthält das Meta-Nest im Falle von rekursiv verschachtelten Verzeichnissen an der Wurzel die gesamte Baumstruktur aller Indizes, die im Vergleich zur Summe aller Dateigrößen um einige Größenordnungen kleiner ausfällt (zumindest bei üblichen durch menschliche Nutzer verursachten Dateigrößen) und allein wegen dieser geringen Lokalität der Ausdehnung einen schnellen Zugriff erlaubt (wobei das Caching des transitiv vorgeschalteten `buffer`-Bausteins separate Konzepte wie Inode- oder Namens-Caches überflüssig macht).

Die Zuordnung der Index-Region zu einem der beiden Eingangs-Nester ist eine Strategie-Entscheidung, die in jedem Einzelfall anders getroffen werden kann; sehr große Index-Regionen lassen sich auch in das reguläre Eingangs-Nest verlagern, so dass die Lokalität des Meta-Nestes möglichst wenig verschlechtert wird.

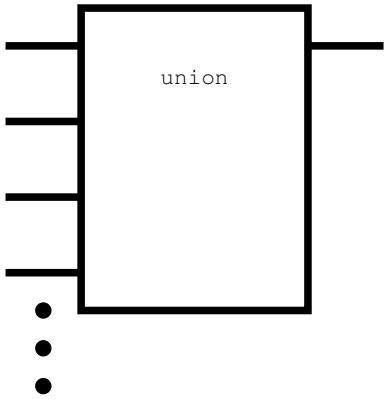
Eine weitere strategische Entscheidungsmöglichkeit für das Lokalitäts-Verhalten besteht darin, einige oder alle der Regionen miteinander zu verschmelzen. Dies kann insbesondere bei der Verquickung der Index- mit der Meta-Region Vorteile bringen, sofern die Meta-Daten der Ausgänge nur wenig Platz beanspruchen. Der von Hans Reiser in `reiserfs` [Rei97] propagierte effiziente Zugriff auf sehr kleine Datei-Größen (wie z.B. bei der Modellierung einzelner Felder von Datensätzen einer Datenbank mit Hilfe von „Mini-Dateien“) lässt sich ggf. durch eine Verschmel-

[15] Konzeptionell entspricht diese Einschränkung dem auch bei konventionellen Dateisystemen vorkommenden Zwang, dass die Summe des Platzbedarfs aller abgespeicherten Dateien nicht die Gesamtgröße des Dateisystems übersteigen darf. Falls in einer `dir_*`-Hierarchie sehr große dünn-besetzte Nester vorkommen, deren Nutzungsgrad eine heuristisch bestimmte Schranke unterschreitet (was bei häufigem Vorkommen zu dem Problem führen kann, dass ein Adressraum mit 64 Bit Breite nicht mehr für die Summe aller vorkommenden Adressraum-Größen ausreicht), dann kann eine Verkleinerung der virtuellen Adressraum-Größen durch einen zwischengeschalteten `map_*`-Baustein erreicht werden.

zung von Index-und Daten-Region erreichen, bzw durch eine fallweise dynamische heuristische Zuordnung zu einer der Regionen[16].

Durch diese Beispiele sollte klar geworden sein, dass spezielle Anforderungen, die bisher oftmals als Anreiz zur Entwicklung aufwendiger und umfangreicher eigenständiger Dateisysteme mit trickreichen internen Implementierungen dienten, durch ein Baustein-Konzept mit orthogonalen Kombinationsmöglichkeiten mindestens ebenso gut abgedeckt werden können. Die hier vorgestellte Baustein-Zerlegung verteilt die in konventionellen Dateisystemen vorkommenden Problematiken und Funktionalitäten auf mehrere Baustein-Arten, isoliert sie voneinander, und ermöglicht vorher unbekannte Kombinationen. Darüber hinaus sind Bausteine intern deutlich einfacher implementierbar als bei stapelbaren Dateisystemen (siehe [HP94, HP95]), da nicht mehr zwischen verschiedenen Ebenen wie „Dateien" versus „Dateisystem-Bäume" unterschieden wird.

4.1.6. `union`

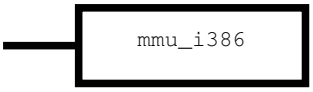


Dieser Baustein stellt in gewisser Hinsicht eine inverse Operation zu `selector` bzw `dir_*` dar: mehrere Eingangs-Nester erscheinen in einem Ausgangs-Nest und werden dabei adressmäßig nebeneinander aufgereiht. Über Parameter bzw. Meta-Informationen wird festgelegt, ob sich ein Eingangs-Adressraum lückenlos an seinen Vorgänger anschließen soll (so dass eventuelle Lücken am Ende des Vorgängers und am Anfang des eigenen Nestes zusammengeschoben werden und `move`-Operationen des Vorgängers mit vollzogen werden), ob der Anschluss über `get_maxlen` erfolgen soll, oder ob er ggf. unter Lückenbildung an festen Adressen „fest genagelt" erscheinen soll (analog zu `selector`). Weitere Spielarten sind denkbar.

Über Kontruktoren bzw. Destruktoren (die als generische Operationen ausgeführt sein können, vgl. Kapitel 6) lassen sich Eingänge zur Laufzeit hinzufügen und entfernen.

Ein wichtiges Anwendungsgebiet von `union` ist die Zusammenstellung verschiedener Regionen bzw. „Segmente" in virtuellen Adressräumen oder Schutzbereichen, beispielsweise die Einteilung in Code-, Stack- und Datensegmente, sowie in Mappings anderer Nester. Verschiedene Mapping-Arten lassen sich durch Vorschalten von Anpassungsbausteinen wie `cow` (Abschnitt 4.1.10) realisieren.

4.1.7. `mmu_*`



Dieser Baustein besitzt genau einen Eingang und keinen Ausgang[17], ist also im Sinne der Verdrahtungslogik ein reiner Konsument. Er stellt die Schnittstelle zur Memory-Management-Unit (MMU) der Rechner-Hardware dar; man könnte ihn als „Geträtetreiber für die MMU-Hardware" charakterisieren. Das Eingangs-Nest wird hierbei 1:1[18] in einen virtuellen Adressraum umgewandelt, der sich über Kontrollflüsse direkt durch Maschinenbefehle des Prozessors adressieren lässt.

Die Realisierung dieses Bausteins ist im Vergleich zu konventionellen monolithischen Implementierungen etwas einfacher: wenn ein Lese-Seitenfehler auftritt, wird der betroffene Datenblock mittels `get` und nachfolgendem `transfer` im `read`-Modus vom Eingangs-Nest angefordert und in die Seiten-Tabellen[19] der MMU eingetragen. Bei Schreib-Seitenfehlern wird der write-Modus verwendet; eventuell wird ein bereits im Lese-Modus verwendeter Datenblock in den Schreib-Modus aufgewertet. Beim Austragen wird `put`, im Falle geänderter Seiten davor auch noch `transfer` im `write`-Modus mit geringer Hintergrund-IO-Priorität aufgerufen.

Über die `notify_*`-Schnittstelle (siehe Kapitel 5) erhält `mmu_*` Kenntnis von physischen Datenblöcken, die von der Speicherverwaltung zur Freigabe vorgesehen wurden, sowie von invalide gewordenen Adressbereichen (z.B. durch `move`-Operationen). Diese werden ebenfalls aus dem virtuellen Adressraum ausgetragen[20]. Hierauf können sie evtl. beim nächsten Seitenfehler gleich wieder angefordert werden, was aber relativ geringe Verzögerungen zur Folge hat, *sofern* sich die Seite in Wirklichkeit noch im LRU-Cache der transitiv vorgeschalteten `buffer`-Instanz befindet. Solche Rückforderungen können daher *auf Verdacht*

[16] Ferner kann man bei der Repräsentation der Felder von Datensätzen einer Datenbank einen funktionalitätsmäßig stark eingeschränkten Baustein-Typ `dir_record` einsetzen, der die gleichen Index-Strukturen von Records gleichen Aufbaus und die ebenfalls immer gleichen Metadaten (in denen z.B. die Feldnamen und Feldlängen und weitere Attribute gespeichert werden) nicht in der Index-Region jeder einzelner Instanz abspeichert, sondern per Referenz aus einer gemeinsamen externen Quelle bezieht und damit nur einen vernachlässigbaren Platz-Overhead bei jeder einzelnen Instanz verursacht.

[17] In Wirklichkeit hat er auch einen Ausgang, nämlich den virtuellen Benutzer-Adressraum. Dieser hat jedoch aus Hardwaregründen nicht mehr dieselbe Schnittstelle mit den hier vorgestellten Grundoperationen. Im abstrakten Sinne führt ein Prozessor-MMU-Gespann dennoch die gleichen Grundoperationen aus: Hauptspeicher-Zugriffe des Prozessors lassen sich als Kombinationen von `get-` und `put-` mit `transfer`-Operationen modellieren; eine MMU betreibt abstrakt gesehen eine eingeschränkte Art der Virtualisierung analog zu einem statischen Nest.

[18] Falls der logische Adressraum eines Nestes größer als der virtuelle Adressraum eines Prozesses ist (z.B. bei 32Bit-Hardware), dann wird nur ein Teil des logischen Nest-Adressraums für den direkten Zugriff durch Maschinenbefehle umgesetzt.

[19] Üblicherweise benötigt MMU-Hardware einen Hilfsspeicher für die Seitentabellen. Zur Vereinfachung ist dieser hier nicht als eigener Eingang eingezeichnet, da eine andere Verdrahtung als mit `device_mem` aus Hardwaregründen meist nicht möglich ist.

[20] Trotz der Notwendigkeit, Hilfsspeicher für die Seitentabellen einzusetzen, arbeitet eine `mmu_*`-Instanz logisch gesehen *statuslos*: man kann jederzeit die gesamte Seitentabelle freigeben, ohne dass ein Schaden (außer evtl. Performanz-Verschlechterung) eintritt, da die danach unweigerlich auftretenden Seitenfehler den Status der Tabelle wieder aus dem Eingangs-Nest soweit notwendig rekonstruieren werden.

und jederzeit von der Speicherverwaltung an zentraler Stelle[21] ausgelöst werden, ohne die Aktivitäten eines Benutzerprozesses merklich zu stören. Macht man dies auf systematische Weise für alle vorhandenen virtuellen Adressräume, entsteht eine fortlaufende Altersbestimmung der Working-Sets (vgl. [Den68, Den71]) dieser Adressräume, was dem Effekt des bekannten Second-Chance-Algorithmus zur Seitenersetzung ähnelt[22]. Als Nebeneffekt werden geänderte Seiten frühzeitig spekulativ mit Hintergrund-Priorität ausgelagert[23], so dass bei einer später tatsächlich eintretenden Speicherknappheit eine Chance besteht, dass die Seite inzwischen *nicht wieder* modifiziert wurde und daher sofort recycelt werden kann, ohne auf die Beendigung von IO warten zu müssen.

In einem `mmu_*`-Baustein braucht keine wie auch immer geartete spezielle Paging-Strategie mit verschiedenen Modi und Abhängigkeiten von Mapping-Arten implementiert zu werden, was bei konventionellen Implementierungen den Löwenanteil an Komplexität ausmacht. Diese Aufgaben einschließlich der Persistenthaltung privater Mapping-Segmente werden von vorgeschalteten Bausteinen übernommen[24].

Damit ein `mmu_*`-Baustein am Ende einer Baustein-Hierarchie stehen darf, muss die übliche Trennung in Spinlocks und Scheduler-aufrufende Locks aufgehoben werden (siehe [K$^+$91, LA93] sowie Abschnitt 3.3.5). Dies ist in SMP-Rechnern (Symmetric MultiProcessing) notwendig, um mehrere `mmu_*`-Instanzen auf verschiedene Prozessoren verteilen zu können oder mehrere auf verschiedene Prozessoren verteilte Kontrollflüsse auf der gleichen `mmu_*`-Instanz laufen zu lassen. Weiterhin ist erforderlich, dass Seitenfehler-Unterbrechungen die Nest-Operationen aufrufen dürfen (siehe [KE95] sowie Abschnitt 3.3.5).

Eine weitere Aufgabe von `mmu_*` ist die Realisierung von Schutzbereichen. Dazu ist eine Verknüpfung mit der Kontrollfluss-Implementierung (vgl. Abschnitt 4.3.1) nötig, die auch ausserhalb der regulären Baustein-Verdrahtung gelöst werden kann, da sie davon unabhängig ist (dies gilt ebenfalls für die Anbindung an die Unterbrechungen; vgl. Abschnitt 4.3.3)[25]. Verschiedene Schutzbereiche lassen sich am einfachsten durch Zuordnen verschiedener Mandate (vgl. Abschnitt 2.7) und unterschiedlicher Behandlung in untergeordneten `check_*`-Instanzen implementieren. Damit werden `mmu_*`-Instanzen zu Verwaltern derjenigen Mandate, die mit den Schutzbereichen zu tun haben. Die Mandats-Verwaltung kann auch an interne oder untergeordnete Bausteine delegiert werden.

4.1.8. `adaptor_*`

Es handelt sich um einen Anpassungs-Baustein, im Regelfall mit jeweils einem Eingang und einem Ausgang, der zwischen Nestern mit verschiedenen Kompetenzen und Verhalten wie beispielsweise verschiedenen `transfer_size`-Attributwerten vermittelt und übersetzt.

4.1.8.1. Adaption zwischen verschiedenen Zugriffs-Modellen

Wie der Tabelle 3.1 auf Seite 32 zu entnehmen ist, gibt es folgende Fälle, in denen kein Anschluß von Eingängen an die Ausgänge anderer Bausteine möglich ist:

1. Der Ausgang hat zu geringe Kompetenzen, um die Anforderungen durch das Verhalten der Eingänge abdecken zu können.

2. Ein Eingang unterstützt nur `singleuser`-Verhalten und verträgt sich daher nicht mit anderen Eingängen.

Für jeden dieser Fälle kann man eine geeignete `adaptor`-Art einführen. Fall 1 lässt sich beispielsweise durch `adaptor_multi` lösen, der am Eingang nicht vorhandene Lock-Operationen nachimplementiert. Zur Herstellung der `multiuser`-Kompetenz brauchen die restlichen Operationen nur durchgereicht zu werden. Aufwendiger ist die Herstellung der `multiversion`-Kompetenz (vgl. Kapitel

[21] Nur diese Stelle hat die volle Übersicht über die momentane Speichersituation und kann einigermaßen verlässlich einschätzen, wie hoch zukünftige Speicher-Anforderungsraten ausfallen könnten oder ob die Situation unkritisch ist. Die Rückforderungen auf Verdacht lassen sich damit den aktuellen Lastverhältnissen anpassen, so dass unnötiger Overhead minimiert wird.

[22] MMU-Hardware besitzt oft Referenz- und Modifikations-Bits (auch Dirty-Bit genannt), die im oben beschriebenen Verfahren nicht genutzt werden. Eine Benutzung dieser Bits kann dadurch erfolgen, dass auf `try`-Rückforderungen bei gesetzten Bits nicht reagiert wird, sondern lediglich das Bit gelöscht wird. Die untergeordnete Anforderungs-Instanz kann aus der nicht durchgeführten Rückgabe schließen, dass die Seite im Moment noch gebraucht wird, was äquivalent zu einer Rückgabe mit sofortiger erneuten Anforderung ist. Dadurch wird unnötiger Datenverkehr eingespart.

[23] Im Vergleich zu konventionellen Strategien, die mit der Auslagerung oftmals erst bei bereits eingesetzter Speicherknappheit beginnen, führt dies zu potentiell erhöhtem IO-Verkehr. Da dieser jedoch mit Hintergrund-Priorität abgewickelt wird, stört er laufende Aktivitäten so gut wie gar nicht.

[24] Ein konventionelles Paging auf Hintergrundspeicher stellt bei der hier vorgestellten Architektur kein eigenständiges Konzept dar. Private Segmente werden werden hier durch temporäre Nester emuliert, die auf Massenspeicher gehalten werden. Im Falle von Stack-Segmenten sind sie am Anfang (beinahe) leer; bei wachsender Größe wird neuer Platz mittels `clear` nachgefordert; die Auslagerung ihres Inhaltes kann spekulativ durch Background-IO erfolgen.

[25] Mitte der 1990er Jahre wurden im Fachgebiet der Betriebssysteme besonders viele Diskussionen um Kern-Größen und um Grenzen zwischen Kern und „Benutzerbereich“ geführt. Ich vermeide den Begriff eines Betriebssystem-Kerns, da er oft mit monolithischen Strukturen assoziiert wird. In der hier vorgestellten Architektur kann es *mehrere* Kerne im konventionellen Sinn geben: beispielsweise kann die Anbindung an die MMU-Hardware getrennt von der Kontrollfluss-Verwaltung erfolgen, ggf. auch in unterschiedlichen Schutzbereichen erfolgen. Wenn von einem bestimmten Hardware-Mechanismus in einem System nur ein einziges Exemplar vorhanden ist (beispielsweise eine gemeinsame Unterbrechungs-Sprungtabelle in einem Multiprozessor-System), dann ergibt sich für die zugehörige Verwaltungs-Software auf natürliche Weise nur eine einzige Verwaltungs-Instanz. Bei prinzipiell voneinander unabhängigen Hardware-Einheiten kann es jedoch durchaus mehrere verschiedene Verwaltungs-Instanzen geben, die man in konventioneller Teminologie mehrere voneinander unabhängige Mikro-Kerne nennen könnte. Eine derartige Modularisierung bringt insbesondere bei der fortlaufenden Anpassung an geänderte oder erweiterte Hardware-Komponenten Vorteile. Diese Problematik tritt insbesondere durch die fortschreitende technische Entwicklung bei Gerätetreibern auf und wird bei vielen Minimalisierungs-Ansätzen von Kernen wie z.B. der Exokern-Architektur [EKO95] nur unzureichend gelöst.

8), da in diesem Modus von jedem physischen Datenblock potentiell mehrere Versionen verwaltet werden müssen.

Zur Lösung des Falls 2 kann man einen Baustein `adaptor_synchronize` einführen. Dieser kann einen Eingang mit `multiuser`- oder `multiversion`-Verhalten voraussetzen, da diese Funktionalität notfalls durch `adaptor_multi` bereitgestellt werden kann. Da an jedem Ausgang immer nur ein einziger Konsument mit `singleuser`-Verhalten angeschlossen werden darf, muss `adaptor_synchronize` entsprechend viele Ausgänge haben, die ggf. zur Laufzeit nachträglich instantiiert oder gelöscht werden müssen.

Die Problematik von `adaptor_synchronize` besteht darin, dass jeder Ausgang die Illusion erhalten muss, er sei der einzige Konsument einer Nest-Instanz, der Änderungen durchführt. Eine solche Illusion wird auch von klassischen Datenbank-Transaktionen erzeugt. Zur Herstellung dieser Illusion ist daher prinzipiell auch der Baustein `transaction` (Abschnitt 4.1.11) geeignet. Aus der Transaktionstheorie ist bekannt, dass eine vollständige Isolation der Teilnehmer bei beliebigen, *nicht vorhersagbaren* und *echt parallelen* Aktionen verschiedener `singleuser`-Teilnehmer nicht möglich ist, ohne die Gefahr von Deadlocks oder Rollbacks in Kauf zu nehmen. Daher schlage ich eine Aufteilung der Funktionalität vor: echte Parallelität soll durch `transaction` ermöglicht werden; eine eingeschränkte, aber leichter zu implementierende Form der Illusion von `singleuser`-Verhalten wird durch `adaptor_synchronize` hergestellt. Dies geschieht folgendermaßen:

Wir nehmen an, dass die Konsumenten an den Ausgängen *statuslos* oder *pseudo-statuslos* arbeiten, d.h. sie fordern Datenblöcke mittels `get` nur für einen relativ begrenzten Zeitraum an und geben sie möglichst bald wieder mittels `put` frei. Bei völliger Statuslosigkeit kann man damit rechnen, dass jeder Konsument recht bald alle seine Blöcke von sich aus wieder freigeben wird; andernfalls ist jederzeit eine vollständige Rückforderung mittels `notify_*`-Operationen (siehe Kapitel 5) möglich. Unter dieser Voraussetzung lässt sich eine einfache Strategie durch Locks implementieren, indem zu einem Zeitpunkt jeweils nur ein einziger Ausgang Zugriff auf den Status des Eingangs erhält; die anderen müssen solange warten, bis dieser sämtlichen Status zurückgegeben hat. Sobald irgendwelche Anforderungen durch einen anderen Ausgang ankommen, während irgend ein Status bereits vergeben ist, dann versucht die `adaptor_synchronize`-Implementierung, den anderweitig vergebenen Status mittels `notify_*` wieder baldmöglichst zurück zu bekommen[26]. Dies schränkt die Parallelität zwar sehr stark ein, ist aber einfach zu implementieren, vermeidet Deadlocks im Kleinen und kommt ohne Rollback-Operationen aus, die bei echt parallelen Transaktionen und vorher nicht bekanntem Zugriffsverhalten prinzipiell nicht vermeidbar[27] sind.

Eine vernünftige Konsequenz aus diesem Dilemma ist meines Erachtens, dass man das `singleuser`-Programmiermodell vermeiden sollte und Konsumenten mit explizitem `multiuser`- oder `multiversion`-Verhalten[28] (Kapitel 8)[29] ausstatten sollte, wann immer es möglich ist (sofern man Wert auf Parallelität, Skalierbarkeit und Performanz legt). In diesem Sinne ist `adaptor_synchronize` nur als Notbehelf zu verstehen und einzusetzen.

4.1.8.2. Adaption zwischen verschiedenen `transfer_size`-Werten

Im Idealfall sollten sich Baustein-Implementierungen nicht um das Problem kümmern müssen, dass die `transfer_size` eines Ausgangs mit derjenigen eines Eingangs zusammenpassen muss. Diese Umwandlungs-Aufgabe sollte an `adaptor_*` delegiert werden können.

Im allgemeinen kann es vorkommen, dass die beteiligten Transfergrößen keine Teiler voneinander bilden. Dieser Fall taucht in der Praxis kaum auf, weil aus guten Gründen meist nur Zweierpotenzen[30] als feste Transfer-

[26] Dies zeigt eine gewisse Ähnlichkeit zu den *opportunistischen Locks*, die insbesondere von Microsoft eingesetzt werden [OpL]. Opportunistische Locks brauchen jedoch nicht unbedingt gewährt zu werden, und sie werden bei Bedarf *gebrochen*, d.h. sie werden mit *brutaler Gewalt* entzogen. Das in Abschnitt 5.3 propagierte Modell ermöglicht dies in Extremfällen zwar auch, verfolgt aber im Normalfall die Idee, dass ein *Konsens* über die Verteilung von Locks hergestellt werden sollte. Opportunistische Locks stellen hingegen ein *eigenständiges* Konzept dar, das neben den „normalen" Locks *zusätzlich* zur Performanz-Steigerung existiert. In dieser Hinsicht unterscheidet sich der hier vorgestellte Ansatz zu dem von Microsoft grundlegend.

[27] Ein Ansatz zur Vermeidung von Deadlocks in Datenbanken ist *konservatives Locking* (siehe z.B. [GR93, VGH93]), bei dem alle von einer Transaktion benutzten Ressourcen einmalig am Anfang atomar angefordert werden müssen (vgl. auch das „Handwerker-Problem" in [Jür73], das eine Verallgemeinerung des bekannten Philosophen-Problems [Lam74] darstellt); dies setzt jedoch entweder genaue a-priori-Kenntnisse über das zukünftige Verhalten voraus, oder es schränkt die Parallelität extrem stark durch unnötige Spekulationen auf später meistens doch nicht wirklich genutzte Ressourcen ein. Bei vorher nicht bekanntem Verhalten der Transaktionen führen inkrementell nach tatsächlichem Bedarf gesetzte 2-Phasen-Sperren zu einem Deadlock-Problem, das sich *a priori* nicht vermeiden läßt (die posteriori-Erkennung ist dagegen relativ einfach; damit werden jedoch unvorhersehbare Rollbacks in Kauf genommen). Deadlocks lassen sich zwar durch halbgeordnetes (zyklenfreies) Setzen von Locks vermeiden, doch auch dafür muss man das zukünftige Verhalten der Transaktionen a priori kennen, was in der Praxis nur selten gegeben sein dürfte. Wenn man Rollbacks wie bei einigen Betriebssystem-Anwendungen unbedingt vermeiden muss, und/oder wenn man das zukünftige Verhalten von Konsumenten nicht kennt, dann hilft im allgemeinen nur, das Scheduling durch Anfordern eventuell später doch nicht wirklich benötigter Locks einzuschränken, d.h. den möglichen Parallelitätsgrad zu senken. Daher ist es von immenser Wichtigkeit, solche Locks zu verwenden, die sich gegenseitig möglichst wenig stören. Das `multiversion`-Modell (Kapitel 8) kann hierzu ebenfalls einen Beitrag leisten.

[28] Deadlocks können nicht nur bei Transaktionen, sondern auch bei `multiuser`- oder `multiversion`-Verhalten von Anwendungen auftreten, die mit der Anwesenheit paralleler Aktivitäten umgehen können (z.B. `code_reentrant` aus Abschnitt 2.6). Ich sehe Deadlocks nicht als dem Transaktions-Paradigma inhärent, sondern der Parallelverarbeitung auf gemeinsamen Daten schlechthin. Im Unterschied zum klassischen Transaktions-Paradigma, das die Auswirkungen und Folgen von Parallelverarbeitung vor den Konsumenten zu verstecken sucht, macht das `singleuser`- und `multiuser`-Modell diese Problematik *explizit*. Dies stellt zwar höhere Anforderungen an die Programmierer, ermöglicht aber feiner gesteuerte Reaktionen auf Fälle von Deadlocks, z.B. indem ein Prozess ein Signal erhält oder zur Rückgabe eines einzelnen Locks gebeten oder gezwungen wird, ohne dass deswegen gleich der Totschlag-Hammer eines Gesamt-Rollbacks geschwungen werden muss.

[29] Ich halte Transaktionen nicht deswegen für sehr nützlich, weil sie eine `singleuser`-Illusion ermöglichen, sondern weil sie den Parallelitätsgrad im Sinne einer optimistischen Strategie steigern können und die Rollback-Funktionalität explizit anbieten; diese ist auch bei `multiuser`-Verwendung von Transaktionen sehr nützlich.

[30] Ausnahmen mit „krummen" Sektorgrößen kommen vor, z.B. ältere Platten-Geräte und einige moderne dazu kompatible Spezial-Festplatten von IBM, die zur Abspeicherung von ISAM-Informationen

größen benutzt werden. Falls er dennoch auftreten sollte[31], kann man sich auf folgende Weise behelfen: man bestimme den ggT (größter gemeinsamer Teiler) der beiden vorkommenden Transfergrößen. Dann schalte man zwei `adaptor_*`-Instanzen hintereinander, wovon die erste von der Eingangs-Transfergröße auf den ggT hinuntertransformiert, die zweite vom ggT wieder auf die Ausgangsgröße hochtransformiert. Im Folgenden beschränke ich mich daher auf die Annahme, dass eine der beiden Transfergrößen ein Vielfaches der anderen darstellt. Es sind zwei Fälle zu unterscheiden:

1. Hochtransformation von einer kleinen `transfer_size` am Eingang zu einer größeren am Ausgang
2. Hinuntertransformation von einer großen `transfer_size` am Eingang zu einer kleineren am Ausgang

Bei der Hochtransformation besteht das Problem, dass der Konsument an seinem Eingang und damit auch am Ausgang des `adaptor_*` erwartet, dass eine `get`-Operation einen physisch *zusammenhängenden* Datenblock mit mindestens seiner Transfergröße liefern wird. Da am Eingang des `adaptor_*` eine kleinere Transfergröße eingestellt ist, braucht der dort angeschlossene Produzent nicht unbedingt Datenblöcke mit dieser Größe und/oder nicht unbedingt an entsprechend ausgerichteten Adressgrenzen im physischen Hauptspeicher zu liefern (allerdings wird jedem Produzenten geraten, dies dennoch zu erfüllen, wenn es im Einzelfall ohne größere Kosten möglich ist).

Im allgemeinen ist es wünschenswert, dass der Eingang an `multiuser`- oder `multiversion`-Verhalten teilnehmen kann und damit weitere parallel arbeitende Konsumenten zulässt, die unabhängig voneinander Blöcke mittels `get` anfordern und halten können. In diesem Fall kommt ein Verschieben von physischen Blöcken im Hauptspeicher an „günstigere“ physische Adressen im allgemeinen nicht in Frage[32]. Wenn man den Eingang nicht im `singleuser`-Modus betreiben möchte, dann bleibt wohl oder übel nichts anderes übrig, als *physische Kopien* von Datenblöcken herzustellen.

Eine von einem `adaptor_*` hergestellte physische Kopie muss auch von diesem Baustein bezüglich der Referenzzähler, Speicherfreigabe usw. verwaltet werden; beim letzten `put` auf einer zum Schreiben freigegebenen Kopie müssen die Änderungen auf das Original bzw auf die kleineren Original-Teile zurück übertragen werden.

Bei genauerer Betrachtung des Problems fällt auf, dass die Herstellung von physischen Kopien eventuell die Chance birgt, die mögliche Parallelität von Operationen dadurch zu erhöhen, dass die verschiedenen Versionen ausdrücklich für ein `multiversion`-Modell am Ausgang genutzt werden, auch wenn der Eingang nur `multiuser`-Verhalten weiterreicht. Daher bietet es sich an, die Funktionalität von `adaptor_multi` gleich hier mit zu integrieren, bzw. `adaptor_multi` gleich mit der Hochtransformations-Fähigkeit auszustatten, so dass ein einziger Baustein-Typ beide Aufgaben erledigt.

Zur Hinuntertransformation: hier besteht das Problem der Transfergrößenunterschiede nicht, da eine kleinere teilbare Transfergröße die Bedingungen der größeren bereits von selbst erfüllt. Bei der Weitergabe physischer Adressen an den Ausgang ist lediglich zu beachten, dass diese *innerhalb* eines Blocks liegen können, der vom Eingang geliefert wurde. Bei der Rückgabe mittels `put` muss eine solche physische Adresse wieder auf die `transfer_size` des Eingangs normiert werden, was durch Divisionen und Multiplikationen, bei Zweierpotenzen auch durch Ausmaskieren von Adressbits erfolgen kann.

Die Hinuntertransformation hat jedoch mit einem Problem zu kämpfen, das bei der Hochtransformation nicht existiert, weil es dort aus Symmetriegründen bereits von selbst erfüllt ist: die `move`-, `clear`- und `delete`-Operationen können vom Konsumenten am Ausgang in kleineren Portionen angefordert werden, als sie vom Produzenten am Eingang zur Verfügung gestellt werden. Daher ist eine einfache Durchreichung dieser Operationen an den Produzenten im Allgemeinfall leider nicht möglich.

Zur Lösung dieses Problems könnte ein `adaptor_*` die `move`-Operation komplett neu implementieren, ohne davon Gebrauch zu machen, dass der Eingang bereits ein dynamisches Nest bereit stellt. Dies widerspräche jedoch dem Ziel, die Bausteine zur „möglichst orthogonalen“ Zerlegung der in Betriebssystemen vorkommenden Aufgaben einzusetzen. Was bedeutet jedoch „möglichst orthogonal“?

Für die Beurteilung der Orthogonalität und Homogenität eines Baustein-Verhaltens sehe ich als wichtiges Kriterium, ob eine Transformation wieder durch eine rückläufige Transformation aufgehoben werden kann, oder zumindest bis auf einen konstanten Rest aufgehoben werden kann (Prinzip der Kompensierbarkeit). Dies würde bedeuten, dass eine Hinuntertransformation mit nachgeschalteter Hochtransformation auf die ursprüngliche `transfer_size` nichts am Inhalt eines Nestes ändern sollte, also insgesamt eine idempotente Abbildung darstellen sollte.

Leider ist dies aus informationstheoretischen Gründen in vollkommener *Reinform* nicht möglich: Um die Aufteilung einer gegebenen festen Anzahl von Nutzbytes auf einen ebenfalls festen größeren Adressraum mit Löchern zu verwalten, muss irgendwo Platz für die Darstellung dieser Zustandsinformation aufgewandt werden. Wenn man den Nutzinhalt auf den größeren Adressraum irgendwie (z.B. zufällig in einer Art Schweizerkäse) verteilt, dann gibt es für die *Anzahl* solcher möglichen Verteilungen um so mehr Möglichkeiten, je kleiner die `transfer_size` gewählt wird (der Schweizerkäse darf bildlich gesprochen immer kleinere Löcher und auch mehr Löcher enthalten, obwohl er weder sein Gewicht noch sein Volumen ändert). Welche dieser vermehrten Möglichkeiten konkret vorliegt, muss irgendwo gemerkt und abgespeichert werden, was aus informationstheoretischen Gründen einen Platzbedarf zur Folge hat, der mit immer kleinerem `transfer_size` zumindest im

gedacht sind. Ferner gibt es in den CD-ROM-Standards einige krumme Sektorformate, auf die von aktuellen Betriebssystemen nur der rohe ungepufferte Zugriff unterstützt wird.

[31] Der hier vorgestellte Nest-Entwurf unterstützt ausdrücklich völlig beliebige Transfergrößen. Wer allerdings Primzahlen als Transfergröße benutzt, der ist selbst schuld und darf in der Folge keine besonders gute Performanz erwarten.

[32] Da es vermutlich doch relativ häufig vorkommt, dass im Einzelfall entweder nur ein einziger Konsument gerade die betroffenen Datenblöcke hält, oder dass eine Rückforderung mittels der später besprochenen `notify_*`-Operationen möglich ist, dürften Verschiebungen in einigen Fällen lohnend sein.

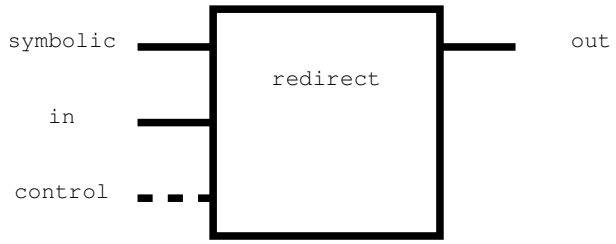
Worst Case ansteigt[33]. Hieraus folgt, dass es im allgemeinen nicht möglich ist, die `transfer_size` nach unten zu transformieren, ohne dass irgendwo zusätzlicher Platz für die Speicherung der nunmehr feineren Abbildungsmöglichkeiten für den Definitionsbereich des Nestes verbraucht wird.

Dies bedeutet jedoch nicht, dass zusätzlicher Platz in *allen* Fällen verbraucht werden muss. Man kann von einer Hinuntertransformation folgende wünschenswerte Eigenschaft verlangen:

Falls am Ausgang der Hinuntertransformation nur solche Operationen ausgeführt werden, die auch ohne die Hinuntertransformation stattfinden könnten, weil sie (zufälligerweise) die Bedingungen der größeren `transfer_size` am Eingang bereits erfüllen, dann sollte am Ausgang der Hinuntertransformation derselbe Nest-Zustand sichtbar sein, als wäre die Hinuntertransformation nicht vorhanden. Dies bedeutet u.a., dass in diesem Fall kein Platz für interne Verwaltungsinformationen abgezwackt werden darf (der Sachverhalt als solcher kann z.B. im Meta-Nest mit geringen Platzkosten vermerkt werden). Eine derartige Hinuntertransformation stellt in gewissem Sinn eine *Verfeinerung* des Eingangs-Nestes am Ausgang zur Verfügung.

Zu realisieren ist eine solche Verfeinerung beispielsweise dadurch, dass nur die Unterschiede zu der gröberen Eingangsstruktur intern verwaltet werden. Bei dieser Verwaltung tritt u.a. das in der Literatur bekannte Problem des internen Verschnitts auf, für das es mehrere Lösungsansätze gibt[34].

4.1.9. `redirect`



Dieser Baustein kann u.a. die Funktionalität von Hard- und Softlinks[35], sowie von Mount-Punkten und -Tabellen realisieren. Vom Eingang `symbolic` wird standardmäßig das Meta-Nest (falls dieses leer sein sollte, auch das Daten-Nest) abgefragt, ob es eine Beschreibung (z.B. Pfad) eines anderen Nestes enthält. Der Eingang `in` wird mit Hilfe von `control`, der an einen `strategy_*`-Baustein (siehe Abschnitt 4.2.2) angeschlossen ist, mit dem spezifizierten Nest automatisch verbunden, das dann wiederum am Ausgang `out` unverändert zur Verfügung steht.

Das Anwendungsgebiet dieses Bausteins sind Fälle, in denen sich die Pfadangabe im `symbolic`-Eingang zur Laufzeit ändern kann. Ansonsten lässt sich die genannte Funktionalität im Regelfall auch einfacher direkt auf der Instantiierungs-Strategie-Ebene lösen (vgl. Abschnitt 4.2.2).

4.1.10. `cow`

Dieser Baustein realisiert die von konventionellen privaten Mappings virtueller Adressräume bekannte Copy-on-Write-Strategie. Er hat zwei Eingänge, die mit `orig` und `tmp` bezeichnet sind, sowie einen mit `merged` bezeichneten Ausgang.

Im initialen Zustand ist das mit `tmp` bezeichnete Eingangs-Nest leer, und am mit `merged` bezeichneten Ausgang erscheint exakt derselbe Nest-Zustand wie am `orig`-Eingang. Die Aufgabe besteht in der *Isolation* des `orig`-Eingangs von allen Änderungen, die vom Konsumenten hinten am `merged`-Ausgang in Auftrag gegeben werden. Jegliche Änderungen am Nest-Inhalt des Ausgangs oder an seinem Adressraum müssen ausschließlich im `tmp`-Nest eingetragen und zwischengepuffert werden, damit sie keine tatsächlichen Änderungen am `orig`-Eingang bewirken.

Die Implementierung von `write`-Transferoperationen bzw. `get` im Schreibmodus ist relativ einfach und folgt der bekannten Methodik, wobei Löcher im Definitionsbereich von `tmp` direkt ausnutzbar sind. Schwieriger ist die Bereitstellung von `move`-Operationen. Eine Analyse dieses Teilproblems ergibt, dass es starke Verwandtschaft mit der Adressverschiebungs-Problematik bei den `map`-Bausteinen besitzt. Eine mögliche Realisierung ist daher die interne Verwendung eines `map`-Bausteins, der allerdings auch mit Löchern im `orig`-Nest korrekt umgehen können sollte.

4.1.11. `transaction`

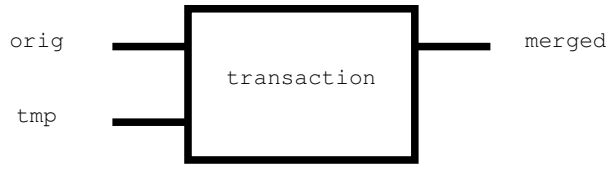
Im Unterschied zu Datenbanken, wo Transaktions-Identifier (TIDs) meist fest mit „Prozessen“ verknüpft sind, verstehe ich unter einer Transaktion eine *logische Sicht* auf einen Datenbestand, die die bekannte ACID-Eigenschaft (oder Eigenschaften anderer Transaktions-Paradigmen) besitzt, und die von beliebig vielen „Prozessen“ kooperativ (bzw. bei Verwendung zusätzlicher Sperren auch im `multiuser`- oder `multiversion`-Modell) nutzbar ist. Durch die Aufgabe fester Zuordnungen zwischen „Prozessen“ und Datenräumen wird insbesondere ermöglicht, dass ein „Prozess“ an mehreren Transaktionen, auch parallel, teilnehmen darf. Das Standard-Szenario einer festen 1:1 Zuordnung zwischen „Prozessen“ und Datenräumen ist als Spezialfall in diesem Modell enthalten.

[33] Dieses Argument läuft parallel zu demjenigen aus der Theorie über Kompressionsverfahren, das besagt, dass eine verlustfreie Datenkompression von n Ausgangs-Bits stets Kodierungen enthalten muss, die länger als n sind, wenn einige Kodierungen kürzer als n ausfallen sollen. Dies liegt daran, dass es insgesamt 2^n verschiedene Eingangs-Kodierungen gibt, die in 2^n verschiedene Ausgangs-Kodierungen zu übersetzen sind, da die Kompression in *allen* Fällen wieder rückgängig machbar sein muss.

[34] Wovon der wichtigste meiner Ansicht nach die *weitgehende Vermeidung* von Verschnitt darstellt.

[35] Zur Nachbildung eines n-fachen Hardlinks muss dieser Baustein n-fach instantiiert werden, außerdem muss noch eine Verwaltungslogik für die Freigabe hinzukommen, die beispielsweise von einem gesonderten Verwaltungsbaustein erledigt werden kann, der sozusagen die „Aufsicht“ über alle Namensänderungen und Verschiebungen aller beteiligten Namen übernimmt. Alternativ dazu kann dieser Baustein auch so ausgeführt werden, dass er n Steuer-Eingänge hinzubekommt, die zusammen diese Aufgabe übernehmen.

4.1.11.1. Sequentielle Transaktionen

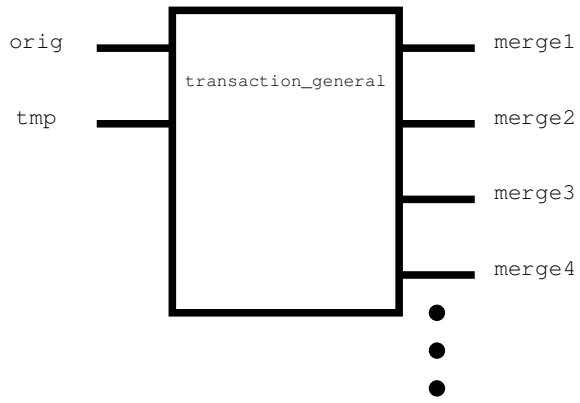


Ein `cow`-Baustein realisiert bereits einen wichtigen Teil der bekannten *Isolations-Funktionalität* von Transaktionen: um eine Rollback-Operation zu simulieren, braucht man lediglich den Inhalt des `tmp`-Nestes zu vergessen und den `merged`-Ausgang wieder auf den initialen Zustand des `orig`-Eingangs zu setzen. Ein `transaction`-Baustein lässt sich im Prinzip als leicht modifizierter `cow`-Baustein realisieren, der seine grundsätzliche Funktionsweise von `cow` erbt.

Die Commit-Operation lässt sich prinzipiell dadurch realisieren, dass man einfach *gar nichts* tut: man hört auf, Änderungen am `merged`-Ausgang vorzunehmen, und betrachtet seinen Zustand als „eingefroren". Ein derartig „eingefrorenes" Nest dient nun insbesondere als Ausgangspunkt für eine zeitlich nachfolgende (nicht-parallele) Transaktion, deren `orig`-Eingang mit dem `merged`-Ausgang der jeweils letzten erfolgreich abgeschlossenen Transaktion verbunden wird, so dass im Endeffekt lange Ketten entstehen, die alle historischen Zwischenschritte der von den Transaktions-Operationen ausgelösten Zustände des Nestes repräsentieren. Der Ausgang der letzten erfolgreich abgeschlossenen `transaction`-Instanz einer Kette wird *Aktualversion* genannt. Hinter der Aktualversion darf vorläufig nur eine einzige, noch nicht abgeschlossene Transaktion angeschlossen werden; ansonsten könnten divergierende Versions-Splits[36] entstehen, die dem Konzept einer einheitlichen logischen Sicht widersprechen.

Es ist klar, dass solche Ketten nicht beliebig lang werden dürfen. Die Entfernung von Zwischen-Instanzen ohne Änderung der Semantik ist auf folgende Weise möglich: eine abgeschlossene Transaktion darf ohne weiteres *invariante interne Operationen* ausführen, die von außen durch die (gegenüber `cow` neu hinzukommende) Operation `integrate` angestoßen werden und bewirken, dass das Ausgangs-Nest weiterhin von außen betrachtet denselben eingefrorenen Zustand behält, während der Inhalt des `tmp`-Eingangs in das `orig`-Eingangsnest *integriert* wird, so dass der Informationsgehalt des `tmp`-Nestes immer kleiner wird, bis es schließlich völlig leer wird. Bei dieser Integrations-Operation wird der Inhalt des `orig`-Eingangs so verändert, dass er schließlich mit dem eingefrorenen `merged`-Ausgang übereinstimmt; damit ist die Isolation aufgehoben. Ab diesem Zeitpunkt darf man den nunmehr funktionslos gewordenen `transaction`-Baustein aus der Kette entfernen, ohne dass sich Seiteneffekte ergeben.

4.1.11.2. Parallele Transaktionen



Mit der vorher vorgestellten Methodik lassen sich nur rein sequentiell nacheinander ablaufende Transaktionen modellieren.

Zur Herstellung echter Parallelität zwischen mehreren `transaction`-Instanzen müssen diese unbedingt an derselben Aktualitätsversion angeschlossen werden (sofern sie nicht absichtlich mit einem veralteten Zustand beginnen wollen). Wenn man dies mit getrennten Einzel-Bausteinen für jede Transaktion realisieren würde, dann müssten sich die verschiedenen Transaktions-Sichten über diesen oder einen anderen Anschluss-Punkt auf dem aktuellen Stand halten und jede für sich den aktuellen Status an ihrem Eingang nachvollziehen, was im Prinzip möglich wäre, aber eine redundante Vervielfachung von immer gleichartigen Operationen an mehrere Stellen bewirken würde[37]. Um diesen Aufwand zu sparen, ist es günstiger, eine konventionelle Implementierung von Transaktionen (siehe z.B. [GR93]), eventuell auch von verschiedenen Transaktions-Modellen (siehe z.B. [Elmbe]) in einen Baustein zu verpacken, der einen internen zentralen Puffer für die Verwaltung aller vorkommenden Versionen benutzt, und/oder dessen Eingang im `multiversion`-Modus arbeitet (dann muss allerdings auch die restliche Infrastruktur des Systems mitspielen).

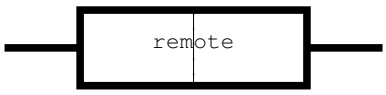
Bei der Kombination von Transaktionen mit Schutzbereichen kann das Baustein-Konzept Funktionalitäten realisieren, die in konventionellen Betriebssystem-Implementierungen einen *erheblichen* Aufwand verursachen würden. Ein Beispiel ist das Exeption-Handling in konventionellen Laufzeitumgebungen. Dieses kann auf einen Fehler nur noch *reagieren*, ihn aber meist nicht *reparieren*, da es beim Auftreten des Fehlers bereits „zu spät" ist. Transaktionen bieten die Möglichkeit, auf frühere Zustände von Adressräumen zurückzusetzen und erneute Versuche zu starten, und zwar auch auf ganze Kollektionen und Konfigurationen von Adressräumen (*spekulative Ausführung*).

Sehr interessant dürfte auch die Kombination von Transaktions-Bausteinen mit den nun folgenden Baustein-Typen sein, die verteilte Systeme in Netzwerken ermöglichen, und zwar je nach Einsatz-Stelle auf verschiedenen Ebenen einer Baustein-Hierarchie.

4.1.12. `remote`

[36] Hierfür gibt es reizvolle Anwendungen, sofern Methoden zur Reintegration von Splits vorhanden sind. Beispiele für Splits treten u.a. bei Versions-Verwaltungssystemen für Software-Quelltexte auf (z.B. `cvs` oder `subversion`)

[37] Für besondere Anwendungen, beispielsweise extrem hohe Ausfallsicherheit, könnte diese Idee eventuell trotz ihrer hohen Kosten interessant sein.

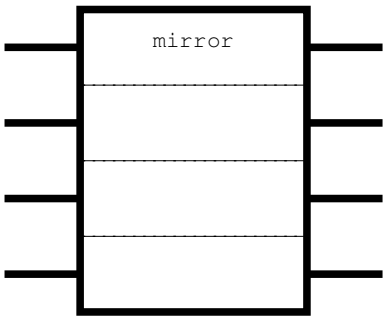


Dieser Baustein implementiert das Client-Server-Paradigma. Ein Nest wird auf einem anderen Rechner so verfügbar gemacht, dass seine funktionalen Eigenschaften nicht von einem lokalen Nest unterscheidbar sind.

Die Realisierung des internen Netzwerk-Protokolls kann statuslos erfolgen, wenn man eine `buffer`-Instanz nachschaltet, die durch ihr Caching den Datenverkehr über die interne Netzwerk-Verbindung in vielen Fällen reduziert und nebenbei die Latenzzeit vieler Operationen senkt. Bei einer statuslosen Realisierung hält der Client-Teil keinerlei Status-Informationen über den Zustand des Nestes vor, sondern muss jede einzelne Operation an den Server durchreichen. Dies führt zu einer hohen Einfachheit, Robustheit, Unabhängigkeit und Absturzsicherheit.

Fragen der Einbruchssicherheit in Netzwerke, Verschlüsselung, Authentifizierung usw. sind eine interne Angelegenheit der Baustein-Implementierung und werden ausführlich in der Literatur behandelt; in dieser Arbeit wird hierauf nicht weiter eingegangen.

4.1.13. `mirror`



Dieser Baustein realisiert das logische Konzept von Verteiltem Gemeinsamem Speicher (distributed shared memory).

Eine Baustein-Instanz von `mirror` darf sich über mehrere physikalisch getrennte Rechner erstrecken, die miteinander über interne (nicht von außen sichtbare) Kommunikationsmechanismen in Verbindung stehen (hierfür bietet sich insbesondere Gruppenkommunikation an). Die Verteilung der Baustein-Instanz über mehrere Rechner wird im Bild durch gestrichelte Linien angedeutet.

Ein- und Ausgänge sind im Grundmodell paarweise in der gleichen Anzahl vorhanden. Die Ausgänge stellen dieselbe Funktionalität dar, wie sie in einem nicht-verteilen System von der gemeinsamen Verdrahtung mehrerer Eingänge auf denselben Ausgang erfüllt wird: überall herrscht die gleiche logische Sicht, der Nest-Inhalt ist logisch betrachtet jeweils identisch und ermöglicht Kooperation im `multiuser`- oder `multiversion`-Modell.

Diese knappe Funktionsbeschreibung lässt viele Möglichkeiten für die Realisierung zu, die im Wesentlichen den bekannten Techniken aus der Literatur folgen kann (siehe z.B. [TSF90, Doe96, Esk96]). Was die Eingänge enthalten, kann bei verschiedenen `mirror_*`-Typen stark voneinander abweichen. Ich werde zwei Extremfälle kurz erläutern:

Bei einer Realisierung namens `mirror_replicate` enthalten alle Eingänge den gleichen Nest-Zustand wie die Ausgänge. Jede an irgend einem Ausgang ankommende Änderung wird sofort auf alle Eingänge aller verteilten Teil-Instanzen durchgereicht. Bei parallelem Zugriff auf mehreren Rechnern erfordert dies im Regelfall verteilte interne Synchronisationsoperationen, um die *Konsistenz* jederzeit zu wahren.

Das dadurch verursachte Problem relativ hoher Latenzzeiten lässt sich durch rechner-lokales Nachschalten je einer `buffer`-Instanz hinter die Ausgänge in vielen Fällen abmildern; daher kann die interne Realisierung der Kommunikationsprotokolle weitgehend statuslos erfolgen.

Eine andere Realisierung namens `mirror_primarycopy` besitzt nur einen einzigen Eingang, der einem ausgezeichneten Rechner fest zugeordnet ist und stets dort verbleibt, auch wenn weitere Ausgänge auf anderen Rechnern dynamisch hinzugefügt oder weggenommen werden.

Schließlich möchte ich noch auf Mischformen zwischen den beiden Extremen der vollständigen Replikation aller Daten und der replikationslosen Primärkopie hinweisen. Dazu gehören Baustein-Varianten, die ihre Eingänge nicht für Replikate, sondern für die Bereitstellung der *Summe* alles vorhandenen Platzes zu verwenden (disjunkt verteilter Speicher). Es ist prinzipiell möglich, einen Baustein `mirror_general` zu schaffen, der sowohl die Extremfälle der vollständigen Replikation als auch der Primärkopie, als auch beliebige Mischformen mit teilweiser Replikation, als auch Mischformen mit disjunkt verteiltem Speicher abdeckt.

Zu erwähnen ist weiterhin, dass die RAID-Konzepte (Redundant Array of Inexpensible Disks, vgl. [CLG+93]) als Spezialfall von `mirror` mit einer je nach RAID-Level besonderen Art der Redundanz-Verteilung aufgefasst werden können: es wird lediglich auf eine Verteilung der Ein- und Ausgänge auf verschiedene Rechner verzichtet. Eine effiziente Realisierung von `mirror` sollte sich daher darum bemühen, dass die Replikation keinen merklichen Overhead zur Folge hat. Damit würde es sich auch für einige der Aufgabengebiete eignen, in denen bisher Software-RAID eingesetzt wird.

Implementierungen von `mirror` sollten auf jeden Fall den Grundideen der Ausfallsicherheit hohe Aufmerksamkeit schenken und z.B. mit dem Fall einer Netzwerk-Partitionierung (vgl. [DGMS85]) sinnvoll umgehen können. Eine detaillierte Behandlung dieser Konzepte würde den Rahmen dieser Arbeit sprengen[38].

Schließlich ist zu erwähnen, dass sich die Funktionalität von Prozess-Migration (vgl. [Smi88]) als Spezialfall von verteilt ablaufenden Kontrollflüssen auf verteilten gemeinsamen Schutzbereichen bzw. durch Wechseln von Konfrollflüssen auf andere Rechner unter Benutzung gespiegelter Schutzbereiche darstellen lässt. Hierzu ist erforderlich, dass die verwendeten `mmu_*`-Bausteine `multiuser`-Verhalten implementieren[39].

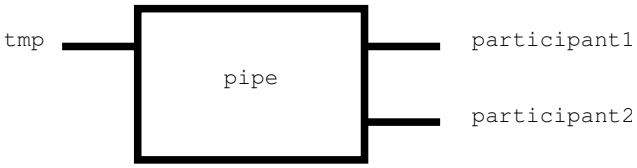
Die Verteilung von Nestern ist auch zur Herstellung verteilter gemeinsamer Namensräume sehr nützlich; `mirror` ist somit die wesentliche Grundlage für ein verteiltes Be-

[38] Es wäre interessant zu ergründen, inwieweit das Baustein-Konzept und einige der hier vorstellten Bausteine bei der internen Realisierung von `mirror` Vorteile bringen können.

[39] Im Prinzip entspricht dies einer in Software realisierten NUMA-Architektur.

triebssystem.

4.1.14. pipe



Zum Verständnis des pipe-Entwurfs muss man sich nochmals in Erinnerung rufen, dass eine Leitung eine Hierarchie-Beziehung zwischen Baustein-Instanzen darstellt und nichts direkt mit Datenfluss-Richtungen zu tun hat. In abstrakter Sicht sind die Teilnehmer an einer Pipe prinzipiell gleichberechtigt; sie werden daher an verschiedenen Ausgängen angeschlossen. Der interne Status einer pipe-Instanz wird in einem tmp-Eingang gehalten, an den sich nicht nur device_ramdisk anschließen lässt, sondern z.B. auch Puffer-Nester, die nicht in den Hauptspeicher eines Kleinrechners passen würden.

4.2. Instantiierung von Bausteinen

Zur Instantiierung der Bausteine wird ein Mechanismus benötigt. Dieser sollte außerhalb oder oberhalb der eigentlichen Baustein-Hierarchie liegen, da es sich um einen übergeordneten Kontroll-Mechanismus handelt. Nach dem Grundsatz der Trennung in Mechanismen und Strategien sind folgende Teilbereiche zu verwenden:

1. Der eigentliche Instantiierungs-Mechanismus (technische Durchführung der Instantiierung)

2. Die logische Kontrolle, welche Instantiierung zu welchem Zeitpunkt und aus welchem Anlass ausgeführt werden soll

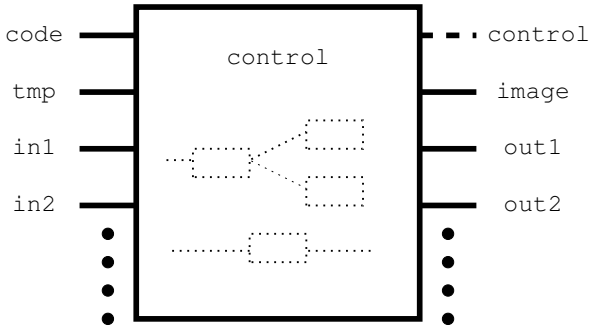
Diese Trennung hat den Vorteil, dass insbesondere für Punkt 2 mehrere verschiedene Verfahren gleichzeitig oder konkurrierend eingesetzt werden können.

4.2.1. Der Mechanismus

Für die technische Durchführung der Instantiierung schlage ich vor, einen Baustein namens control einzusetzen, von dem es auf jedem Rechner beliebig viele Instanzen geben kann, wobei im Normalfall jedoch eine Instanz ausreicht, die ihrerseits beim Urstart auf eine Weise instantiiert worden sein muss, die ausserhalb des normalen Instantiierungs-Mechanismus liegt (und logischerweise auch liegen muss[40]).

Der control-Baustein verwaltet alle von ihm erzeugten Baustein-Instanzen samt Verdrahtung. Er selbst benötigt ebenfalls Betriebsmittel.

[40] Nach dem Urstart müssen bereits Bausteine vorhanden sein, damit das System in Betrieb gehen kann (unverzichtbare Gerätetreiber und Äquivalent eines Root-Dateisystems).



Ich schlage vor, folgende Funktionsbereiche zu trennen: Die von einer control-Instanz verwalteten Baustein-Instanzen können i.a. in einem anderen (virtuellen) Adressraum (der wiederum in Schutzbereiche aufgeteilt sein kann) liegen als die control-Instanz selbst; damit sind z.B. beliebige hierarchische Schachtelungen von Adressräumen oder ganzen virtuellen Maschinen möglich[41]. Die steuernde control-Instanz bedient sich des tmp-Eingangs, um darin den gesamten notwendigen Status zu halten, und stellt am image-Ausgang ein „Prozessabbild“ bzw. eine Menge von Schutzbereichen zur Verfügung, die in einer nachfolgenden (ggf. über Zwischenbausteine wie union angeschlossene) mmu_i386 oder mmu_dummy-Instanz zur eigentlichen Ausführung gebracht werden. Damit ist eine Separation zwischen dem Adressraum möglich, der die control-Instanz beherbergt, und dem Adressraum, in dem die verwalteten Instanzen laufen sollen. Durch diese Trennung ist es u.a. möglich, Teile eines Betriebssystems in „Benutzer“-Adressräumen ablaufen zu lassen. Benutzer können auf diese Weise voneinander unabhängige Subsysteme schaffen, die von der Funktionalität her virtuellen Betriebssystemen gleichkommen. Damit verschwimmt die klassische Einteilung in Kern- und Benutzer-Zuständigkeiten; die Verteilung dieser Zuständigkeiten ist nur noch eine Frage der Konfiguration.

Eine Trennung zwischen der control-Instanz und ihrem image in separate Adressräume ist jedoch keine Pflicht[42]: wenn man den image-Ausgang in zyklischer Weise in die gleiche mmu_*-Instanz einblendet, in der auch der control-Baustein beheimatet ist, kann man ohne weiteres Single-Address-Space-Strategien[43] fahren. Als weiterer Sonderfall ist es prinzipiell auch möglich, dass ein

[41] Diese Trennung bewirkt u.a., dass sich die Abstraktionen Nest und Baustein in uniformer Weise auch in konventionellen „Benutzerprozessen“ verwenden lassen, eventuell sogar in sogenannten „Anwendungs-Programmen“ – es besteht kein prinzipieller Unterschied zwischen „Kern“- und „Benutzer“-Adressräumen bzw. -Schutzbereichen.

[42] Im Extremfall kann das gesamte System in einem einzigen Adressraum ablaufen, was z.B. bei Echtzeit-Steuerungen Performanz-Vorteile bringt. Falls Schutzbereiche genutzt werden, läßt sich trotzdem ein brauchbarer Zugriffsschutz realisieren (vgl. [C+94]). Im Unterschied zu bekannten Architekturen (vgl. [K+97a]) erlaubt die hier vorgestellte Trennungsmöglichkeit beliebige Zwischenstufen zwischen reinen Single-Address-Space-Modellen und vollkommener Aufplitterung aller Funktionen in jeweils getrennte mmu_*-Instanzen.

[43] Die Benutzung eines einzigen virtuellen Adressraums im gesamten System läßt sich dadurch modellieren, dass nur eine einzige mmu_*-Instanz vorhanden ist, in der das gesamte System abläuft. Obwohl diese Strategie in letzter Zeit hohe Aufmerksamkeit in der Literatur erhalten hat [C+94, Ros94, Ass96], sehe ich darin keinen grundlegenden Vorteil. Die Umschaltung einer Standard-MMU auf verschiedene Schutzbereiche kostet (mit wenigen Ausnahmen) im Worst Case größenordnungsmäßig ebenso viel wie die Umschaltung zwischen verschiedenen Adressräumen; lediglich der Best Case geht effizienter. In welcher Größenordnung sich dieser mögliche Performanz-Gewinn auf das *Gesamtsystem* auswirkt, und ob Effekte gleicher Größenordnung nicht auch

control-Baustein sich selbst verwaltet[44]; in diesem Fall besteht kein Unterschied zwischen „innen“ und „außen“; eine Destruktion der control-Instanz würde dann nicht nur zur Destruktion aller verwalteten Instanzen führen, sondern nie mehr eine erneute Re-Konstruktion ermöglichen (dies kann ausnahmsweise beim Herunterfahren des Rechners auch erwünscht sein).

Die Befehle zum Instantiieren / De-Instantiieren der verwalteten Instanzen sowie zum Verdrahten werden über den control-Ausgang gegeben. Eine Möglichkeit ist die Einführung weiterer Elementar-Operationen, oder die Benutzung der Schnittstelle von generischen Operationen (Kapitel 6). Da auf dieser Leitung nur Steuer-Operationen benutzt werden, ist sie gestrichelt gezeichnet. Über den control-Ausgang lassen sich fernerhin die Attribute der instantiierten Bausteine, Ein- und Ausgänge und die Verdrahtungs-Struktur abfragen. Auftraggeber für diese Operationen können beliebige Bausteine sein; i.A. dürfen diese auch in image liegen[45]. Zu Zwecken der Kommunikation nach außen existiert eine dynamische Anzahl von Ein- und Ausgängen in* und out*[46], die als Schnittstellen für LRPC dienen und die zu beliebigen Zwecken einsetzbar sind[47].

Es ist sinnvoll, das Instantiieren / De-Instantiieren der in image befindlichen Instanzen vom Konstruieren / Destruieren zu trennen: Beim Instantiieren wird lediglich Platz für die Baustein-Infrastruktur (z.B. statische Attribute) angelegt und die Verdrahtung ermöglicht. Die Parametrierung der Baustein-Instanz (z.B. Festlegen dynamischer Attribute) braucht erst später beim Konstruieren zu erfolgen. Wenn man einen Baustein nur destruiert, aber nicht de-instantiiert, wird er logisch betrachtet lediglich in den Zustand der Statuslosigkeit geschaltet (d.h. nach dem Abarbeiten aller evtl. noch vorliegenden Aufträge wird der evtl. noch vorhandene interne Status auf die Eingänge abgewälzt, danach werden keine Operationen mehr bearbeitet), und er kann anschließend durch Konstruieren wieder in Betrieb genommen werden.

Details zur internen Realisierung von control werden in dieser Arbeit nicht behandelt; die grundlegende Funktionsweise soll hier nur skizziert werden:

Im code-Eingang wird dynamisch linkbarer Maschinencode für alle instantiierbaren Baustein-Typen bereit gehalten[48]. Bei der erstmaligen Verwendung wird er nach tmp kopiert und dabei gelinkt[49]. Die Instanzen der verwalteten Bausteine werden in zwei Phasen erzeugt: aus den statischen Attributen des zu instantiirenden Baustein-Typs wird der Speicherplatzbedarf für eine Instanz ausgelesen und entsprechender Platz in tmp reserviert. Anschließend wird eine bausteinspezifische Instantiierungs-Routine aufgerufen; diese kann bei Bedarf weitere Aktionen ausführen[50]. Nach der Instantiierung sollte im Regelfall die Verdrahtung statt finden; hierbei kann bereits auf Konflikte zwischen statischen Kompetenz-Attributen und Verhaltens-Attributen geprüft werden. Die Konstruktion erfolgt ebenfalls durch Aufruf einer baustein-spezifischen Routine. Da u.U. erst zu diesem Zeitpunkt dynamische Attribute festgelegt werden, kann die Konstruktion prinzipiell auch fehlschlagen[51]. Destruktion und De-Instantiierung funktionieren analog.

4.2.2. Einige mögliche grundlegende Strategien

Bei den *Anlässen* für eine Instantiierung sind mehrere mögliche Szenarien zu unterscheiden:

1. Manche Baustein-Arten wie z.B. dir_* können nur an bestimmten vorgegebenen oder vorgebbaren Stellen einer Baustein-Hierachie instantiiert werden, weil sie einen *Interpreter* für ein bestimmtes *konkretes Datenformat* darstellen. Es macht i.a. keinen Sinn, andere Baustein-Arten als die für die jeweilige Interpretation geeigneten zu instantiieren; häufig existiert für ein bestimmtes Datenformat nur eine einzige Interpreter-Baustein-Art (in einem gewachsenen System allerdings oft in verschiedenen *Versionen*, von denen bei der Instantiierung die geeignete ausgewählt werden muss).

2. Andere Baustein-Arten wie z.B. mmu_* oder geschachtelte control-Instanzen werden fast ausschließlich auf Veranlassung von Benutzer-Aktivitäten instantiiert.

durch andere architekturelle Maßnahmen erzielbar sind, sind genauer zu untersuchende Fragen. Meiner Ansicht nach stellt die Einengung auf ein *festes* Single-Space-Modell eine Einschränkung dar, die mit der hier vorgestellten Architektur aufgebrochen werden kann.

[44] Die einfachste Lösung hierfür besteht darin, dass der tmp-Eingang bereits bei der „Ur-Instantiierung“ das richtige Prozessabbild mit der Baustein-Konfiguration enthält, mit dem zu starten ist. Wenn dieses auch die steuernde control-Instanz enthält, entsteht automatisch die Selbstverwaltung. Bei der Systemgenerierung eines bootfähigen Systems wird auf einem (fremden) Rechner ein entsprechendes Prozessabbild mit allen zum Urstart notwendigen Instanzen erstellt; dieses kann dann mit konventionellen Ladern analog zu klassischen Unix-Kernen geladen werden.

[45] Hierbei kann es i.A. zu Problemen mit Rekursion kommen, wenn derartige Auftraggeber sich selbst betreffende Operationen in Auftrag geben.

[46] Alternativ zu diesen Ein- und Ausgängen könnte man auch LRPC-Varianten von remote benutzen; dadurch müssen keine Verdrahtungsleitungen zwischen Kommunikationskanälen auf dem selben Rechner verwaltet werden (Einfachheit). Die explizite Verwaltung von Leitungen durch eine übergeordnete Instanz kann jedoch Vorteile in Bezug auf Sicherheit bringen, da eine Leitung bereits eine rudimentäre Form von Authentifizierung bereitstellt, die bei geeigneter Abschottung der Subsysteme (z.B. Partitionierung in Schutzbereiche) von einem vertrauenswürdigen „logischen Supervisor“ einbruchssicher gemacht werden kann, ohne zu aufwendigeren Massnahmen wie Verschlüsselung beim LRPC greifen zu müssen.

[47] Dies ist nicht nur zur Herstellung mehrerer virtueller Rechner-Partitionen interessant oder zu deren Kommunikation über gemeinsame Daten wie „Dateisysteme“, sondern beispielsweise auch zum Checkpoint / Restart einer Betriebssystem-Instanz, wenn man beispielsweise vor tmp eine transaction-Instanz vorschaltet. Eine weitere Anwendung wäre beispielsweise das Vorhalten eines kompletten Ersatz-Betriebssystem-Abbildes, das bei Störungen einspringen kann.

[48] Um reibungslose Re-Instantierungen bzw. Re-Konstruktionen zu garantieren, darf der Inhalt von code zur Laufzeit nur um neue Baustein-Typen bzw. um neue (fehlerbereinigte) Versionen erweitert werden, nicht jedoch grundlegend abgeändert werden.

[49] Mehrmalige Herstellung von jeweils anders gelinkten Kopien ist möglich und sinnvoll, z.B. um direkte statt indirekte Prozeduraufrufe verwenden zu können. Dem steht jedoch der Nachteil von Code-Reduplikation gegenüber.

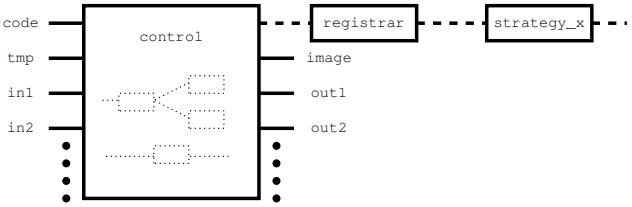
[50] Bei sicherheitskritischen Anwendungen kann die Ausführung von Instantiierungs- und Konstruktor-Routinen durch die control-Instanz eine Gefährdung darstellen, insbesondere wenn den in der nachgeschalteten mmu_*-Instanz ablaufenden Aktivitäten oder dem Inhalt von code nicht zu trauen ist. In diesem Fall sollte der Aufruf durch LRPC stattfinden und im Kontext der nachgeschalteten mmu_* ausgeführt werden.

[51] Zur Lösung dieses Problems könnte man die Attribut-Prüfung von der eigentlichen Konstruktion abtrennen; dadurch erhöht sich jedoch die Komplexität der Instantiierungs-Logik.

3. Es gibt Mischformen zwischen beiden Extremen: ein Verzeichnis-Pfadname wird zwar vom Benutzer vorgegeben, die Art des jeweils zu inantiierenden Bausteins hängt jedoch vom Datenformat ab.

4. In manchen Fällen können Instantiierungen oder Re-Instantiierungen auch ohne *konkreten* äußeren Anlass geschehen, z.B. zeitgesteuert in selbsttätig optimierenden Betriebssystemen (vgl. [BR76]) oder zur Erzielung von Fehlertoleranz und Ausfallsicherheit.

Der Fall 1 lässt sich entweder direkt in `control` abhandeln, oder durch einen eigenen Baustein-Typ `registrar`, der als *Registrar* für verschiedene Datenformate dient. Datenformate sollten vorzugsweise im zugehörigen Meta-Nest beschrieben werden; es gibt aber auch auch Sonderfälle wie Platten-Partitionen oder `dir_*`-Bausteine für konventionelle Archiv-Datei Formate (wie `.tar` oder `.zip`), bei denen die Format-Erkennung durch Inspektion des Inhaltes des betreffenden Nestes erfolgen muss. Ein Registrar verwaltet daher auch die Methoden für die automatische Format-Erkennung.



Um eine möglichst hohe Flexibilität bei verschiedenen Auslösern von Instantiierungen zu erreichen, schlage ich die Einrichtung von `strategy_*`-Bausteinen vor. Diese steuern den `control`- oder `registrar`-Baustein und erhalten von dort Informationen über vorhandene Datenformate und Kompetenzen; sie werden ihrerseits von anderen Steuer-Instanzen wie z.B. Benutzer-Operationen oder andere `strategy_*`-Instanzen gesteuert. Im Bild sind die Steuerleitungen gestrichelt gezeichnet.

Über die Steuerleitungen werden vornehmlich Konstruktions- und Destruktions-Anweisungen gegeben. Diese brauchen sich jedoch nicht unbedingt auf physische Baustein-Instanzen zu beziehen, sondern können sich auch auf *virtuelle Instanzen* beziehen. Aufgabe von `strategy_*` ist u.a., solche virtuelle Instanzen vorzuspiegeln und ihre Übersetzung in physische Instanzen durchzuführen. Eine Konstruktions-Anweisung spezifiziert nur das *Was*, nicht das *Wie* einer durchzuführenden Aktion. Das Problem des Wie wird in `strategy_*` gelöst (daher auch der Name).

Verzweigungen im `control`-Pfad-Netzwerk entsprechen unterschiedlichen *Sichten* (personalities) auf ein Betriebssystem.

Beispiele: `strategy_asciipath` könnte Pfadnamen in ASCII-Codierung akzeptieren und indirekt über `registrar` bzw. `control` auf einen Verzeichnisbaum von `dir_*`-Instanzen abbilden. Ein weiterer Baustein `strategy_utf_ascii` ließe sich dazwischen schalten, um Pfadkomponenten in Unicode- oder UTF-8-Codierung nach ASCII zu übersetzen und dabei UTF-8-Sonderzeichen in eine umkehrbar eindeutige äquivalente Mehrzeichen-ASCII-Repräsentation zu übertragen. Ein weiteres Beispiel wäre `strategy_ascii_bin`, der binär kodierte Zahlen-Schlüssel (z.B. Feldschlüssel in Datenbanken) aus äquivalenten ASCII-Repräsentationen erzeugt. Weitere `strategy_*`-Arten könnten z.B. die Verwaltung von Internet-URLs übernehmen; der Erweiterbarkeit und Flexibilität werden durch diese Architektur kaum Grenzen gesetzt. Um ständig wiederholende Instantiierungen / De-Instantiierungen von Bausteinen zu verhindern, kann `strategy_cache` einen „Vorrat“ von häufig benutzten (virtuellen) Instanzen anlegen, die bei Nichtbenutzen evtl. lediglich auf „statuslos“ geschaltet werden. Die Funktionalität des klassischen `fork`-Systemaufrufs von Unix lässt sich ebenfalls durch `strategy_*`-Bausteine, diejenige von `/proc`-Dateisytemen durch spezialisierte Arten von Registraren herstellen, die für automatische Default-Instantiierungen registrierter Komponenten sorgen.

Das Problem von inkompatiblen Kompetenzen und Verhalten von versuchten Baustein-Verdrahtungen bzw. Konstruktionen lässt sich durch einen dazwischen geschalteten `registrar_intermediate` lösen. Dieser kennt alle möglichen `adaptor_*` und sonstigen Konversions-Baustein-Typen samt ihren Eigenschaften und veranlasst bei Bedarf die automatische Zwischenschaltung der geeigneten Komponenten. Auf ähnliche Weise lässt sich Netzwerk-Transparenz durch automatisierte Zwischenschaltung von `remote` und `mirror` erzielen. Es ist sogar möglich, die Existenz von dazwischen geschalteten Bausteinen nach oben hin zu verbergen und damit den virtuellen Eindruck eines allein stehenden Systems herzustellen (transparent verteiltes Betriebssystem).

Bei Implementierung geeigneter Strategien (z.B. [LS94]) lässt sich Fehlertoleranz durch automatisches Re-Instantiieren von ausgefallenen Bausteinen erzielen, oder in Kombination mit der Netzwerk-Transparenz zur dynamischen Lastbalancierung nutzen. Häufig zitierte Schlagworte wie Verfügbarkeit, Autonomie, etc. lassen sich auf diese Weise realisieren. Es sind unzählige weitere Anwendungen für `strategy_*` und `registrar_*`-Varianten denkbar.

Für Datenbank-Konstrukteure könnte eine reizvolle Aufgabe darin bestehen, einen Baustein `strategy_sql` zu entwickeln, der sich bei geeigneter Auslegung nicht nur für die klassischen Einsatzgebiete in Datenbanken eignet[52], sondern das Prinzip der Trennung von logischen und physischen Zugriffspfaden auch auf solche Einsatzgebiete ausdehnt, die bisher als Domäne von Betriebssystemen[53] oder verteilten Systemen betrachtet wurden (insbesondere das Management virtueller Benutzer-Adressräume[54]).

[52] Durch dazwischen geschaltete `remote`-Bausteine dürften sich auch bei den klassischen Einsatzgebieten von Datenbanken neue Konfiguationsmöglichkeiten ergeben, die bei bisherigen Datenbank-Architekturen nicht einfach zu realisieren waren. Zu nennen ist hier insbesondere die Einbeziehung des Verhaltens von Flaschenhälsen wie `remote`-Instanzen in Rechner-übergreifende Query-Optimierungs-Strategien. Es kann beispielsweise in manchen Szenarien durchaus vorteilhaft sein, Tabellen erst mittels `remote` auf Client-Rechner zu exportieren und dort mittels `buffer` zu cachen, bevor umfangreiche Joins oder andere Produkte daraus hergestellt werden.

[53] Durch derartige Verbindungen könnte man beispielsweise Pfadnamen durch SQL-Abfragen ersetzen.

[54] Dieser Ansatz geht über die Integration von Dateien in Datenbanken (vgl. [HR01, Abschnitt 3.5]) weit hinaus, da man sowohl Dateien als auch Datenbank-Datensätze jeweils sowohl über SQL als auch über Verzeichnis-Pfade zugreifbar machen kann; bei dieser Stufe der Integration verschwimmen die konventionellen Unterschiede zwischen Datenbanken und Betriebssystemen völlig.

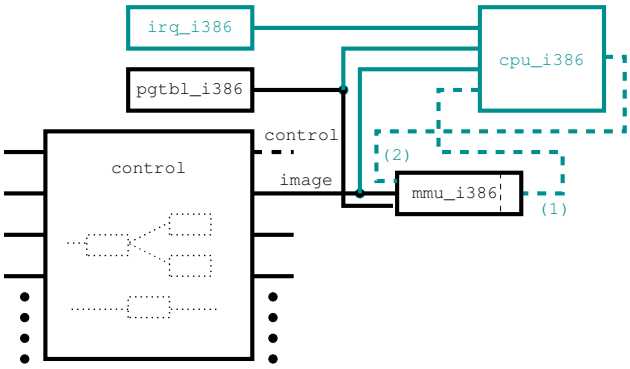
4.3. Lösung von Infrastruktur-Aufgaben

Einige Infrastruktur-Aufgaben wie das Herstellen der Verdrahtung, die Prüfung der Kompatibilität von Kompetenzen und Verhalten, oder das Platzmanagement innerhalb von `image`-Nestern können und sollen von `control`-Bausteinen übernommen oder verantwortet werden. Bei anderen Aufgaben wie der Kontrollfluss-Implementierung oder der Hardware-Anbindung kann dies ebenso gelöst werden; es gibt jedoch noch weitere Lösungmöglichkeiten. Da diese Alternativen von der Realisierung in konventionellen Betriebssystem-Architekturen abweichen, sollen sie in diesem Abschnitt etwas näher betrachtet werden.

4.3.1. Kontrollfluss-Implementierung

In konventionellen Betriebssystem-Architekturen werden Kontrollflüsse als eigenständige Abstraktion betrachtet. Zur Begründung der in Abschnitt 2.3.1 erwähnten Behandlung von Kontrollflüssen als Hilfsabstraktion soll nun gezeigt werden, wie man Kontrollflüsse prinzipiell mittels Bausteinen und Nest-Abstraktionen implementieren kann.

Die Grundidee besteht darin, einen Baustein `cpu_i386` zu schaffen, den man als „Treiber für die CPU-Hardware“ betrachten kann und der u.a. das Multiplexen einer physikalischen CPU auf mehrere „virtuelle CPUs“ übernimmt, die die Kontrollflüsse darstellen. Von mehreren Umsetzungs-Möglichkeiten dieser Idee sei hier eine herausgegriffen:



Die grau gezeichneten Teile entsprechen dem Infrastruktur-Anteil, der in bisherigen Darstellungen zur Vereinfachung nicht betrachtet wurde. Der Baustein `irq_i386` enthält die Tabelle der Unterbrechungs-Vektoren; üblicherweise existiert in einem physikalischen Rechner nur eine einzige Instanz davon. Auch von `cpu_i386` ist normalerweise nur eine einzige Instanz vorhanden, da symmetrische Multiprozessoren meistens auch mit symmetrischer Behandlung bei den Scheduling-Strategien (die ihrerseits wieder durch separate Bausteine gekapselt werden können) realisiert werden[55]. Die Ausführung einer Unterbrechung beginnt zunächst in der `cpu_i386`-Instanz, die für die Umsetzung der Unterbrechung in einen vollwertigen Kontrollfluss sorgt (vgl. [KE95]).

Da `cpu_i386` wenigstens auf die physikalische Startadresse der MMU-Seitentabellen zugreifen muss (z.B. um die Adresse bei der Schutzbereichs-Umschaltung in ein Spezialregister zu laden), wurde eine Trennung zwischen den Seitentabellen `pgtbl_i386` und der eigentlichen `mmu_i386`-Instanz vorgenommen. Bei mehreren Adressraum-Instanzen kommen `mmu`-Bausteine üblicherweise in mehreren Instanzen vor, die auf separate Eingänge von `cpu_i386` zu verdrahten sind. Zur Durchführung von Stackverwaltungs-Aufgaben und dergleichen benötigt `cpu_i386` weiterhin Zugriff auf das `image`, das in der `mmu_i386`-Instanz in einen virtuellen Adressraum umgesetzt werden soll.

Die Wechselwirkung zwischen `cpu_i386` und `mmu_i386` beim Erzeugen neuer Kontrollflüsse lässt sich auf verschiedene Arten modellieren. Wenn man `cpu_i386` als einzige *aktive Komponente* des Gesamtsystems auffasst, dann sollte sie die oberste Stellung in der gesamten Baustein-Hierarchie innehaben, was durch die gestrichelte Verdrahtung (1) angedeutet werden soll; der in `cpu_i386` erzeugte neue Kontrollfluss delegiert sich selbst in untergeordnete Bausteine, auch transitiv. Wenn man dagegen den *Auslöser* einer Kontrollfluss-Erzeugung betrachtet, so liegt er in irgend einem Baustein, der im `mmu`-Abbild läuft; demnach müsste man mittels Leitung (2) die `cpu_i386`-Instanz der `mmu`-Instanz unterordnen. Man kann den Kontrollfluss-Erzeugungs-Aufruf in Lösung (1) jedoch auch als *Rücklieferung* des Kontrollflusses an den Ur-Eigentümer `cpu_*` betrachten (vgl. Argumentation in Abschnitt 5.1).

Das Modell (1) ermöglicht eine systematische Interpretation: passive Hardware-Komponenten wie der Hauptspeicher werden von Ur-Eigentümern repräsentiert, die im Regelfall ganz unten in einer Baustein-Hierarchie beheimatet sind; die aktive Hardware-Komponente CPU wird durch einen „Treiber“ repräsentiert, der ganz oben sitzt und dessen Verantwortungs-Delegation genau in umgekehrter Richtung abläuft. Es ist auch eine noch radikalere Interpretation möglich: die Betriebssystem-Software besitzt selbst keine vorgegebenen Eigentumsrechte, sondern erhält sie beim Urstart von der Hardware als Unterbesitz verliehen, den sie dann innerhalb der Baustein-Hierarchie gemäß Abschnitt 5.1 weiter verleihen darf. Auch die CPU wird von der äußeren Hardware nur als „Leihgabe“ an die `cpu_*`-Instanz vergeben. Der Entzug oder die Hinzunahme von Hardware-Komponenten zur Laufzeit fügt sich nahtlos in dieses Modell ein.

Diese Interpretation der Baustein-Hierarchie unterscheidet sich von den klassischen Betriebssystem-Hierarchie-Konzepten [Dij68] und [HFC76] auf grundlegende Weise: in THE [Dij68] wurde die Virtualisierung der CPU auf unterster Ebene (Schicht 0) vorgenommen, die Virtualisierung des Speichers auf zweitunterster Ebene (Schicht 1). Die höheren Schichten können sich in dieser Denkweise auf „virtuelle Maschinen“ abstützen und diese um neue Operationen *erweitern*. In der hier vorgestellten Architektur werden sowohl CPU als auch Speicher als *überall nutzbare Infrastruktur* angesehen, die durch die Nest-Abstraktion repräsentiert wird und deren *individuelle Instanzen* mittels Verdrahtungs-Leitungen an alle Stellen *transportiert* werden können, wo sie benötigt werden. Die gegensätzlichen Haupt-Transportrichtungen für aktive und passive Hardware-Komponenten unterscheiden sich von den genannten klassischen Hierarchie-Konzepten. Eine Konsequenz hieraus ist, dass Hardware-Treiber je nach Funktion

[55] Gelegentlich werden Rechner mit CPUs unterschiedlicher Befehlssätze bzw. unterschiedlicher Hersteller ausgerüstet. In diesem Fall ist es sinnvoll, unterschiedliche `cpu_*`-Instanzen einzusetzen.

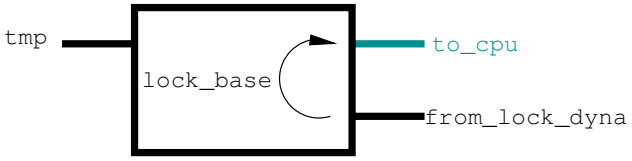
ganz oben oder ganz unten in der Baustein-Hierarchie stehen können, wodurch eine weitere Verfeinerung von Mikro- oder Exo-Kern-Aufteilungsprinzipien möglich ist.

Das *Konzept* einer derartigen hierarchischen Gliederung impliziert nicht, dass sämtliche Hierarchie-Beziehungen unbedingt *explizit* als direktionale Verdrahtungs-Leitungen repräsentiert werden *müssen*. In realen Implementierungen können die grau gezeichneten Leitungen beispielsweise auch durch verbindungslose Kommunikation im Stil einiger Middleware-Architekturen ersetzt werden. Kontrollfluss-Wechsel und LRPC werden dann als überall nutzbare Infrastruktur angesehen. Die zugehörigen Mechanismen sollten jedoch nur relativ zu einer gemeinsamen *Anker-Instanz* wie z.B. einer gemeinsamen `cpu`- oder `control`-Instanz implementiert werden, damit verschiedene `cpu_*`- oder `control_*`-Arten mit ggf. unterschiedlichen Kommunikations-Mechanismen oder -Implementierungen möglich sind. LRPC kann auch in speziellen `lrpc`-Bausteinen konzeptuell lokalisiert werden, ähnlich zur Lokalisierung von RPC durch `remote`.

4.3.2. Reflektoren

Die *Durchführung* von LRPC sowie von Basis-Synchronisation geschieht stets in Anker-Bausteinen wie `cpu_*`. Die Nest-Schnittstelle enthält nur abstrakte *Auslöse*-Mechanismen dieser Operationen in einer einheitlichen Repräsentation; die tatsächliche Durchführung dieser in allen Nest-Instanzen verwendbaren Infrastruktur-Aufgaben erfolgt dagegen in Ankern oder damit zusammen hängenden (Unter-)Bausteinen. Dieses Modell benutzt jedoch Kommunikation, die außerhalb der Baustein-Verdrahtung geschieht. In diesem Abschnitt stellen wir einige theoretische Überlegungen an, wie sich auch außerhalb der Baustein-Verdrahtung stattfindende Kommunikation *in abstrakter Weise* durch das unidirektionale hierarchische Verdrahtungs-Modell *darstellen*[56] lässt.

Die Realisierung der `lock`-Operationen (vgl. Abschnitt 3.3.5) kann beispielsweise mit Hilfe des folgenden Bausteins hierarchisch gegliedert werden:



Der Baustein `lock_base` realisiert nur nicht-überlappende Locks auf relativ niedriger Ebene, indem er als *Reflektor*[57] agiert, der die am unteren Eingang ankommende Lock-Operationen in `notify_*`-Operationen (siehe Abschnitt 5.2) umwandelt und über den oberen Eingang an `cpu_i386` in Gegenrichtung weiter schickt, der schließlich das Blockieren und Aufwecken von Konfrollflüssen ausführt. Wechselseitig überlappende Locks können in einem anderen Baustein `lock_dyna` implementiert werden, der sich um höherwertige Belange kümmert.

Die Benutzung eines Reflektors geschieht hier zur *Abkürzung* eines ansonsten sehr komplizierten oder gar unmöglichen Rückweges durch eine Baustein-Hierarchie, bei dem die dynamische Aufruf-Kette (Stack) nicht zerstört werden darf. Die Existenz dieser Abkürzung bzw. die Umgehung der normalen Hierarchie-Beziehungen könnte zu der Frage Anlass geben, ob gleichberechtigte und/oder verbindungslose Kommunikation nicht besser geeignet wäre als die hier propagierte direktionale verbindungsorientierte Kommunikation entlang von Verdrahtungs-Leitungen.

Meiner Ansicht nach stellt das scheinbar weniger mächtige eingeschränkte Instrument einer direktionalen verbindungsorientierten Kommunikation ein *Hilfsmittel zur Einhaltung von Strukturierungsprinzipien* dar. Die Vorteile dieser gesicherten und disziplinierenden Hilfsmittel in einem großen Gesamtsystem überwiegen meiner Ansicht nach den Nachteil, dass die Disziplin an wenigen Stellen aus Gründen durchbrochen werden muss, die zur *inhärenten Problematik*[58] direktionaler hierarchischer Strukturen gehören, und die auch in hierarchisch unstrukturierten Systemen in irgend einer Form vorkommen (wo sie lediglich nicht *explizit* modelliert werden). In dieser Sichtweise werden Reflektoren nicht vorrangig als Mittel zur Behebung der Beschränkungen des Modells gesehen, sondern als Umsetzung des Lokalitätsprinzips durch *explizite Kennzeichnung* derjenigen Stellen, wo eine Hierarchie ausnahmsweise durchbrochen wird oder werden *muss*. Das Merkmal der expliziten Kennzeichnung zyklischer Beziehungen ist meistens nicht in verbindungslosen Kommunikations-Infrastrukturen oder in miteinander verzeigerten Objekt-Geflechten objektorientierter Entwürfe zu finden und dort auch nur sehr schwer einzuführen[59].

[56] Damit wird keine Implementierung impliziert.

[57] Das Konzept des Reflektors lässt sich so verallgemeinern, dass praktisch *alle* entgegen der Stromrichtung ankommenden Operationen in äquivalente `notify_*`-Operationen umgesetzt werden und in dieser Form eine Baustein-Hierarchie in Gegenrichtung (Stromrichtung) durchlaufen. Eine exzessive und unreflektierte Ausweitung dieser Idee kann jedoch die hierarchische Gliederung eines Systems aushöhlen und unterwandern, sowie bei unbedachtem Einsatz von Gegen-Reflektoren am oberen Ende einer Hierarchie auch zu Endlos-Zyklen ähnlich einer Endlos-Rekursion führen. Die Abgrenzung sinnvoller oder unumgänglicher Einsatz-Szenarien ist schwierig und bedarf weiterer Forschungen.

[58] Wenn man die CPU-Treiber ähnlich wie in THE [Dij68] ganz unten in der Baustein-Hierarchie anordnen würde, dann bliebe das Problem der „Gegenrichtung“ bzw. des „Gegenverkehrs“ trotzdem erhalten: irgendwie müssen aktive Kontrollflüsse ja in hierarchisch hochstehende Bausteine gelangen, und von dort aus müssen sie Aufträge an tiefere Schichten erteilen können. Bei der *regulären* Auftrags-Abarbeitung geschieht die vorzugsweise *logische* Kontrollfluss-Übergabe *von oben nach unten*. Daher ist es vorteilhaft, den *Standard-Fall* durch eine Baustein-Hierarchie mit oben stehender CPU zu modellieren und nur die *Ausnahmen* wie z.B. Signale oder Unterbrechungen als „Gegenrichtung“ zu modellieren.

[59] In [Ass96] werden dynamische Aufruf-Beziehungen in einem Geflecht von Objekt-Instanzen, die verbindunglos kommunizieren, *horizontale Abhängigkeiten* genannt. In [Ass96, Abschnitt 3.2.6.3] wird das Problem zyklischer horizontaler Abhängigkeiten auf zyklische Abhängigkeiten auf Entwurfsebene zurück geführt (vgl. hierzu Abschnitt 6.2.2); die Erkennung zur Übersetzungszeit eines Systems wird jedoch wegen der Turing-Vollständigkeit als unmöglich betrachtet. Daher wird ein Algorithmus zur Laufzeit-Erkennung vorgestellt, mit dessen Hilfe jedoch wegen der Turing-Vollständigkeit kein Nachweis für die Zyklenfreiheit des Gesamtsystems unter allen Betriebsbedingungen geführt werden kann.

Im Gegensatz dazu verhindert der hier vorgestellte direktionale verbindungsorientierte Architektur-Ansatz die Entstehung zyklischer horizontaler Abhängigkeiten von vornherein durch die am Anfang dieses Kapitels genannten Einschränkungen bei den Verdrahtungs-Regeln. Durch die in Kapitel 5 vorgestellten `notify_*`-Operationen können zwar zyklische Abhängigkeiten wieder eingeführt werden, diese lassen sich jedoch durch verzögerte Bearbeitung (Pufferung) entschärfen, indem ein dazwischen geschalteter Beobachter-Baustein gleich lautende

4.3.3. Anbindung an Hardware und Unterbrechungen

In Abschnitt 4.3.1 wurde bereits der Baustein `irq_i386` als beispielhafte Möglichkeit für die Anbindung an die Unterbrechungen der Rechner-Hardware vorgestellt. Anbindungen an Memory-Mapped-IO (z.B. bei Grafikkarten-Puffern) lassen sich ziemlich direkt als Treiber-Bausteine realisieren. Bei Zugriffen mittels spezieller (privilegierter) IO-Prozessorbefehle kommen Umsetzungen in Generische Operationen (siehe Kapitel 6) in Frage, aber auch die Darstellung des nur durch privilegierte Befehle ansprechbaren IO-Adressraums durch eine spezielle Nest-Instanz. Letztere Möglichkeit realisiert universelle Generizität auf unterster Ebene, indem zunächst ein universeller „Treiber“ für die speziellen IO-Befehle des Prozessors bereit gestellt wird, mit dessen Hilfe erst auf der nächst höheren Ebene konkrete Treiber für spezielle Hardware implementiert werden.

Die Verwendung privilegierter Befehle oder Adressräume kollidiert unter Umständen mit der rekursiven Schachtelung von Adressräumen durch `control`- und `mmu`-Instanzen. Die Benutzung privilegierter Befehle ist abhängig von Hardware-Schutzmechanismen einer Prozessor-Architektur. Viele Prozessoren sind von ihrer Hardware-Ausrüstung her (insbesondere wegen der Unterbrechungen) nur für *eine einzige* Adressraum-Instanz gerüstet, in denen privilegierte Befehle benutzt werden dürfen, wodurch u.U. eine direkte rekursive Schachtelung von `mmu_*`-Instanzen nicht auf einfache Weise möglich ist. Andere Prozessoren haben feinere Abstufungen verschiedener Privilegierungs-Stufen (insbesondere in Form sogenannter Schutz-*Ringe* [Gra68]).

Wenn man auf hohe Performanz Wert legt, wird man bei den meisten zeitgenössischen Prozessor-Architekturen sämtliche Treiber-Baustein-Instanzen, die privilegierte Befehle *direkt* benutzen, in einen gemeinsamen ausgezeichneten Adressraum oder Schutzbereich legen oder legen müssen[60]. Andere Treiber, die keine privilegierten Befehle direkt benutzen, können in unprivilegierte Adressräume bzw Schutzbereiche ausgelagert werden (vgl. Exokern-Architekturen [EKO95]). Welche dieser Instantiierungs-Strategien benutzt werden soll, ist bei der hier vorgestellten Software-Architektur im Rahmen der Hardware-Begrenzungen frei wählbar, ggf. auch dynamisch zur Laufzeit beim Nachladen eines Treibers. Durch geeignete `strategy_*`-Bausteine lässt sich Transparenz bezüglich der Wahl eines konkreten Plazierungs-Adressraums für einen Treiber herstellen, sowie für die automatische Zwischenschaltung von geeigneten LRPC-Mechanismen sorgen.

In vielen Fällen kann man eine *Weiterleitung* von Unterbrechungen in andere Adressräume auch durch Software realisieren. Da die Ausführung privilegierter Befehle in unprivilegierten Adressräumen bzw Schutzbereichen eine Unterbrechung auslöst, kann man die Weiterleitung dadurch auslösen. Falls die Weiterleitung so geschieht, dass am Ende die gleiche Semantik wie bei tatsächlich privilegierten Adressräumen bzw Schutzbereichen herauskommt, ist eine echte Schachtelung virtueller Maschinen möglich (vgl. [Cre81, BPS81]).

Die Herstellung der gleichen Semantik wie bei „echten“ privilegierten Befehlen ist jedoch mit einem relativ hohen Laufzeit-Aufwand verbunden; weiterhin werden bereits auf unterer Ebene erfolgte Transformationen in höherwertige Abstraktionen wieder auf niedrige Abstraktionsniveaus hinuntertransformiert bzw nicht ausgenutzt. Diese „Umwege“ lassen sich vermeiden, wenn man in rekursiv geschachtelten virtuellen Maschinen keine Treiber für reale Hardware benutzt, sondern *Pseudo-Treiber*. Ein Beispiel hierfür ist `mmu_virtual`, der nicht an die physischen Unterbrechungen angebunden wird, sondern letztlich auf die Funktionen einer äußeren `mmu_i386`-Instanz zurückgreift (vgl. [Lie95c]). Analog dazu entspricht `cpu_virtual` einem virtuellen CPU-Multiplexer ähnlich zum Konzept der *scheduler activations* [A+91], der innerhalb einer rekursiv geschachtelten virtuellen Maschine ein eigenes logisches Kontrollfluss-Konzept realisiert, das sich letztlich transitiv auf die äußere `cpu_i386`-Instanz abstützt. Für virtuelle Gerätetreiber gilt analoges.

gepufferte `notify`-Nachrichten eliminiert oder andere Techniken des Rekursions-Abbruchs implementiert.

[60] Falls die Unterbrechungs-Sprungtabellen frei wählbare Adressraum-Kennzeichen enthalten, auf die beim Ausführen einer Unterbrechung automatisch von der Hardware umgeschaltet wird, gilt diese Einschränkung nicht, und man kann verschiedene privilegierte Gerätetreiber durchaus in verschiedenen Adressräumen oder Schutzbereichen betreiben, was die Stabilität eines Systems auch im Falle instabiler Hardware positiv beeinflussen kann.

5. Callbacks durch `notify`-Operationen

Hierarchische Beziehungen drücken ein Abhängigkeitsverhältnis aus. Baustein-Instanzen, die „höher“ in einer Hierarchie stehen, haben „Macht“ oder „Kontrolle“ über solche, die weiter unten stehen. Die Leitungen des Baustein-Modells drücken derartige Hierarchie-Beziehungen aus.

Es gibt jedoch Fälle, in denen sich eine eindeutige Aufruf-Hierarchie nicht immer reinrassig ausführen lässt. Betriebssystem-Konstrukteure versuchen traditionell, solche Fälle möglichst zu vermeiden. Im Idealfall sollte ein Betriebssystem durch ein Schichtenmodell beschreibbar sein (vgl. [Dij71]). Die `notify`-Operationen sind dazu gedacht, um ein Schichtenmodell auch dann noch beibehalten zu können, wenn es aus irgendwelchen Gründen klemmen würde (vgl. [Cla85]). In dieser Sichtweise haben `notify`-Operationen einen *Ausnahmecharakter*, der nicht missbraucht werden sollte. Ich werde versuchen, diejenigen Anwendungsfälle zu begründen, in denen der Einsatz von `notify`-Operationen sinnvoll oder gar notwendig ist.

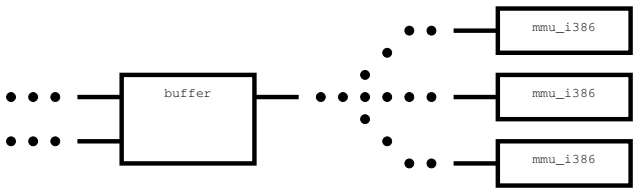
5.1. Eigentums- und Besitzverhältnisse

Die Begriffe *Eigentum* und *Besitz* stammen aus dem Rechtswesen. Dort kennzeichnen sie den Erstreckungsbereich eines Rechtes: wer *Eigentümer* einer Sache ist, der hat weiter gehende Rechte als der *Besitzer*. Es ist insbesondere möglich, Sachen *auszuleihen* oder zu *vermieten*; dabei bleibt man weiterhin Eigentümer, obwohl der Besitzer (kurzzeitig) wechselt. Der neue Besitzer kann transitiven *Unterbesitz* an einer Sache oder Teilsache an ein oder mehrere *Unterbesitzer* weiter geben; diese *schulden* die Rückgabe an den *Hauptbesitzer*, das ist derjenige, von dem sie den Unterbesitz erhalten haben; dieser schuldet wiederum die Rückgabe des Besitzes an den Eigentümer. Der Eigentümer ist jemand, der niemandem die Rückgabe des Besitzes schuldet[1]. Im realen Leben kommen solche Rechtsverhältnisse z.B. bei der Vermietung und Untervermietung von Wohneigentum vor. Ein Mieter einer 5-Zimmerwohnung kann z.B. jedes Zimmer einzeln an Studenten untervermieten.

Diese Analogie möchte ich zur Darstellung der Verhältnisse in einem Betriebssystem einsetzen; dabei werde ich als Beispiel das Eigentum und den Besitz an Datenblöcken und an Locks nehmen; diese Methodik ist auch auf andere Ressourcen bzw. Objektarten übertragbar.

Eigentums- und Besitzverhältnisse sind von einer Hierarchie unabhängig[2] und können sich dynamisch zur Laufzeit ändern. Eine Baustein-Hierarchie teilt die Gesamtverantwortung eines Betriebssystems in Zuständigkeiten auf; bei der Übertragung eines Besitzes wird dagegen die temporäre Verantwortung für die zu verwaltenden Objekte innerhalb der Hierarchie weiter gegeben.

Bei physischen Datenblöcken gibt es einen *natürlichen Eigentümer*: das ist diejenige Instanz, die die physische Adresse festlegt und verwaltet, z.B. indem sie den Datenblock „erzeugt“, d.h. in Existenz bringt[3]. In der Regel stellt dieser natürliche Eigentümer den Datenblock her, um ihn anschließend zu „verleihen“, d.h. das *Zugriffs-* oder *Besitzrecht* an eine andere Instanz für eine bestimmte Zeit abzutreten. Der neue Besitzer kann im Allgemeinfall hierarchisch tiefer oder höher stehen. Unabhängig von der Hierarchie aber muss er den Besitz irgendwann wieder an den Eigentümer zurückgeben, ansonsten droht das bekannte Problem der Ressourcenverknappung durch nicht mehr recycelbare Ressourcen. Das folgende Bild zeigt einen Ausschnitt aus einem Szenario eines Rechners, der mit MMU-Hardware ausgestattet ist:



Prinzipiell kommt als Eigentümer oder Hauptbesitzer von Datenblöcken jede der beteiligten Baustein-Arten in Frage. Es bringt jedoch Vorteile, wenn die Adressvergabe der physischen Datenblöcke von der Adressvergabe im jeweiligen logischen Adressraum getrennt wird. Bei den logischen Adressen gibt es einen *notwendigen* Eigentümer bzw. Hauptbesitzer, der darüber zumindest in den Grundzügen bestimmen können muss: das ist im Regelfall der Benutzer, der ganz hinten als Endkonsument sitzt und bestimmen möchte, was der Rechner und das Betriebssystem für ihn tun soll; er kann zwar einige dieser Aufgaben an den Compiler oder das Laufzeitsystem (Benutzer-Heap-Verwaltung) delegieren, aber letztlich ist er der Herr über seinen virtuellen Adressraum.

Physische Adressen lassen sich prinzipiell sowohl ganz oben bei den `mmu_*`-Instanzen als auch ganz unten bei der `buffer`-Instanz bzw. der evtl. vorgeschalteten `device_mem`-Instanz verwalten. Bei Vorhandensein von MMU-Hardware ist es jedoch aus mehreren Gründen vorteilhaft, die Verwaltung phyischer Adressen und damit das Eigentumsrecht der `buffer`- bzw. `device_mem`-Instanz und nicht den vielen `mmu`-Instanzen zu übertragen:

- Verschiedene `mmu`-Instanzen greifen häufig auf dieselben Datenblöcke zu; beispielsweise gemeinsamer Maschinencode. Wenn das Eigentumsrecht bei der

[1] Die einzige Ausnahme ist der Tod eines Eigentümers.

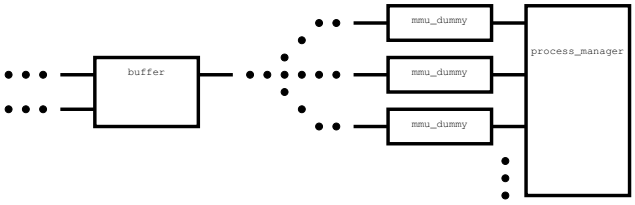
[2] Beispielsweise kann auch in einer absoluten Monarchie ein Untertan Eigentum erwerben, auf das der König keinen Zugriff hat. Wenn dies nicht möglich wäre, müssten die Untertanen verhungern, da sie ohne Zustimmung des Königs kein Eigentum und keinen Besitz an Lebensmitteln erwerben könnten, auch nicht an solchen Lebensmitteln, die sie selbst hergestellt haben.

[3] Beim Start des Betriebssystems gehört sämtlicher Speicher einem Urbesitzer, der ihn dann an andere Instanzen wie Caches oder `device_ramdisk` „ausleiht“.

buffer-Instanz liegt, ist die gemeinsame Nutzung von Datenblöcken relativ einfach zu realisieren. Billigt man das Eigentum dagegen den mmu-Instanzen zu, dann findet entweder keine gemeinsame Nutzung statt, oder diese muss wesentlich aufwendiger z.B. über ein Cache-Kohärenz-Protokoll hergestellt werden, das sich über die gesamte Hierarchie hinweg erstrecken kann.

- Die Gefahr von Deadlocks ist leichter vermeidbar, da eine zentrale Instanz besser die Übersicht über vergebene Ressourcen wie Locks behalten kann.

Im folgenden Beispiel ist keine MMU-Hardware vorhanden. Die Verwaltung der physischen Datenblock-Adressen *muss*(!) wegen der notwendigen Übereinstimmung von logischen und physischen Adressen bei den mmu_dummy-Instanzen erfolgen, die als Ersatz für die fehlende MMU-Hardware stehen:



Wenn keine MMU vorhanden ist, dann ist eine unabhängige Übersetzung von logischen Adressen nach physischen Adressen nicht möglich; es bleibt als einzige Lösung, die physischen Adressen mit den logischen Adressen gleichzusetzen. Daher ist es günstiger, das Eigentumsrecht zumindest an denjenigen Datenblöcken, die in Prozessabbildern verwendet werden, ganz oben in der Hierarchie bei mmu_dummy oder process_manager zu verwalten (andernfalls könnten keine Transformationen durch dazwischen liegende dir_*-Instanzen durchgeführt werden, oder es müssten zusätzliche Performanz fressende Kopien erstellt werden[4]).

Diese Erörterung sollte bewusst gemacht haben, dass Eigentums- und Besitzverhältnisse prinzipiell unabhängig von Hierarchie-Beziehungen sind, und dass eine gut durchdachte Zuordnung Implementierungs-Aufwand sparen und Performanz verbessern kann.

Die in Abschnitt 3.1 eingeführte Unterscheidung zwischen logischem und physischem IO auf Nestern korrespondiert mit dem Konzept der Eigentums- und Besitzverhältnisse auf folgende Weise:

- Ein Eigentümer bzw Hauptbesitzer bietet stets logischen IO an seinem Ausgang an. Die höheren Hierarchie-Instanzen sind gezwungen, logischen IO zu verwenden, wenn sie in den (temporären) Besitz eines Datenblocks gelangen wollen; zur Verwaltung der „Ausleih-Rechtsverhältnisse“ kann z.B. der in Abschnitt 3.3.3 eingeführte Referenzzähler dienen; im juristischen Kontext würde man dies als „Schuldverhältnis“ charakterisieren. Ein „Schuldner“ muss irgendwann seine „Schulden“ wieder „zurückzahlen“; dies geschieht im Beispiel-Entwurf durch korrekte Paarung mit put.
- Ein Eigentümer betreibt an seinem Eingang ausschließlich physischen IO; die unterliegenden Instanzen brauchen nur diese Betriebsart zu unterstützen.

Es ist jedoch sinnvoll, dass dazwischen liegende Instanzen wie dir_* sowohl die Kompetenzen zu logischem als auch zu physischem IO haben sollten. Dies ermöglicht nicht nur den Einsatz der Baustein-Architektur in MMU-freier Hardware wie *embedded systems*, sondern rein theoretisch sogar den gemischten Einsatz von solchen Prozessen, die in virtuellen Adressräumen laufen, neben MMU-übersetzungsfreien Prozessen[5].

5.2. Rückforderung von Eigentum

Wie im realen Leben kann es auch in einem Betriebssystem vorkommen, dass ein „Schuldner“ seine „Schulden“ nicht bezahlt, oder nicht „rechtzeitig“ bezahlt. Die Einführung eines „Mahnwesens“ ist prinzipiell geeignet, den Systemdurchsatz zu verbessern oder in hoffnungslosen Mangelsituationen überhaupt noch ein Weiterarbeiten zu ermöglichen.

```
notify_data(nest,log_addr,len,
            phys_addr,urgency) → success
notify_lock(nest,log_addr,len,
                    try_address,try_len,
                  kind,urgency) → success
```

Diese Operationen durchlaufen eine Baustein-Hierarchie in umgekehrter Richtung, d.h. vom Eigentümer hin zu den Konsumenten. Die Konsumenten werden gebeten oder gezwungen, die jeweilige Ressource (ein Datenblock mit der angegebenen logischen und physischen Adresse, oder ein Lock der angegebenen Art) zurückzugeben. Der Parameter urgency kann einen der folgenden Werte annehmen:

ask	Es wird lediglich angefragt, ob eine Rückgabe der Ressource ohne größere Mühen und Störung des Betriebes möglich wäre. Der Empfänger braucht außer der boolschen success-Rückmeldung nichts zu unternehmen.
try	Der Empfänger wird gebeten, die Ressource zurückzugeben, wenn es den Betrieb nicht unverhältnismäßig stört. Ein genauer Zeitpunkt für die Rückgabe wird nicht vorgeschrieben.
command	Der Empfänger wird aufgefordert, die Rückgabe unverzüglich (d.h. noch vor der success-Rückmeldung) vorzunehmen. Eine Verweigerung der Rückgabe durch Rücklieferung von success=false ist nur dann statthaft, wenn der Empfänger wegen der fehlenden Ressource seine Tätigkeit ganz einstellen müsste.

[4] Wenn man Kopien herstellt, dann betreibt man letzlich eine Mischform der Zuordnung von Eigentumsrechten an verschiedene Instanzen. Die „Originale“ werden unten in der Hierarchie verwaltet (siehe MMU-freies Ur-Unix mit Buffer-Cache in [Lio96]), die Kopien entstehen dann implizit z.B. bei read- oder write-Operationen.

[5] Wieviel Sinn dies macht, ist eine andere Frage. Theoretisch kann ein MMU-freier Prozess schneller laufen, da der Hardware-Aufwand zur Adressübersetzung wegfällt, da garantiert keine Verzögerungen durch Seitenfehler-Unterbrechungen auftreten. Letzterer Effekt wird in echtzeitfähigen Betriebssystemen jedoch auch durch vorgeladene und nicht auslagerbare Speicherseiten erzielt; lediglich der Aufwand durch TLB-Misses kann wegfallen.

`force` Der Empfänger muss die Ressource unverzüglich zurückgeben, auch wenn er dadurch zum Abbruch seiner Tätigkeit gezwungen wird. Falls er sich dennoch weigern sollte[6], werden alle seine Ressourcen „konfisziert“, d.h. vom Eigentümer so behandelt, als wären sie zurückgegeben worden, und es finden im Zukunft keine Operations-Ausführungen mehr statt, die durch die Empfänger-Instanz in Auftrag gegeben worden sind oder noch gegeben werden.

Die `notify_*`-Operationen haben den Charakter einer *Ausnahme* und sind konzeptuell mit den Signalen von Unix und anderen Betriebssystemen vergleichbar. Prinzipiell sind daher *Wettrennen* zwischen `notify_*`-Operationen und Anforderungs- bzw. Rückgabe-Operationen des Besitzers möglich. Diese Wettrennen treten auch in vergleichbaren Situationen von konventionellen Betriebssystemen auf und sind meiner Ansicht nach *prinzipbedingt* (siehe Diskussion in Anhang C). Wettrennen lassen sich zwar durch architekturelle Maßnahmen prinzipiell vermeiden (vgl. Anhang C), jedoch geht dies i.a. zumindest bei verteilten Systemen zu Lasten der erzielbaren Parallelität und Performanz, da zumindest dort das Wettrenn-Problem inhärent ist und ohnehin gelöst werden muss.

Eine vom Wettrenn-Problem unabhängige Frage betrifft die „rechtmäßige“ Ausübung „brutaler Gewalt“, insbesondere der `force`-Modus. Es versteht sich von selbst, dass ein Eigentümer oder Hauptbesitzer große Macht über jeden ausüben kann, der etwas von ihm als Unterbesitz erhalten hat, da er auch dann auf Rückgabe bestehen kann, wenn dadurch die Existenz des Unterbesitzers gefährdet wird. Da die Besitzschuldverhältnisse prinzipiell unabhängig von den in Abschnitt 2.7 besprochenen Zugriffsrechten sind, könnte man die Frage stellen, ob diese Machtstellung insbesondere von hierarchisch untergeordneten Instanzen nicht unangemessen sein könnte. Hierauf gibt es eine prinzipielle Antwort: Nein. Wer (auf welche Weise auch immer) einmal die Macht erhalten hat, bestimmte Ressourcen verwalten zu dürfen, der kann von dieser Macht bereits bei der Ressourcenvergabe unangemessenen Gebrauch machen. Er kann rein theoretisch bereits zu diesem Zeitpunkt „bösartig“ handeln und eventuelle Zugriffsbeschränkungen umgehen; ein typisches Beispiel ist die Hauptspeicherverwaltung, die bei Kompromittierung sämtliche Schutzmechanismen des gesamten Systems aushebeln kann, die auf ihr basieren[7]. Daher muss man einem Ressourcenbesitzer im Hinblick auf die Einhaltung geforderter Zugriffsrechts-Beschränkungen ohnehin vertrauen. Das Recht zur Rückforderung benötigt in dieser Sichtweise kein *zusätzliches* Vertrauen. Wenn man einer Instanz nicht vertrauen kann, dann darf man sie von vornherein nicht mit der Verwaltung von Eigentum oder Unterbesitz beauftragen (z.B. indem man für eine geeignete Aufteilung der Macht sorgt).

Es gibt weitere Gründe, die den Entzug beliebiger Ressourcen zu einem Pflicht-Mechanismus in zukunftssicheren Betriebssystemen machen. Großrechner unterstützen seit langem das so genannte „Hot-Plugging“ von Hardware-Komponenten, also den Austausch mitten im laufenden Betrieb. Dies betrifft nicht nur CPUs, die in fast allen Betriebssystemen entziehbare Ressourcen darstellen, sondern auch Speicher-Module. Die Problematik des Laufzeit-Austausches taucht fernerhin im IO-Bereich wie z.B. bei RAID auf; bei sicherheitskritischen Anwendungen wird ein prophylaktischer Austausch von Hauptspeicher-Modulen manchmal sogar turnusmäßig durchgeführt. Die Fähigkeit zum Austausch im laufenden Betrieb erfordert einen Mechanismus, mit dem der Entzug von Ressourcen in geordneten Bahnen durchgeführt werden kann.

Microsoft hat das Konzept der *opportunistischen Locks* [OpL] in die Betriebssystem-Schnittstelle der neueren Windows-Versionen integriert und damit zumindest für einige Anwendungen auch Anwendungsprogrammen verfügbar gemacht. Opportunistische Locks[8] sind dazu gedacht, um das Cache-Kohärenz-Problem in verteilten System durch den *gezielten* Entzug von Ressourcen zu beschleunigen; wenn kein aktueller Bedarf für einen Entzug vorhanden ist, dann wird auf das zeitaufwendige Durchreichen der Ressource über das Netzwerk verzichtet („aggressives Caching“). Im hier vorgestellten Modell wird der Entzug mittels brutaler Gewalt nur als *letztes Mittel* betrachtet; für das im nächsten Abschnitt näher behandelte Cache-Kohärenz-Problem wird der `try`-Modus favorisiert. Ein weiterer grundlegender Unterschied der hier favorisierten spekulativen Locks zu opportunistischen Locks besteht darin, dass auf Rückforderungen im `try`-Modus nicht mit der Rückgabe des *gesamten* gesperrten Bereiches, sondern auch nur eines Teiles reagiert werden kann; dies ist insbesondere in Kombination mit der spekulativen Anforderung von vergrößerten Adress-Bereichen durch die `try_address`- und `try_len`-Parameter vorteilhaft (vgl. Abschnitt 3.3.5). Die Unterscheidung zwischen einem Kernbereich und einem spekulativ erweiterten Bereich wird auch in `notify_lock` getroffen; damit kann der Empfänger einer Rückforderung selbst entscheiden, wieviel er ohne größere Behinderung seiner Arbeit zurückgeben kann.

5.3. Synchronisation zwischen Kopien: spekulative Locks

Ausführlichere Beschreibungen der in diesem Abschnitt vorgestellten Methodik finden sich in [ST04c, ST04a].

Verteilte Systeme unterscheiden sich aus Sicht der hier vorgestellten Architektur hauptsächlich[9] durch eine Ei-

[6] Wenn der Empfänger z.B. in eine Endlosschleife geht oder bei der Ausführung der `notify_*`-Operation in einen Deadlock gerät oder wenn z.B. im RPC-Modell der Bearbeiter-Kontrollfluss verstirbt, kann eine „Weigerung“ auch schlicht darin bestehen, dass nichts geschieht. Da interne Angelegenheiten eines Bausteins nicht von außen „aufgemischt“ werden sollten, bleibt zur Lösung dieses Problems letztlich nur das Setzen einer Zeitschranke, deren Ablauf als „Weigerung“ interpretiert wird.

[7] Sofern sie nicht durch andere Instanzen wie z.B. Wächter daran gehindert wird. Dies ist jedoch kein Gegenargument, da eine Wächter-Instanz konsistenterweise auch die Rückforderungen überwachen sollte, so dass im Endeffekt eine Aufteilung der Macht auf mehrere Instanzen erfolgt (gegenseitige Überwachung).

[8] Microsofts opportunistische Locks sind stellen eine eigene Locking-Art dar, die von anderen Locking-Arten unabhängig sind. Dies erhöht nicht nur die Komplexität der Schnittstellen, sondern kann z.B. zu der Situation führen, dass eine maliziöse Anwendung, die einen opportunistischen Lock entzogen bekommen hat, mittels regulärer Locks das System dennoch lahm legen kann.

[9] Das häufig angeführte Problem der *Ausfallsicherheit* sehe ich nicht als spezifisch für verteilte Systeme an, obwohl eine brauchbare Lösung dort von besonderer Wichtigkeit ist. Konzepte zur Erzielung von Ausfallsicherheit sollten für alle Bausteine uniform sein, unabhängig von

genschaft von einem allein stehenden Rechner: das ist die Zugriffs- bzw Latenzzeit der remote- und mirror-Bausteine, sowie die maximale Datenrate, die ihren Durchsatz begrenzt. Dieses Problem tritt in Gestalt der bekannten „Speicherlücke“ auch in allein stehenden Systemen auf und ist als so genanntes *Flaschenhals-Problem* lange bekannt, das sich z.B. bei virtuellem Speicher als *Thrashing* (vgl. [DS72]) äußern kann. Die spezifische Problematik steckt bei verteilten Systemen lediglich im *Ort* (Hierarchie-Ebene), an dem sich der Flaschenhals befindet.

Zur Lösung des Flaschenhals-Problems gibt es zwei erprobte grundlegende Strategien: man versucht den Flaschenhals durch Hardware-Aufrüstung zu erweitern, und/oder die zur Lösung einer Aufgabe notwendige Belastung zu senken. Während sich bei der Hardware klassischer Peripheriegeräten in der Vergangenheit ständige Fortschritte sowohl bei den Latenzzeiten als auch beim Durchsatz ereignet haben und voraussichtlich weiter ereignen werden, sind derartige Fortschritte bei räumlich weit verteilten Systemen nur beim Durchsatz, nicht hingegen bei der Latenzzeit möglich (vgl. [TL93]), da die Kommunikation von der Lichtgeschwindigkeit begrenzt wird. Daher ist unnötiger Datenverkehr bzw. unnötiges Warten auf die Ausführung von Operationen bei verteilten System noch dringender zu vermeiden als in alleinstehenden Systemen. Die hier vorgestellte Architektur legt auch aus diesem Grund besonderen Wert auf asynchrone Kommunikation.

Wie bereits in Abschnitt 4.1.2 (Baustein buffer) dargestellt, besteht der Zweck eines Caches in der Entkoppelung von Flaschenhälsen und der Reduktion der darüber laufenden transfer-Operationen. Während bei alleinstehenden Rechnern Konfigurationen mit nur einer zentralen buffer-Instanz möglich sind, ist der Einsatz kaskadierter verteilter Caches (buffer-Instanzen) bei verteilten Systemen aus Performanz-Gründen de facto unumgänglich. Kaskadierte Caches führen zwangsläufig zum bekannten Cache-Kohärenz-Problem.

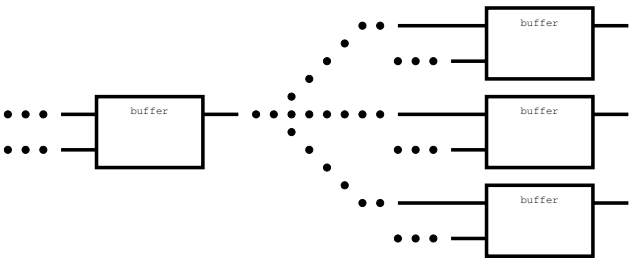
Das Cache-Kohärenz-Problem lässt sich in der Systematik der hier verwendeten Begriffe auf folgende Weise formulieren: es können *mehrere Besitzer* derselben *logischen Ressource* vorhanden sein.

Im Rechtswesen taucht dieses Phänomen in den Begriffen *Gemeinschaftseigentum* (z.B. bei Wohnungseigentümer-Gesellschaften) und *gemeinschaftlicher Besitz* (z.B. Anmietung durch einen Verein) auf. Grundsätzlich sind mehrere Besitzer gleichberechtigt, und sie müssen sich bei allen Fragen betreffs des Besitzes gegenseitig *auseinandersetzen*; dies geschieht im so genannten *Innenverhältnis*. Der Grund hierfür ist, dass ein Besitz exklusiv nutzbare Eigenschaften haben kann, so dass eine gleichmäßige und/oder gleichzeitige Nutzung durch alle Besitzer nicht immer möglich ist.

Die Auseinandersetzung über exklusiv nutzbare Ressourcen findet in traditionellen alleinstehenden Betriebssystemen bevorzugt durch Delegation der Verwaltungsaufgaben an eine zentrale Instanz (z.B. ein Lock-Manager) statt. Diese Lösung hat den Vorteil hoher Einfachheit bei geringen Kosten, da durch die Delegation praktisch vernachlässigbare Latenzen entstehen. In einem verteilten System ist dagegen eine Delegation an eine einzige Instanz i.a. wegen der Kommunikations-Latenzen sehr kostenintensiv.

der konkreten physischen Verteilung.

Man kann sich Worst-Case-Zugriffsszenarien ausdenken, bei denen die Latenz-Kosten eines verteilten Systems *zwangsläufig bei jedem einzelnen Zugriff* entstehen müssen, weil sie wegen einer „genügend bösartigen“ Aufgabenstellung nicht zu umgehen sind (analog zum Working-Set-Modell). Dies gilt auch für Caches alleinstehender Systeme, die man ebenfalls mit solchen Zugriffsmustern belasten kann, dass sie den bekannten *Thrashing-Effekt* zeigen, bei dem sie ihre Wirksamkeit fast vollständig einbüßen können. Bei alleinstehenden Systemen gilt die lange bekannte Grundregel, dass Thrashing nur dann vermeidbar ist, wenn der *Working Set* [Den68] der Last in den Cache passt. Diese lange bekannte Grundregel gilt aber auch in einem verteilten System mit mehreren Caches, allerdings in nochmals verschärfter Form. Schauen wir uns dazu das folgende Szenario aus mehreren örtlich weit verteilten Caches an, die ich „hierarchisch oben stehende Caches“ nenne, die an einen gemeinsamen Basis-Cache angeschlossen sind:



Damit ein Entkopplungs-Effekt durch die oben stehenden buffer-Instanzen entstehen kann, müssen ihre Working-Sets, bezogen auf die zeitliche Größenordnung der Kommunikations-Latenzen, „einigermaßen disjunkt“ sein. Andernfalls tritt analog zu [DS72] zwangsläufig ein Thrashing-Effekt *zwischen* den oben stehenden buffer-Instanzen ein, der in der hier gewählten Hierarchie-Struktur über die unten stehende Basis-buffer-Instanz abgewickelt wird und auch durch Vergrößerung der Caches nicht eliminierbar ist. Gegen diese Art von Thrashing hilft meiner Ansicht nach nichts: es ist verteilten Systemen inhärent, dass modifizierte Daten irgendwie an den Ort der anderen buffer-Instanz gebracht werden müssen, wenn sie dort von der Aufgabenstellung her unbedingt benötigt werden, und das kostet zwangsläufig Latenzzeit (weil die Lichtgewindigkeit als Naturkonstante nicht umgehbar ist). Ich sehe daher keine andere Möglichkeit, als die Aufgabenstellung so zu gestalten, dass ein derartiges Thrashing nicht stattfindet. Dazu ist eine notwendige Bedingung, dass die Working-Sets an verschiedenen Orten einigermaßen disjunkt voneinander sein müssen.

Unter der Annahme, dass die Working-Sets der oben stehenden buffer-Instanzen disjunkt sind, sieht die Lösung des Problems folgendermaßen aus: es ist nichts anderes zu tun, als die vorkommenden Working-Sets einigermaßen genau zu bestimmen. Dies macht ein ausschließlich von Anforderungen getriebener Cache von Natur aus ganz von selbst. Anders sieht die Situation höchstens dann aus, wenn das Laden der oberen Cache-Instanzen mit Daten zusätzlich auf *spekulative* Weise erfolgt, z.B. beim so genannten *Preloading*.

Zur Analyse des Preloadings sehen wir uns den Best Case an, bei dem wir die folgenden idealisierten Annahmen treffen:

1. Die Caches seien jeweils genügend groß, so dass alle insgesamt benötigten Datenmengen in jedem einzelnen Cache Platz haben.

2. Die Datenübertragungs-Rate sei näherungsweise unendlich groß, die Latenzzeit dagegen ein fester Wert echt größer Null.

3. Jegliche Änderung eines Datums werde *sofort* an alle anderen Cache-Instanzen propagiert (was wegen der unendlichen Datenübertragungs-Raten auch kein Problem darstellt); diese Propagation kostet näherungsweise nichts.

4. Welche Version eines Datums während der Kommunikations-Latenzzeiten an welcher Stelle zur Verfügung steht, wird idealisierend als ständig überall bekannt vorausgesetzt.

Trotz dieser bestmöglichen Annahmen, die unter der vorgegebenen räumlichen Verteilung nicht mehr zu verbessern sind, kann hier immer noch ein Thrashing zwischen den oben stehenden Caches stattfinden, sofern man nur die Anforderungen durch die Aufgabenstellung genügend „bösartig“ gestaltet. Ist diese Bösartigkeit jedoch nicht gegeben (weil die Working Sets aller Teilnehmer-Instanzen disjunkt sind), dann braucht *niemals* irgend eine `buffer`-Instanz auf eine andere zu warten. Damit haben wir den Best Case dessen, was mit Hilfe von Caching-Strategien erreichbar ist, nämlich die totale Entkopplung aller Aktivitäten und damit die Illusion, dass Latenzzeiten nicht vorhanden seien.

Da die getroffenen Annahmen in der Praxis natürlich nicht zutreffen[10], lautet die Frage, wie man den Best Case approximieren kann. Für die Annahme 1 gibt es lange bekannte und erprobte Strategien der Adaption von Caches an die Working-Sets der Anforderungen, insbesondere LRU und LFU. Zur Lösung des Problems 2 wird insbesondere in Hardware-Architekturen (beispielsweise in Prozessor-Cache-Hierarchien) die *explizite Priorisierung* der Datentransfers eingesetzt: beim Transport durch einen Flaschenhals müssen solche Transport-Aufträge dringender behandelt werden, von deren Ausführung andere Aktivitäten abhängen; dagegen sollten spekulative Transport-Aufträge unbedingt geringer priorisiert werden, um dringendere Aktivitäten nicht durch Verstopfung des Flaschenhalses zu behindern. Dies wird beim hier vorgestellten Entwurf durch die IO-Prioritäten ermöglicht (siehe Abschnitt 2.8.3). Das Problem 3 lässt sich dadurch lösen, dass jegliche Änderungen grundsätzlich *sofort*[11] durch Erzeugen eines IO-Auftrages (`transfer`-Operation) bekannt gemacht werden, allerdings standardmäßig nur mit Background-IO-Priorität, die nur bei Vorliegen höherer Dringlichkeit erhöht wird. Das Problem 4 erfordert eine genauere Analyse:

Wie bereits in Abschnitt 3.3.3 dargestellt und in Abschnitt 5.1 begründet, erfordert die Verwaltung von Besitz eine transiente Zuordnung des Besitz-Status (z.B. als Referenzzähler, ggf. auch der Versions-Zuordnung). Dieser Status ist an den *Ort* gekoppelt, für den der jeweilige (Unter-)Besitz-Status verwaltet wird. Da alle oben stehenden Caches aus Sicht des Basis-Cache Konsumenten sind, müssen sich diese wie alle anderen parallel arbeitenden Konsumenten an ein `multi*`-Verhalten halten, um die Determinanz für ihre jeweiligen Konsumenten sicherstellen zu können. Daraus folgt inbesondere, dass Locks prinzipiell bis auf den Basis-Cache durchgereicht werden müssen, ansonsten könnten die Verhaltens-Modelle verletzt werden. Wenn man jeden Lock einzeln durchreichen würde, dann entstünde jedesmal eine Wartezeit durch die Kommunikations-Latenzen, was sich katastrophal auf den Durchsatz auswirken würde. Diese Pflicht zum Durchreichen ist jedoch eine *Mindest-Anforderung*, die auch *übererfüllt* werden darf. Die Idee besteht nun darin, dass bei der erstmaligen Anforderung eines Locks nicht die vom Endkonsumenten angeforderte Granularität durchgereicht wird, sondern möglichst ein *spekulativ vergrößerter* Adressbereich (z.B. ein gesamtes Unter-Nest oder ein gesamtes Verzeichnis-Nest). Ein dazu geeigneter Mechanismus ist bereits in Abschnitt 3.3.5 vorgestellt worden (Parameter `try_address` und `try_len`). Wenn es einem der oben stehenden Caches gelungen ist, den Working Set seiner Konsumenten einigermaßen realistisch spekulativ einzuschätzen (was leicht geht, wenn die Working-Sets wie häufig beobachtet aus wenigen zusammenhängenden Bereichen bestehen), dann braucht er spätere Lock-Anforderungen seiner Konsumenten nicht jedesmal Zeit raubend zum Basis-Cache durchzuschalten, sondern kann den bereits erhaltenen spekulativen Besitz ohne Wartezeit an seine Konsumenten als Unterbesitz weitergeben, sobald diese eine entsprechende Anforderung stellen. Zu beachten ist dabei lediglich, dass spekulativ vorab angeforderte Lock-Arten die korrekte Semantik bei späterer Nachforderung durch die Endkonsumenten ergeben müssen.

Spekulationen über den zukünftigen Working-Set der Endkonsumenten können insofern fehl gehen, als dass zu große Bereiche vom Basis-Cache angefordert wurden, die später doch noch von anderen obenstehenden Caches benötigt werden[12]. Fehlspekulationen lassen sich durch Rückforderungen mittels `notify_*`-Operationen korrigieren. Der Basis-Cache kann grundsätzlich bei Anforderung bereits vergebener Locks entsprechende `notify_*`-Operationen an den/die momentane(n) Besitzer-Instanz(en) generieren. Falls er sich die Unterschiede zwischen vergebenen Kern- und Erweiterungsbereichen merkt (sowie ggf. von später hinzu kommenden Kernbereichen durch den neuen Besitzer zeitnah, aber asynchron informiert wird), kann er auf das Stellen „aussichtsloser“ Rückforderungen, die vergebene Kernbereiche betreffen, auch verzichten. Rückforderungen von Kernbereichen sollten möglichst nur für solche Adressbereiche gestellt werden, die ein anderer Cache auch wirklich aktuell benötigt. Es ist Sache des Fehlspekulanten, ggf. auch größere Bereich als den aktuell zurückgeforderten oder sogar größere als den vorgeschlagenen Erweiterungsbereich an den Basis-Cache zurückzugeben, um damit weiteren zukünftigen Rückforderungen spekulativ vorzubeugen. Eine Rückgabe ist darüber hinaus auch jederzeit spontan möglich, beispielsweise wenn erkannt wurde, dass ein bestimmter Bereich auf jeden Fall nicht mehr benötigt

[10] Die Fortschritte bei den Kommunikationstechniken gehen jedoch in die Richtung, dass die Best-Case-Annahmen immer besser durch verbesserte Technik angenähert werden.

[11] Damit ist nicht die Granularität einzelner Schreibe-Maschinenoperationen gemeint, sondern eine „zeitnahe“ (asynchrone) Meldung in der Größenordnung der Kommunikations-Latenzen.

[12] Falls die Annahme disjunkter Working Sets doch nicht zutreffen sollte, liegt bei der hier vorgestellten Betrachtungsweise die „Schuld“ nicht bei den Caches, sondern beim Verhalten der Endkonsumenten.

wird.

Insgesamt bewirkt diese Vorgehensweise eine spekulative Vorab-Verteilung von Ressourcen, die durch die aktuellen Anforderungen nachträglich korrigiert wird. Die Rückgabe spekulativ angeforderter Ressourcen kann durch verschiedene Strategien erfolgen. Eine mögliche Rückgabe-Strategie ist, den zwischen dem nächstliegenden tatsächlich selbstgenutzten Bereich und dem zurückgeforderten Bereich liegenden spekulativ angeforderten Bereich zu halbieren, also die Hälfte weiter zu behalten, die andere Hälte dem Rückforderer zu überlassen. Falls die Working Sets der Endkonsumenten disjunkt und näherungsweise zusammenhängend sind, dann ergibt sich dadurch im Worst Case eine logarithmische Anzahl von Rückforderungen (in Abhängigkeit von der Größe des aufzuteilenden Adressbereiches). Nach dieser höchstens logarithmischen Anzahl an Rückforderungen ist der vorab unbekannte genaue Verlauf der disjunkten Working Sets empirisch festgestellt worden, und es finden keine weiteren Rückforderungen mehr statt (eingeschwungener Zustand des Systems). Nach dem Einschwingen sind keine weiteren Synchronisationen mehr erforderlich, sofern sich die Working Sets nicht ändern; damit ist der obige Best Case relativ gut approximiert. Bei „langsamen“ Änderungen der Working Sets finden zwar ab und zu noch Rückforderungen statt, diese haben jedoch ein relativ geringes Gewicht in den Gesamtzahl aller Operationen; damit ist hier ebenfalls der Best Case relativ gut approximiert. Bei zu starker Überlappung der Working-Sets bzw zu großer Änderungsrate kann es zu einem Thrashing kommen, das jedoch laut obiger Argumentation prinzipiell nicht vermeidbar ist.

6. Generische Operationen

Generische Operationen sind Operationen, die über eine *polymorphe Schnittstelle* aufrufbar sind.

In Betriebssystemen wird das Paradigma der universellen Generizität bzw. der Polymorphie bereits sehr lange eingesetzt; bekannte Beispiele finden sich u.a. bei polymorphen Kern-Schnittstellen wie `ioctl` in Unix, oder bei den Operations-Sprungtabellen von IBM-Betriebssystemen, die verschiedene Arten von Polymorphie bereits in den 1960er Jahren implementierten.

Bevor wir zur Untersuchung der Arten von Polymorphie kommen, möchte ich klären, was ein Operations-Aufruf ist:

Ein *Operations-Aufruf* ist die Übergabe von *Information* von einer *Aufrufer-Instanz* an eine *Bearbeiter-Instanz*.

Nach dieser Definition ist die Rückgabe von Ergebnissen, z.B. in Ergebnis-Parametern, ebenfalls ein Operations-Aufruf, nur in umgekehrter Richtung, d.h. vom ursprünglichen Bearbeiter zurück zum ursprünglichen Aufrufer. In diesem Fall spricht man von einem *Operations-Aufrufs-Paar*.

Operations-Aufrufe bzw -Paare lassen sich auf verschiedene Weisen realisieren. Bekannte Realisierungs-Arten sind z.B. Prozeduraufrufe mittels Übergabe von Parametern auf dem Stack eines Programmiersprachen-Laufzeitsystems, oder der RPC (Remote Procedure Call). Ersterer Fall lässt nur den *synchronen Aufruf*, letzterer Fall auch den *asynchronen Aufruf* zu. Weitere bekannte Realisierungs-Arten sind das Starten von Kontrollflüssen (asynchrones Verhalten), oder die Weitergabe der Flusskontrolle in einem Coroutinen-Modell durch Aufruf einer Transfer-Operation (dies bewirkt *synchrones* Verhalten aus Sicht des Gesamtsystems).

Ich möchte auf eine weitere bekannte Art der Realisierung von Operations-Aufrufen fokussieren: das so genannte *Nachrichten-Paradigma*[1], das zwar oft im Kontext von „communicating sequential processes“ (CSP, siehe [Hoa78]) in Erscheinung tritt, prinzipiell jedoch unabhängig von sequentiellen Kontrollflüssen ist. Das Nachrichten-Paradigma wird grundlegend in der Hardware verwendet und steckt z.B. hinter den weit verbreiteten read-write-Schnittstellen, bei denen ja auch Informationen von einer Instanz an eine andere logisch weitergegeben werden.

Man sollte sich unbedingt klar machen, dass zwischen einer reinen Nachrichten-Übertragung und einem Operations-Aufruf *kein prinzipieller Unterschied* besteht. Ein solcher Unterschied entsteht erst durch die *Interpretation* der Nachricht durch den Empfänger, sowie ggf. durch seine *Reaktion* auf die interpretierte Nachricht.

Dies bedeutet für unsere Betriebssystem-Architektur, dass wir zur Realisierung von generischen Operationen keine neuen Konzepte einführen müssen, sondern die bereits ausführlich vorgestellten Abstraktionen Nest und Baustein verwenden können.

Konkret kann dies z.B. auf folgende Weise realisiert werden: Einer Nest-Instanz s wird ein so genanntes *Operations-Nest* als dynamisches Attribut zugeordnet, oder mit Hilfe einer einer eigenen Elementarfunktion `get_ops`(s). Das Operations-Nest dient zum Austausch der Nachrichten für die generischen Operationen, die auf dem *zugeordneten Nest* s ausgeführt werden sollen. Zu einem Operations-Nest *op* kann man umgekehrt das zugeordnete Nest durch eine Elementarfunktion `get_nest`(op) zugänglich machen. Es gilt dann `get_nest(get_ops`$(s)) = s$.

Zum Austausch der Nachrichten werden die in Abschnitt 3.3.8 behandelten Elementaroperationen `get_address`, `put_address`, `get`, `put` , `transfer` und `wait` verwendet. Das Operations-Nest darf zu einem beliebigen Zeitpunkt mehrere Operations-Aufrufe enthalten und ist daher prinzipiell geeignet, um die Funktionalität von Auftrags-Warteschlangen, Gerätetreiber-Warteschlangen u.ä. auszuführen.

Zum Übertragen der Nachrichten in einem Operations-Nest müssen auf jeden Fall Elementaroperationen verwendet werden[2], die an einen Schnittstellen-Mechanismus aus Abschnitt 2.3.2 gebunden sind, und mit deren Hilfe erst das Konzept der generischen Operationen auf einer höheren Abstraktionsstufe implementiert wird. Rein theoretisch wäre es möglich, auf die Implementierung der Elementaroperationen im zugeordneten Nest zu verzichten und diese ausschließlich durch generische Operationen im zugehörigen Operations-Nest auszuführen. Nicht nur aus Performanz-Gründen, sondern auch aus Symmetriegründen propagiere ich die umgekehrte Methodik, dass Elementaroperationen auf *beide* Arten aufrufbar sein sollen (und dabei dasselbe tun sollen). Es spielt dann keine Rolle, ob Elementaroperationen direkt auf dem zugeordneten Nest aufgerufen werden oder indirekt über das Operations-Nest in Auftrag gegeben werden[3]. Es wäre jedoch zu überlegen, die Operationen `lock` und `unlock` ausschließlich mittels generischer Operationen zu realiseren und damit nicht mehr als Elementaroperationen zu betrachten, da diese für die Herstellung von generischen Operationen nicht unbedingt benötigt werden; damit würden die Schnittstellen-Mechanismen aus Abschnitt 2.3.2 die Eigenschaft der Minimalität erfüllen.

Weiterhin bietet ein Operations-Nest die Möglichkeit zur *atomaren Bündelung* von Lock-Operationen. Dazu werden mehrere Lock-Operations-Beschreibungen in ein gemeinsames Datenpaket gepackt. Die Ausführung erfolgt entweder zusammen als atomare Einheit oder gar nicht. Damit lässt sich insbesondere das Handwerker-Problem [Jür73] ohne

[1] Nachrichten dienen zur Übertragung von Information (umgekehrt kann jedoch auch das Nicht-Senden oder das Nicht-Ankommen einer Nachricht eine Information darstellen, z.B. beim Ablaufen eines Timeouts; vgl. [Lam84]). Eine Übertragung von Information muss nicht unbedingt durch Herstellen einer *Kopie* erfolgen; beispielsweise stellt auch die Benutzung eines gemeinsamen Puffers durch mehrere Instanzen eine Realisierung einer Nachrichten-Übertragung dar.

[2] Dies ließe sich durch Benutzung von `get_ops(get_ops(`s`))`, also Einführen eines Operations-Nestes zum Operations-Nest vermeiden.

[3] In letzterem Fall ist ein *asynchroner Aufruf* möglich. Wie viel Sinn dies macht, sei dahin gestellt, da einige Elementaroperationen eigens zur Darstellung von Asynchronität entwickelt wurden.

Einführung weiterer Konzepte lösen. Bei der Bündelung mehrerer `wait`-Operationen ließe sich ggf. auch die Semantik disjunktiven Wartens ohne Einführung zusätzlicher Konzepte realisieren.

Nun zur Problematik der Polymorphie. Aufrufer und Bearbeiter müssen eine generische Operations-Nachricht auf zueinander passende Weise interpretieren, sonst entsteht eine Fehlfunktion des Systems.

Um Informationen darüber aufzubewahren, wie ein Nest-Inhalt zu interpretieren ist, ist das Konzept des Meta-Nestes vorgesehen (vgl. Operation `get_meta` in Abschnitt 3.4.5; vgl. Abschnitt 4.2)[4]. Dies kann nun analog auf Operations-Nester übertragen werden. Das Operations-Meta-Nest soll einen *Konsens* zwischen dem Aufrufer und dem Bearbeiter über die zu verwendenden *Datenformate* bzw *Datentypen* vermitteln.

Wir brauchen also ein *Typsystem*, wie es prinzipiell bei Programmiersprachen schon lange eingesetzt wird. In der Literatur dieses Gebiets werden unzählige Varianten von Typsystemen behandelt. Die Auswahl eines konkreten Typsystems fällt leichter, wenn man sich folgende Anforderungen vor Augen hält, die auch schon bisher in Betriebssystemen traditionell eine große Rolle gespielt haben:

1. Eine Typprüfung (d.h. ein Test, ob zwei Datenformate zueinander passen) *braucht nur* zur Laufzeit erfolgen. Es gibt genügend Anwendungsfälle, wo dies auf jeden Fall erst zur Laufzeit geschehen kann; dynamische Typprüfungen sind also unbedingt erforderlich und können statische Typprüfungen notfalls vollständig ersetzen[5]. Als Konsequenz hieraus brauchen statische Typprüfungen höchstens bei einfachen Grund-Datentypen und nicht notwendigerweise bei komplexen Datentypen vorgesehen zu werden. Dies vereinfacht ein Typsystem *enorm*.

2. Welche Typen (d.h. Datenformate) ein Bearbeiter akzeptieren und bearbeiten kann, kann nur dieser alleine festlegen. Typ-Informationen dienen zur Einweg-Kommunikation zwischen Bearbeiter und Aufrufer; damit kommen komplexere Mechanismen wie Typ-Inferenzsysteme nicht *auf Systemebene*[6] in Frage.

Es ist klar, dass trotz dieser Einschränkungen viele verschiedene Typsysteme und noch mehr konkrete Realisierungen derselben in Frage kommen. In nächsten Abschnitt werde ich zwei einfache Typsystem vorstellen, die wichtige Grundfunktionen abdecken sollten und lediglich als Beispiele zu verstehen sind.

[4] Das beim RPC oft als notwendig betrachtete Marshalling (Umwandlung zwischen inkompatiblen Datenformaten) wird hier nicht in die Grundmechanismen hineingesteckt, sondern kann bei Bedarf durch Anpassungsbausteine implementiert werden, die verschiedene Datenformate ineinander umwandeln.

[5] Bei genauer Betrachtung aktueller monolithischer Kern-Implementierungen im Unix-Umfeld fällt auf, dass diese exzessiven Gebrauch von dynamischen Typprüfungen machen. Dies wird z.B. durch die Existenz der Fehler-Rückmeldung EINVAL (Invalid Argument) in Unix belegt. Es gibt kaum Systemaufrufe, die Parameter verlangen, wo anstelle der Bearbeitung eines Aufruf-Auftrages nicht dieser Fehlercode als Rückmeldung kommen kann.

[6] Damit sind Typ-Inferenzsysteme jedoch nicht generell ausgeschlossen: die Abschnitt 4.2.2 vorgestellten `strategy_*`-Bausteine können nicht nur Laufzeit-Tests auf Typkompatibilität auszuführen, sondern im Prinzip beliebige Folge-Instantiierungen mit beliebigen Parametern auslösen. Damit sollte es theoretisch möglich sein, sogar ein Hindley-Milner-Typsystem nachzubilden.

6.1. Beispiel-Typsysteme

Einige Betriebssystem-Konstrukteure haben sich in der Vergangenheit geweigert, Typkonzepte einzuführen, was u.a. zu der historischen Spaltung zwischen Betriebssystem- und Datenbank-Konstrukteuren beigetragen hat. Argumente hierfür waren oder sind immer noch, dass im Betriebssystem fest verankerte Typen die Flexibilität mindern, die Schnittstellen verkomplizieren und obendrein Performanz kosten. Ich hoffe, diese Sichtweise relativieren zu können. Es geht mir nicht darum, fest verankerte Typen einzuführen, sondern *auf der Basis generischer Schnittstellen* die Deklaration *beliebiger* Typen zu ermöglichen, indem es *erweiterbar* ist. Typbeschreibungen stellen ein *optionales* Konzept dar, das auch weg gelassen werden kann; dann erhält man Generizität in Reinkultur. Richtig eingesetzt, kann es generische Schnittstellen ergänzen und vereinfachen, indem traditionell von Hand codierte Vorgänge (beispielsweise bei Verwendung von Assembler bis in die 1970er Jahre hinein) *automatisiert* werden. Ich versuche in den folgenden Beispielen aufzuzeigen, dass exzellente Performanz von Typprüfungen durch die Wahl geeigneter Sprachklassen und Kodierungen erreichbar ist.

6.1.1. Minimal-Typsystem

Hier geht es um die Frage, was ein Typsystem *unbedingt* enthalten muss, damit generische Operationen einigermaßen brauchbar und benutzbar sind. Alles, was dazu nicht erforderlich ist, versuchen wir wegzulassen und auf Laufzeit-Tests durch den Bearbeiter zu verlagern.

Wenn ein Operations-Nest mehrere verschiedene generische Operationen ermöglichen soll, dann gibt es eine Information, die in jedem Falle sowohl beim Aufrufer als auch beim Bearbeiter vorhanden sein muss: der *Name* der Operation.

Eine einfache Realisierung eines Minimal-Typsystems besteht darin, dass das Meta-Nest des Operations-Nestes eine Liste der implementierten zueinander disjunkten Operationsnamen enthält. Diese Liste lässt sich beispielsweise als lückenlos liegende Folge von Paketen realisieren, deren Grenzen im adjungierten Nest des Meta-Nestes angezeigt werden. Für die Kodierung der Operationsnamen kommen beispielsweise Text-Zeichenketten in Frage, die gegenüber Integer-Konstanten[7] zu bevorzugen sind.

Mehr braucht ein minimales Typsystem nicht zu enthalten[8]; ob die Anzahl, Größen und Werte der Operations-Parameter korrekt sind, kann der Bearbeiter selbst prüfen

[7] Diese werden z.B. in Unix zur Unterscheidung der durchzuführenden Operationen bei `ioctl` und `fcntl` eingesetzt. Über die damit verbundenen Probleme und Auswirkungen ist bereits genügend in der Folklore-Literatur diskutiert worden. Bei der Geschwindigkeit heutiger Prozessoren kann der minimal größere Aufwand beim Zugriff auf Zeichenketten gegenüber dem Zugriff auf ein Maschinenwort kein schlagendes Argument mehr sein, vor allem, wenn man den Zugewinn an Komfort dagegen stellt.

[8] Man könnte theoretisch sogar noch die Liste der Operationsnamen weglassen. Der Bearbeiter kann dann zwar das Datenformat aller eingehenden Aufträge selbst prüfen, der Aufrufer ist in diesem Falle aber vollkommen „blind“ und kann die Operations-Namen nur erraten, wenn er sie nicht aus einer anderen externen Quelle (wie z.B. eine Schnittstellen-Definition) weiß. Wenn überhaupt keine Typinformation kommuniziert wird, dann handelt es sich nicht um ein Typsystem im eigentlichen Sinne.

und ggf. mit der Rücklieferung einer Fehlermeldung reagieren.

6.1.2. Einfaches erweiterbares Typsystem

Das soeben vorgestellte minimale Typsystem ist bereits insofern *erweiterbar*, als dass der Wertevorrat für die Kodierung der Operationsnamen so groß gewählt werden kann, dass zukünftige Erweiterungen um neue Operationen praktisch unbeschränkt möglich sind. Es geht jetzt um die Frage von *Schnittstellen-Erweiterungen*, die man unter die Rubrik „Erweiterungs-Generizität“ (vgl. Abschnitt 2.1.2) einordnen kann[9]. Vor dem Einsatz von Erweiterungs-Generizität sollte man gewissenhaft prüfen, ob der gleiche Effekt nicht auch durch kompositorische Generizität (vgl. Abschnitt 2.1.3) erreichbar ist. Kompositorische Generizität hat grundsätzlich Vorrang vor Erweiterungs-Generizität; diese Philosophie ist ein wichtiges Unterscheidungsmerkmal des hier vorgestellten Ansatzes gegenüber den meisten konkurrierenden Ansätzen.

In der Literatur aus dem Gebiet der Programmiersprachen werden Typsysteme als *algebraische Struktur* aufgefasst, für die eine *abstrakte Syntax* angegeben werden kann und meist auch nur angegeben wird. Dies genügt i.A. für unsere Zwecke nicht: ein Betriebssystem muss auch die *konkreten Datenformate* (sog. Aufruf-Konventionen) festlegen, da die System-Schnittstellen übergreifend für unterschiedliche Aufrufer-Typen gelten sollen, insbesondere den Anschluss an unterschiedliche Programmiersprachen und Laufzeitsystem-Modelle ermöglichen sollen. Wir benötigen daher eine Spezifikation der algebraischen Struktur in *konkreter Syntax*[10].

Zur Spezifikation einer konkreten Syntax eignen sich bekanntermaßen *kontextfreie Grammatiken*. Diese bringen jedoch in ihrer Allgemeinform das Problem der *Mehrdeutigkeit* mit sich: bei unbedachter Verwendung zur Beschreibung von Datenformaten kann es vorkommen, dass ein konkret vorliegendes Datenformats-Muster verschiedene Interpretationen zulässt. Aus der Theorie der formalen Sprachen ist bekannt, dass die Mitgliedschaft einer beliebigen Grammatik in der Klasse der *eindeutig* kontextfreien Sprachen nicht entscheidbar ist; zum Glück gibt es jedoch entscheidbare Unterklassen wie die bekannten $LR(k)$ oder $LL(k)$-Grammatikklassen. Diese sind nicht nur entscheidbar, sondern reichen für praktische Zwecke vollkommen aus[11].

Ich möchte noch einen Schritt weiter gehen und zeigen, dass für die Zwecke von Betriebssystemen die Klasse derjenigen $LL(1)$-Grammatiken ausreichend ist, die sich zusätzlich in Greibach-Normalform[12] befinden. Das Wortproblem bzw. das Parsing-Problem ist bei diesen Grammatiken trivial lösbar, da man beispielsweise das bekannte Verfahren des *rekursiven Abstiegs* in *interpretativer Weise* direkt auf einer Kodierung der Grammatik ausführen kann, ohne die bei $LL(1)$ notwendigen FIRST-Mengen berechnen zu müssen, da diese trivialerweise mit dem ersten Terminalsymbol übereinstimmen, mit dem jede Grammatikregel wegen der Greibach-Normalform beginnen muss. Die beim rekursiven Abstieg notwendigen Parsing-Entscheidungen werden dadurch trivial.

Um sicherzustellen, dass eine Greibach-Grammatik auf jeden Fall die $LL(1)$-Eigenschaft erfüllt, gibt es eine einfache Methode, die leicht zu erlernen und zu beherrschen sein dürfte: Man verlangt als *Ersatz-Forderung* für die $LL(1)$-Eigenschaft, dass keine verschiedenen Produktionsregeln der Form $X \rightarrow a\alpha$ und $X \rightarrow a\beta$ mit gleichem Nichtterminalsymbol X und gleichem Terminalsymbol a, aber verschiedenen Fortsetzungen α und β vorkommen dürfen. Diese Forderung lässt sich dadurch einhalten, dass man für α bzw. β ein neues Nichtterminalsymbol Y einführt, das die Rolle der bisherigen Reste α und β gemeinsam ausführen und übernehmen soll. Da alle Produktionen für Y wiederum den gleichen Erfordernissen unterliegen, ergibt sich die $LL(1)$-Eigenschaft beim *Entwurf* der kontextfreien Grammatik durch einen Menschen ganz von selbst. Ich nenne eine Greibach-Grammatik mit dieser Ersatz-Forderung eine *Trivial-Grammatik*.

Eine Trivial-Grammatik setzt die folgenden bekannten Grundprinzipien des Entwurfs von Datenstrukturen direkt in leicht zu handhabende Grammatik-Regeln um:

- Die Schachtelung
- Die Sequenz
- Die Alternative

Eine Schachtelung von Datenformaten wird durch die Unterscheidung zwischen Terminal- und Nichtterminalsymbolen ausgedrückt; ein Nichtterminalsymbol lässt sich an verschiedenen Stellen *wiederverwenden*. Eine Sequenz von Datenformaten[13] wird durch Hintereinanderschreiben von Terminal- oder Nichtterminalsymbolen in einer Grammatikregel ausgedrückt; mathematisch gesehen repräsentiert es ein kartesisches Produkt. Alternativen zum gleichen Nichtterminal müssen im Gegensatz zu beliebigen kontextfreien Grammatikregeln *stets* durch voneinander verschiedene Terminalsymbole eingeleitet werden, so dass die jeweils richtige Alternative stets am Anfang eines rekursiven Abstiegs durch einen *trivialen Diskriminator*[14] erkannt werden kann.

Im Gegensatz zu Programmiersprachen wird diese Methodik in Betriebssystemen vollkommen dynamisch eingesetzt; die Erkennung eines Datenformats kann i.A. nicht vom Typsystem eines Compilers vorweg genommen werden, sondern muss mindestens bei der Verdrahtungs-Operation zur Laufzeit durchführbar sein. Hierfür eig-

[9] Diese Problematik taucht in Betriebssystemen immer wieder auf, und zwar nicht nur in solchen, die Objektorientierung als Leitmotiv angeben. Microsoft hat beispielsweise den Begriff der „DLL-Hölle“ geprägt. Damit sind zueinander inkompatible Versionen von Systembibliotheken gemeint, die aufgrund intern geänderter Schnittstellen nicht zusammen passen.

[10] Beispiele für die konkrete Syntax von system-übergreifenden Schnittstellen finden sich u.a. in den Internet-RFCs (Request For Comments). Datenformate werden dort fast durchwegs durch kontextfreie Grammatiken in einer zur Backus-Naur-Form ähnlichen Notation angegeben.

[11] Ein Beleg hierfür sind die formalen Spezifikationen von Netzwerk-Protokollen und -Datenformaten in den Internet-RFCs. Diese scheinen mir meist in der Klasse $LL(k)$ zu liegen.

[12] Bei der Greibach-Normalform müssen alle Produktionsregeln die Form $X \rightarrow a\alpha$ haben, wobei X ein Nichtterminalsymbol, a ein Terminalsymbol, und α eine beliebige Zeichenfolge aus Terminal- oder Nichtterminalsymbolen ist.

[13] Typische Vertreter von Sequenzen sind Arrays und Records.

[14] Dieses Konzept taucht z.B. in den *Tags* von varianten Records von Pascal-ähnlichen Programmiersprachen auf; in Ada ist die Verwendung eines Tag-Feldes sogar Pflicht.

net sich die *Laufzeit-Interpretation*[15] von Datenformaten mit Hilfe kontextfreier Grammatikregeln ganz besonders. Performanz-Steigerungen durch *Vorübersetzung* von Trivial-Grammatiken in deterministische Kellerautomaten sind dadurch nicht ausgeschlossen[16].

Eine konkrete Realisierung mittels interpretierbaren Trivial-Grammatiken könnte beispielsweise folgendermaßen aussehen:

Nichtterminalsymbole werden durch ASCII-Zeichenketten aus Großbuchstaben und Unterstrich dargestellt. Ein Beispiel wäre „`IP_ADDRESS`“.

Terminalsymbolklassen verhalten sich logisch wie Nichtterminale, zu denen keine Regeln definiert werden dürfen, da es bereits fest eingebaute Regeln für sie gibt[17]. Sie werden durch ASCII-Zeichenketten in Kleinschreibung dargestellt, denen eine ASCII-kodierte Zahl angefügt ist, die den Platzbedarf in Bytes beschreibt. Beispiele sind „`int1`“, „`int2`“, „`int4`“ oder „`int8`“, die vorzeichen-behaftet zu interpretierende Maschinenwörter der jeweiligen Länge darstellen sollen, oder „`space512`“ für einen uninterpretierten Datenblock. Weitere Terminalsymbolklassen wie „`bigendian_unsigned_int4`“ lassen sich später jederzeit hinzu erfinden; welche Bedeutung sich hinter dem Namen verbirgt, ist für das Parsen des Datenformats egal, nur der Platzbedarf ist entscheidend.

Terminalsymbole bezeichnen ein konkretes Bitmuster und werden durch Hintereinanderschreiben einer Terminalsymbolklasse und einer konkreten Ausprägung in Klammern notiert. Beispiele sind „`int2(7)`“ oder „`int4(-1)`“. Für Zeichenketten variabler Länge kann man eine Konvention der Art „`string('Name')`“ einführen, bei der sich der aktuelle Platzbedarf aus dem Text zwischen den Hochkommata ergibt.

Grammatikregeln lassen sich beispielsweise auf folgende Weise kodieren:
„`STRUCTURE=int1(0);SUBSTRUCTURE;int4;`“
Im Beispiel werden die Strichpunkte nicht nur als Trennzeichen, sondern auch als Abschlusszeichen einer Regel eingesetzt. Die Trivial-Grammatik-Eigenschaft ist erfüllt, wenn keine zwei Regeln vorhanden sind, die bis zum Auftreten des ersten Strichpunktes miteinander übereinstimmen; weiterhin müssen alle Regeln zum gleichen Nichtterminal „`STRUCTURE`“ hinter dem Gleichheitszeichen mit der gleichen Terminalsymbolklasse beginnen. Falls ein Nichtterminal keine Alternativen enthalten soll oder darf, dann ist als Sonderfall eines Terminalsymbols auch „`0(0)`“ zulässig, das einen Diskriminator repräsentiert, der keinen Platz beansprucht und von dem es deshalb nur eine einzige Ausprägung gibt (so dass als Folge davon keine weiteren Alternativen mehr zulässig sind).

Für das *Startsymbol* der Grammatik kann man eine Konvention einführen, beispielsweise ein Nichtterminalsymbol „`OP`“ für die Beschreibung von Operations-Nestern, oder „`DATA`“ für die Beschreibung von Datenformaten in persistenten Nestern. Die Beschreibung der vier wichtigsten Grundoperationen sieht dann beispielsweise folgendermaßen aus:

```
OP=string('get');int8;int8;addr8;int8;
int1;res_addr8;res_int8;res_int1;res_
int1;
OP=string('transfer');int8;int8;addr8;
int8;addr8;int1;int1;addr8;res_int1;
OP=string('wait');int8;int8;addr8;int8;
int1;int1;res_int1;res_int1;
OP=string('put');int8;int8;addr8;int1;
```

Dieses Beispiel stellt nur die wichtigsten Grundelemente eines Typsystems vor und sollte für den praktischen Einsatz um weitere Ausdrucksmöglichkeiten erweitert werden, etwa Feldnamen zur Beschreibung von Records oder weitere Terminalklassen zur Beschreibung von Bitfeldern oder von Bereichstypen, sowie von Ausrichtungs-Konventionen („alignment“). Mit Hilfe weiterer Konventionen ist beispielsweise die automatisierte Überprüfung der Gültigkeit von Adressbereichen oder von Deskriptoren möglich, so dass diese nicht mehr wie in bisherigen Betriebssystem-Implementierungen „von Hand“ ad hoc getestet werden muss.

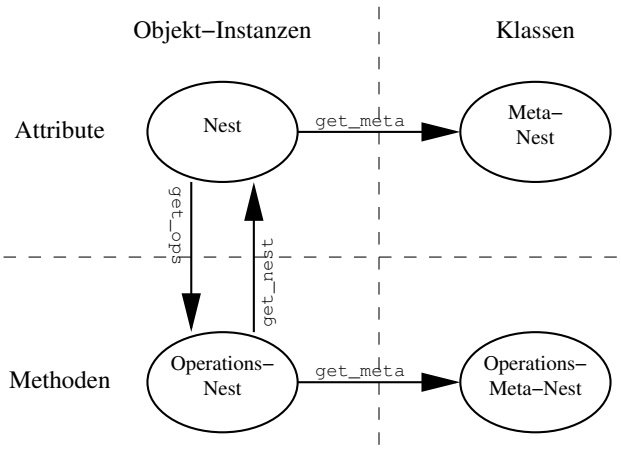
[15] Dies schließt nicht die Verwendung eines Prä-Compilers aus, der immer wiederkehrende Teile der Interpretation einmal vorweg berechnet (sog. *Precomputing*).

[16] Ob die Vorübersetzung in ausführbare Maschinenbefehle („just-in-time compiler“) weitere Vorteile bringen kann, wäre zu untersuchen.

[17] Dieses Konzept taucht in Compilern als Token-Klasse auf, z.B. die Token-Klasse aller Identifier oder aller Zeichenketten.

6.2. Zusammenhang mit der Objektorientierung

6.2.1. Simulation der Vererbung durch Bausteine



Das Bild zeigt den Zusammenhang der vier vorgestellten Nest-Arten mit bekannten Konzepten der Objektorientierung. Die gestrichelten Trennlinien sollen eine Unterteilung des Feldes nach zwei verschiedenen Kriterien andeuten:

- Unterteilung in Objekt-Instanzen und Klassen: Nest-Instanz versus zugehöriges Meta-Nest
- Unterteilung in Attribute und Methoden: Daten-Nest versus Operations-Nest

Die Beziehungen von Instanzen der jeweiligen Nest-Art sind durch Pfeile mit Beschriftung der zugehörigen Zugriffsfunktionen gekennzeichnet. Eine Nest-Instanz steht mit ihrer zugehörigen Operations-Nest-Instanz in einer 1:1-Beziehung, ähnlich wie beim objektorientierten Zusammenpacken von Attributen und Methoden zu einer Objekt-Instanz. Der *Status* einer Objekt-Instanz wird hierbei in einer Nest-Instanz abgespeichert.

Der Zusammenhang mit der *Vererbung* lässt sich durch zwei Anpassungs-Bausteine darstellen, die entsprechende Transformationen auf den Nestern bzw. Meta-Nestern durchführen.

Beim `downcast` wird die Schnittstelle erweitert; zu den am Eingang `base_class` verfügbaren Attributen und Methoden können neue hinzukommen, die am Ausgang `son_class` zur Verfügung gestellt werden; ggf. kann dabei auch die *Implementierung* (bzw. das *Verhalten*) einiger Methoden abgeändert werden (*überschriebene Methoden*). Da sich dabei der Umfang des abgespeicherten Status vergrößern kann, ist ein Eingang `tmp` vorgesehen, um diesen aufzunehmen. Bei geeigneten Konventionen über die Platz-Allokation im `base_class`-Nest lässt sich der Status auch dort abspeichern, so dass `tmp` überflüssig wird.

Beim `upcast` wird lediglich die Schnittstelle verkleinert, so dass nicht mehr alle Attribute und Methoden von außen[18] zugreifbar sind (Prinzip der *Verbergung*).

Aus dieser Beschreibung wird ersichtlich, dass die Konzepte der Objektorientierung einen *Spezialfall* der hier vorgestellten Methodik darstellen. Durch die Abstraktionen Nest und Baustein sind folgende Konzepte realisierbar, die bei *üblichen*[19] Ausprägungen von Objektorientierung im Regelfall nicht vorgesehen sind:

- Austausch einzelner Methoden-Implementierungen zur *Laufzeit*, und zwar unabhängig von einer Klassenhierarchie[20]
- Änderung der *Repräsentation* von Attributen (beim Erweitern der Schnittstelle und/oder zur Laufzeit); vgl. Anhang B.
- Aufspaltung eines Status-Raums in beliebige Kombinationen von Teilräumen[21]
- Einfache Verwendung *beliebig unterschiedlicher* Methoden-Sätze[22]

6.2.2. Grundlegende Unterschiede zur Objektorientierung

In der Objektorientierung kann jede Objekt-Instanz prinzipiell die Methoden jeder anderen Objekt-Instanz aufrufen; Schutzmechanismen zur Einschränkung existieren meist nur auf Klassen-Ebene (z.B. `private` versus `public`). Die Objektorientierung basiert daher auf einem Kommunikations-Paradigma, das als *verbindungslose Kommunikation* charakterisiert werden kann.

Im Unterschied dazu sorgen Baustein-Verdrahtungen und ihre Verdrahtungsregeln dafür, dass eine *direktionale verbindungsorientierte Kommunikationsstruktur* eingehalten wird. Der Begriff „direktional“ bedeutet, dass es

[18] Eine Verkleinerung läßt sich dadurch erzielen, dass Attribute und Methoden in den jeweiligen Meta-Nestern nicht mehr erscheinen. Ein besserer *Zugriffsschutz* läßt sich dadurch realisieren, dass bei der Transformation auch Ausblendungen in den jeweiligen adjungierten Nestern vorgenommen werden.

[19] Eine Ausnahme ist Smalltalk, wo Methoden-Änderungen oder sogar Klassen-Änderungen zur Laufzeit prinzipiell möglich sind, da alle Strukturen dynamisch angelegt sind.

[20] Beispiele sind Pipe-Operationen in Linux, deren Semantik vom Modus abhängt, in dem die Pipe geöffnet wurde. Es gibt Implementierungen, die jeden Modus als eigene Prozedur realisieren und über eine einheitliche generische Schnittstelle verfügbar machen.

[21] Übliche Schutzkonzepte von objektorientierten Programmiersprachen wie C++ oder Java sehen eine global gültige Unterteilung in private und öffentliche Regionen vor. Demgegenüber erlaubt das Konzept der adjungierten Nester auch Sichten, die für jede Teilnehmer-Instanz anders ausfallen können.

[22] Bei der objektorientierten Vererbung werden Methoden-Schnittstellen meist nur *erweitert*. Bausteine lassen hingegen auch das *Verdecken* bisheriger Attribute oder Methoden zu. Ich halte dies für sehr nützlich, weil es das *Steigern der Abstraktionsstufe* in einer Baustein-Hierarchie ermöglicht: logisch betrachtet handelt es sich nach einer Steigerung der Abstraktionsstufe zwar immer noch um dieselbe Objekt-Instanz, die Zugriffsmethoden werden jedoch „abstrakter“. Man kann Benutzern der höheren Abstraktionsstufen untersagen, diese mit Methoden niedrigerer Stufe gemischt zu verwenden, um beispielsweise die Umgehung von erst weiter oben gegebenen *Zusicherungen* zu unterbinden („schwarze Schnittstellen“).

eine *Vorzugsrichtung* der Operationsaufrufe gibt (nämlich vom Konsumenten zum Produzenten). Falls Kommunikation in Gegenrichtung stattfindet (z.B. bei `notify_*`-Operationen; vgl. Kapitel 5), wird diese ausdrücklich gekennzeichnet. Der Vorteil dieser Kommunikations-Einschränkung liegt darin, dass zyklische Aufrufe (vgl. [Ass96]) entweder gleich von vornherein unterbunden, oder wenigstens leichter lokalisierbar werden.

Verbindungsorientierte Kommunikation ist ein Grundmechanismus, der verbindungslose Kommunikation anscheinend nur mühsam simulieren kann (vgl. Reflektoren in Abschnitt 4.3.2). Durch Baustein-Arten wie `socket_*` oder `rpc_*` ist jedoch eine Implementierung verbindungsloser Kommunikation auf höherer Ebene möglich. Daher stellt die Verwendung von direktionaler verbindungsorientierter Kommunikation als Grundkonzept keine Einschränkung der erzielbaren Mächtigkeit eines Betriebssystems dar.

Ein weiterer grundlegender Unterschied der hier vorgestellten Architektur zur Objektorientierung besteht darin, dass Hierarchie-Beziehungen auf *Instanz*-Ebene und nicht auf Klassen-Ebene eingehalten werden (vgl. Einleitung von Kapitel 4). In OO-Entwürfen wird sehr häufig nur auf Klassen-Ebene argumentiert und nachgedacht; disziplinierende Einschränkungen auf dieser Ebene werden mit der Hoffnung verbunden, dass sich Objekt-Instanzen dann zur Laufzeit ebenfalls diszipliniert verhalten mögen. Ich möchte aufzeigen, dass diese Hoffnung in manchen Fällen trügerisch sein kann.

In OO-Systemen dürfen Objekt-Instanzen *Referenzen* (Pointer[23]) auf andere Objekt-Instanzen enthalten. Die Verantwortung für den Inhalt dieser Referenzen liegt beim Programmcode der jeweiligen Klasse (oder einer Oberklasse), zu der das Objekt gehört. Sowohl die Initialisierung als auch das Überschreiben von Referenzen ist damit durch Programmcode erlaubt, der prinzipiell Turing-vollständig ist (Äquivalenz zum Halteproblem), so dass sich *im allgemeinen* nicht voraussagen lässt, welche Objekt-Instanzen auf welche anderen zur Laufzeit verweisen werden. Gleiches gilt auch für die dynamischen Aufruf-Beziehungen (dynamische Kette) zur Laufzeit. Damit ist i.A. unentscheidbar, welche Produzenten-Konsumenten-Beziehungen oder andere Hierarchie-Beziehungen sich zur Laufzeit zwischen den Instanzen herausbilden werden (vgl. [Ass96]). Diese theoretische Schwäche wird durch die praktische Erfahrung ergänzt, dass sich in großen gewachsenen OO-Systemen oftmals kaum durchschaubare graphenartige Geflechte zwischen den Objekten bilden, deren Entstehung und deren Verhalten schwer nachvollziehbar ist; die Fehlersuche mit Hilfe von Debuggern scheint in solchen Fällen unverzichtbar zu sein und kann teuer werden.

Im Gegensatz zur Objektorientierung strebt der hier vorgestellte Entwurf keine Hierarchiebeziehungen auf Klassen- bzw. Baustein-Arten-Ebene an, sondern ganz im Gegenteil die beliebige Kombinierbarkeit durch kompositorische Generizität. Dafür wird jedoch auf Instanz-Ebene eine rigorose Disziplin durch Verdrahtungsregeln gefordert, deren Durchsetzung nicht auf viele einzelne Baustein-Arten oder -Instanzen verteilt wird, sondern einer (teilweise) zentralen Instanz wie `control` verantwortlich übergeben wird. Baustein-Instanzen haben selbst keinen direkten Einfluss darauf, mit welchen anderen Instanzen sie kommunizieren. Referenzen werden ausschließlich als Verdrahtungs-Leitungen modelliert, deren Kontrolle zentralistisch durch `control` und die daran angeschlossene Strategie-Ebene erfolgt (vgl. Abschnitt 4.2.2). Die Prüfung der Korrektheit der Implementierung und der resultierenden hierarchischen Laufzeit-Beziehungen ist damit im Gegensatz zur Objektorientierung stets auf zentrale Stellen lokalisiert.

Die Beachtung von Instanz-Beziehungen und die Verwaltung durch eine getrennte Strategie-Instanzen wie `control` lässt sich als Umsetzung eines *Programmierparadigmas* verstehen, das *Instanz-Orientierung* [ST04b] genannt wird und in Kapitel 7 vorgestellt wird.

Ein weiterer grundlegender Unterschied zum Objekt-Paradigma besteht darin, dass Bausteine *statuslos* (vgl. 4.1.2) implementiert werden können (aber nicht unbedingt müssen). Statuslosigkeit bedeutet, dass die *Verantwortung* zum Aufbewahren von Zustands-Informationen an vorgeschaltete Baustein- bzw. Nest-Instanzen *delegiert* wird. Da Nest-Inhalte virtuell berechnet werden können, kann die Status-Information an dieser Virtualisierung teilnehmen.

In der Objektorientierung werden dagegen Objekt-Repräsentationen als weit gehend *festes* Grundkonzept betrachtet, auf dem dann weitere Konstrukte wie z.B. Meta-Hierarchien [Mae87, YTY+89, Yok92] aufgebaut werden können; charakteristisch hierfür ist die Denkweise „alles ist ein Objekt“. Objekte sind durch ihren *Status* und durch ihr *Verhalten* charakterisiert. Der Status ist damit ein *grundlegender Bestandteil* objektorientierter Konstruktionen.

Die hier propagierte Statuslosigkeit von Bausteinen weicht von diesem Grundansatz erheblich ab. Statuslose Baustein-Instanzen lassen sich zur Laufzeit ersetzen, sie brauchen keine feste Identität zu besitzen. Eine Nest-Instanz besitzt zwar ebenfalls einen Status; wo dieser jedoch effektiv *aufbewahrt* oder *hergestellt* wird, ist eine *unabhängige* Frage. Statt eine feste Repräsentation im Speicher vorauszusetzen, muss jeder Zugriff über die Elementaroperationen abgewickelt werden. Die beauftragte Instanz kann daher die Repräsentation in beinahe beliebiger Weise beeinflussen. Damit werden die Repräsentationen „open ended“.

Reflektive Architekturen wie [Mae87] können zwar die Repräsentationen im Endeffekt auch auf beinahe beliebige Weise beeinflussen, indem sie auf semantischer Ebene den *Interpreter* für die Repräsentation abänderbar gestalten. Dies geschieht jedoch auf Basis einer *festen* Objekt-Repräsentation, deren abänderbare Semantik sich durch eine beeinflussbare Interpretation *oberhalb* davon auf einer *Meta-Ebene* ergibt. Die hier vorgestellte Denkweise erlaubt hingegen auch den Austausch der Repräsentation *unterhalb* der Anwendungs-Logik, sowie das Einfügen von zusätzlichen Interpretationen auf unterer Ebene, ohne deshalb den Interpreter für die Anwendungs-Logik abändern zu müssen[24]. Weiterhin erlaubt die Instanz-Orientierung *mehrere*

[23] Eine bekannte folkloristische Kritik an Pointern lautet, dass sie ein Pendant zum GOTO auf Datenstruktur-Ebene darstellen. Das Folge-Problem der potentiellen Unstrukturiertheit von Instanz-Beziehungen verschwindet nicht durch den Ersatz von Low-Level-Pointern durch Referenzen (z.B. in Java), die lediglich eine uniforme Verwendung von Referenz-Semantik erzwingen.

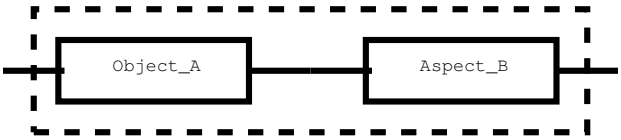
[24] Die in reflektiven Architekturen verwendeten Techniken dürften sich im hier vorgestellen Ansatz nachbilden lassen; die Fragestellungen der Reichweite und der Vor- und Nachteile verschiedener Lösungsalternativen für „open endedness“ sollten in zukünftigen Forschungen geklärt

Sichten auf ein System; reflektive Architekturen sind hingegen auf die Abänderung einer einzigen Sicht beschränkt.

6.3. Zusammenhang mit der Aspektorientierung (AOP)

Dieser Abschnitt setzt die Kenntnis der Aspektorientierung aus der Literatur voraus und kann ohne Schaden übersprungen werden.

Die Aspektorientierte Programmierung (AOP) [K+97b] lässt sich unter das Dach der Generizität einordnen. In der allgemeinsten Form geht es dabei um mehrere verschiedenartige Formen der Komposition von Funktionen, die nicht kompatibel zu einander sind („cross-cutting concern"). Häufig werden Aspekte in der Praxis auch dann zur Komposition von Funktionen benutzt, wenn nur eine einzige Art der Komposition notwendig ist (siehe z.B. [K+97b][25]). Dieser Spezialfall lässt sich durch Bausteine auf relativ einfache Weise simulieren, sofern man als *join-point-Modell* (siehe [K+97b]) jeweils die Operations-Aufrufe wählt:

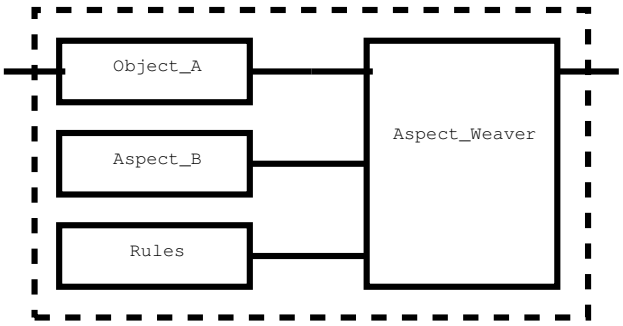


Zur Vereinfachung betrachten wir hier nur Objekt-Bausteine mit jeweils einem Eingang und einem Ausgang; die Erweiterung auf den allgemeinen Fall ist relativ einfach und wird hier nicht detailliert. Ein Aspekt, der nur `receptions`-*pointcuts* (siehe [K+01]) enthält, wird durch Nachschalten eines Bausteins realisiert, der den *advice*-Code (siehe ebenfalls [K+01]) zur Realisierung des Aspekts enthält. Ein am Ausgang des Aspekt-Bausteins ankommender Operations-Aufruf wird zunächst von ihm bearbeitet, indem er erst den `before`-advice ausführt, anschließend die eigentliche Operation des Objektes über seinen Eingang aufruft (entsprechend dem `proceed()` bei `around`-advice [K+01]), und schließlich den `after`-advice des Aspektes ausführt.

Zur Realisierung von `calls`-pointcuts müssen die Objekt- und Aspekt-Instanzen in ihrer Reihenfolge vertauscht werden. Falls beide pointcut-Arten gemischt verwendet werden sollen, sind im bisher propagierten zyklenfreien Verdrahtungs-Modell jeweils eine vorgeschaltete und eine nachgeschaltete Teil-Aspekt-Instanz notwendig; durch Zulassen von Schleifen-Verdrahtungen kann diese Trennung auch überwunden werden. Bei der Zuordnung von mehreren Aspekten zu einer Objekt-Instanz entstehen längere Ketten mit mehreren Vorschalt- oder Nachschalt-Bausteinen. Die Zuordnung eines Aspektes an mehrere Objekt-Arten lässt sich relativ bequem durch `strategy_*`-Bausteine automatisieren, die für das automatische Zwischenschalten der Aspekt-Bausteine bei der Instantiierung durch `control` sorgen (vgl. Abschnitt 4.2.2). Diese Realisierung erlaubt prinzipiell auch vollkommen dynamische Laufzeit-Entscheidungen, welche Objekt-Instanzen mit welchen Aspekt-Instanzen zusammen gefasst werden sollen.

Bei vollkommen statuslos realisierten Objekt-Bausteinen werden interne Daten-Zugriffe auf Attribute über eine Nest-Schnittstelle (meist mit `tmp` bezeichnet) explizit abgewickelt. Da diese Operations-Aufrufe ebenfalls join points darstellen, lassen sich damit auch `gets`- und `puts`-pointcuts (siehe [K+01]) nachbilden. Falls Aspekte zusätzliche Status-Informationen wie eigene Attribute einführen, kann diese beispielsweise in zusätzlichen `tmp`-Eingängen aufbewahrt werden.

Nachteilig an der Realisierung von Aspekten durch Vorschalt- oder Nachschalt-Bausteine ist, dass spezifische crosscutting-concerns, die vom Typ des Objekt-Bausteins abhängen, entweder jeweils separat implementiert werden müssen, oder zu komplexen Fallunterscheidungen innerhalb einer universellen Aspekt-Implementierung führen können. Dieser Nachteil lässt sich durch einen (universellen) Interpreter umgehen, der ähnlich zu einem Aspekt-Weber die Verknüpfungen an den join points durchführt (lediglich interpretativ zur Laufzeit; das Interpreter-Verhalten kann notfalls durch Techniken der *partiellen Evaluation* wie z.B. dynamische Code-Generierung vermieden werden). In diesem Falle ist es sinnvoll, im Aspekt-Baustein nur advices zu implementieren, und die Spezifikation von pointcuts in ein separates Nest `Rules` zu legen, das sich unabhängig vom advice-Code austauschen lässt:



In Erweiterung zu „flachen" aspektorientierten Modellen [K+01] lässt sich das Prinzip der Zuordnung von Aspekten auch rekursiv (z.B. durch Baustein-Schachtelung) fortsetzen[26], beispielsweise um Aspekte mit anderen Aspekten

werden.

[25] Nach Definition in [K+97b, Abschnitt 3.3] handelt es sich bei Funktionen, die nur auf eine einzige Weise komponiert werden müssen, um Komponenten und nicht um Aspekte, da sie beispielsweise mit Hilfe von Makro-Techniken auf klare Weise separat gekapselt werden können. In [K+97b, Tabelle in Abschnitt 4] wird als Beispiel-Aspekt für Matrix-Algorithmen die Matrix-Repräsentation angeführt, obwohl sich diese meiner Ansicht nach problemlos als unabhängige Komponente realisieren lässt und ein „cross-cutting concern" nicht notwendigerweise entstehen muss. Diese schon in der Original-Arbeit über Aspektorientierung vorhandene Auffassung von Aspekten als (verallgemeinerte) Komponenten scheint ein in der Praxis häufig vorkommender Fall zu sein. Eine eindeutige Abgrenzung zwischen Komponenten und Aspekten dürfte nicht in allen Fällen möglich sein, da diese Abgrenzung von der gewählten Abstraktionsebene, vom Standpunkt des Betrachters, seiner schulischen Herkunft, und von anderen Faktoren abhängen kann. Daher wäre zu überlegen, ob man die Aspektorientierung nicht als echten Unterbegriff des neu einzuführenden Oberbegriffes „Komponenten-Orientierung" auffassen sollte. Möglicherweise bekommt man bei geeigneter Definition der Begriffe auch die Objektorientierung als Spezialfall unter das Dach einer derartigen Verallgemeinerung.

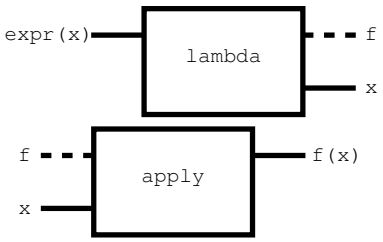
[26] In zukünftigen Forschungen sollte untersucht werden, wie sich die algebraischen Eigenschaften der Komposition von Bausteinen (z.B. das Assoziativgesetz) auf interpretative Aspekt-Weber-Bausteine übertragen lassen. Eventuell ergäbe sich hieraus eine interessante Rückmeldung an die Gemeinschaft der Experimentatoren, die sich zur Zeit sehr

zu verknüpfen, oder um „Vererbung“ zwischen Aspekten nachzubilden.

Weitere pointcut-Arten und join-point-Modelle (vgl. [K+01]) werden in dieser Arbeit nicht behandelt; ob und wie weit diese ebenfalls durch Nester und Bausteine simulierbar sind, sollen zukünftige Forschungen zeigen. Dieser Abschnitt soll lediglich plausibel machen, dass einige wesentliche Grundkonzepte der Aspektorientierung durch das hier vorgestellte Modell nachgebildet werden können; dies betrifft insbesondere häufige Anwendungsfälle der Aspektorientierung in Betriebssystemen wie Prüf- und Sicherheits-Bausteine, Sperrbehandlung, Debugging / Tracing u.v.m. Ob die Mächtigkeit der hier verwendeten kompositorischen Generizität in der Praxis vollkommen ausreichend für sämtliche Betriebssystem-Zwecke ist, oder ob die Hinzunahme spezifisch aspektorientierter Formen von kompositorischer Generizität (beispielsweise durch Erweitern des in Abschnitt 2.6 erwähnten Quelltext-Präprozessors zu einem vollwertigen Aspekt-Weber) notwendig oder sinnvoll ist, sollte in zukünftigen Forschungen geklärt werden.

6.4. Zusammenhang mit dem Funktionalen Paradigma

Die syntaktische Struktur von Ausdrücken des *puren Lambda-Kalküls* [Fie88] lässt sich mittels der folgenden Bausteine nachbilden:



Für jeden Lambda-Ausdruck der Form $f \equiv \lambda x.expr(x)$ wird ein Unternetzwerk `expr(x)` mit einem oder mehreren freien (intern unverdrahteten) Eingängen `x` erstellt, der/die auf den `x`-Ausgang einer `lambda`-Baustein-Instanz verdrahtet wird/werden. Der Ausgang des `expr(x)`-Netzwerks wird mit dem gleichnamigen Eingang des `lambda`-Bausteins verdrahtet, so dass im Endeffekt ein Zyklus entsteht. Für jede Funktionsapplikation $f\,x$ wird eine `apply`-Instanz generiert, die `f` und `x` zu einem Netzwerk `f(x)` gemäß der gleichnamigen Ein- und Ausgänge verknüpft. Auf diese Weise lässt sich jeder Ausdruck des *puren* Lambda-Kalküls eineindeutig in ein äquivalentes Baustein-Netzwerk übersetzen.

Die Auswertung eines solchen Ausdrucks kann am einfachsten auf `control`-Ebene durch einen Baustein `strategy_lambdarewriting` erfolgen, der Substitution von Termen, ggf. auch Graph-Reduktion [Fie88, SPvE93] durchführt.

Weiterhin lassen sich auch nicht-pure Varianten des Lambda-Kalküls durch Einführung geeigneter Bausteine für die Repräsentation von Grundtermen simulieren.

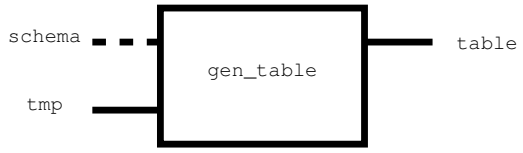
Zu beachten ist bei dieser Art von Simulation, dass es sich um eine rein syntaktische Ersetzung handelt, bei der zwischen den Netzwerken keine echten Informationen ausgetauscht werden müssen, sondern die Verdrahtungs-Leitungen lediglich zur Repräsentation einer algebraischen Struktur missbraucht werden. Andere Arten der Evaluation, insbesondere ohne Veränderung eines gegebenen statischen Netzwerkes, dürften ebenfalls möglich sein. Eine sehr einfache Möglichkeit dazu besteht darin, eine Repräsentation des aktuellen Laufzeit-Graphen innerhalb des statischen Netzwerkes mittels generischer Operationen zu genau derjenigen Stelle zu schicken, an der die nächste Substitution durchgeführt wird. Andauerndes Herumkopieren unveränderter Teilgraphen lässt sich ggf. durch geeignete Strategien vermeiden, deren konkrete Ausarbeitung den Rahmen dieser Arbeit bei weitem sprengen würde, da auch Funktionen höherer Ordnung korrekt behandelt werden müssen. Es geht im Rahmen dieser Arbeit nur darum, prinzipielle Zusammenhänge und Möglichkeiten aufzuzeigen.

In der Praxis wird man den hier aufgezeigten Zusammenhang mit funktionalen Paradigmen eher nicht in der hier vorgestellten Form einsetzen (da Nester nicht als Konkurrenz zu etablierten Laufzeitsystemen für funktionale Berechnungen konstruiert wurden). Einige Formen funktionaler Denk- und Konstruktionsmethoden dürften jedoch durchaus wertvolle praktische Anwendungen haben. Beispielsweise können universell einsetzbare Interpreter[27] mit funktionalen Paradigmen arbeiten, die konzeptuell zur hier propagierten Statuslosigkeit besonders gut passen. Allerdings hat man in der Betriebssystem-Praxis eher mit „hemdsärmeligen“ Interpretern wie beispielsweise dem Perl-Interpreter [Haj00] zu tun, die intern ebenfalls bereits einiges aus der Welt der funktionalen Programmierung gelernt und übernommen haben. Realisiert man universelle Interpreter als Baustein, dann lässt sich der Programmcode auf natürliche Weise von den zu bearbeitenden Daten durch getrennte Eingänge separieren. Damit lassen sich beispielsweise Prototypen komplexer Bausteine sehr schnell entwickeln.

Bausteine und ihre Verdrahtungen sind für sich genommen bereits universelle Konstrukte, die sich beispielsweise auch mit den Hotz'schen X-Kategorien [Hot74] beschreiben lassen.

6.5. Zusammenhang mit Datenbank-Schemata

Relationale Datenbank-Schemata lassen sich mittels Nestern und Bausteinen erzeugen, umwandeln, und in Datenbank-Instanzen umsetzen, mit denen sich die später instantiierten Daten verarbeiten lassen. Dieser Abschnitt skizziert eine mögliche Umsetzung dieser Idee:



intensiv mit der Aspektorientierung in objektorientiertem Kontext befassen.

[27] Ein klassiches Beispiel universeller Interpreter-Konstrukte höherer Ordnung ist der #!-Operator zusammen mit der Argumentübergabe an Prozesse in Unix, mit dem sich beliebige Interpreter starten lassen, die ihrerseits wieder beliebige weitere Operationen auf beliebigen Daten ausführen können. Eine Nachbildung durch Bausteine ist leicht möglich.

Im Nest `table` werden Datenbank-Tabellen und Indizes abgelegt, das zugehörige (Teil-)Schema wird im Meta-Nest bzw in den Attributen in einer geeigneten Repräsentation gehalten. Durch den Baustein `gen_table` wird ein reines Schema `schema`, das ein oder mehrere Metabeschreibungen von Tabellen enthält, in ein oder mehrere[28] entsprechende Tabellen instantiiert. Der Status wird im `tmp`-Eingang (persistent) gehalten. Es wird dafür gesorgt und geprüft, dass die instantiierten Datensätze sowohl syntaktisch als auch semantisch dem Schema-Aufbau entsprechen.

Datensätze fester Länge lassen sich am einfachsten als zusammenhängendes Array abspeichern. In diesem Fall braucht das adjungierte Nest nicht unbedingt die Grenzen jedes einzelnen Datensatzes darzustellen, da diese Information leicht berechnet werden kann. Bei variabler Datensatz-Länge ist die Repräsentation lückenlos liegender Pakete (vgl. Abschnitt 3.4.1) im adjungierten Nest angebracht. Falls aus Gründen der Uniformität diese Art der Darstellung auch bei festen Datensatzlängen gefordert wird, lässt sich der Inhalt des adjungierten Nestes intern auf virtuelle Weise berechnen, ohne Platz im persistenten Speicher zu verbrauchen. Eine weitere Möglichkeit besteht darin, die Bestandteile von Datensätzen mit fester Länge von denjenigen mit variabler Länge zu trennen, so dass eine performante Array-Darstellung der Datensätze mit Verweisen auf einen Heap-Bereich entsteht, der die variablen Teile enthält; derartige Verweise entsprechen konzeptionell den Datensatz-Beschreibungen im adjungierten Nest und sollten aus Gründen der Uniformität an einer Stelle zentralisiert werden; das Konzept des adjungierten Nestes eignet sich hierfür besonders[29]. Um eine gute Kombinierbarkeit von Bausteinen zu erhalten, sollten diese Konventionen einheitlich festgelegt werden.

Über die Sortierung von Datensätzen wurde bisher nichts gesagt. Es ist möglich, eine Tabelle gleich von vornherein nach einem Primärschlüssel sortiert zu halten. Da Sekundärschlüssel jedoch unabhängige Sichten auf die Daten erlauben, ist es günstig, die Schlüssel unabhängig von ihrer Art und Anzahl ebenfalls im Nest zu halten. Eine Schlüssel-Repräsentation kann beispielsweise als Array von Index-Nummern der Datensätze erfolgen, die nach dem Schlüssel sortiert sind. Einfügungen und Löschungen von Datensätzen werden dann durch `move`-Operationen auf den jeweiligen Arrays bewerkstelligt. Natürlich ist es ebenfalls möglich, klassische Repräsentationen wie B^+-Bäume zu verwenden, doch sind diese wesentlich komplizierter aufgebaut und schwerer zu interpretieren. Da Nester vorzugsweise universelle Generizität mittels sehr einfacher Datenstrukturen repräsentieren sollten, halte ich die Verwendung sortierter Arrays für günstiger, zumal eine billige `move`-Operation zur Verwaltung dieser Array-Strukturen eingerichtet werden kann. Performante Suchstrukturen wie B-Bäume werden dadurch nicht überflüssig; sie werden lediglich an anderer Stelle einer Baustein-Hierarchie eingesetzt, nämlich bei der Erzeugung dynamischer Nester (Baustein `map` in Abschnitt 4.1.3).

Die Operationen der Relationen-Algebra wie Selektion, Projektion und Kartesische Produktbildung durch verschiedene `join`-Varianten lassen sich direkt durch Bausteine darstellen und ausführen. Details werden weiteren Arbeiten zur Integration von Datenbanken und Betriebssystemen überlassen. Einige weitere Ideen hierzu finden sich in [ST03b].

[28] Mehrere Tabellen, Indizes etc. sollten sich durch Vorreservierung genügend grosser Adressbereiche aus mindestens 64 Bit Adressraum-Gesamtgröße herausschneiden lassen, so dass Verschiebungen ganzer Tabellen gar nicht oder nur extrem selten benötigt werden.

[29] Eventuell wäre es eine Idee, adjungierte Nester zum adjungierten Nest einzuführen. Das adjungiert-adjungierte Nest enthält dann die Beschreibung ganzer Tabellen und Indizes, das einfach adjungierte Nest die konkrete Beschreibung der jeweiligen Tabelle.

7. Instanz-Orientierung

In Abschnitt 6.2.2 wurden grundlegende Unterschiede zwischen der hier vorgestellten Architektur und der Objektorientierung beschrieben. Diese Unterschiede geben Anlass, ein allgemeines Programmierparadigma zu formulieren:

> *Instanz-Basierung* ist ein Programmierparadigma, bei dem nichttriviale konkrete Beziehungen zwischen Laufzeit-Instanzen ausschließlich von einer unabhängigen weiteren Instanz verwaltet und überwacht werden.

Alternative Definition:

> Bei der Instanz-Basierung wird das Lokalitätsprinzip auch bei der Verwaltung und Überwachung der konkreten Laufzeit-Instanz-Beziehungen beachtet.

In Anlehung an die ausführlichere Darstellung in [ST04b] wird nun definiert:

> *Instanz-Orientierung* ist Instanz-Basierung mit der zusätzlichen Möglichkeit, *mehrere Sichten* auf die Laufzeit-Beziehungen gleichzeitig zu haben.

Hierzu einige Anmerkungen und Begriffsklärungen: Ein *Programmierparadigma* ist lediglich eine systematische Denk- und Anschauungsweise für menschliche Betrachter, die sich auf die Konstruktion von Software bezieht. Eine *Instanz* ist ein gedanklich oder konkret leicht abgrenzbares System mit einer zuweisbaren *Identität*, das aus einer *Schablone* (Template, Klasse, Makro, etc.) automatisch generiert werden kann. Bei einer *Laufzeit*-Instanz findet die Instantiierung dynamisch zur Laufzeit eines übergeordneten Systems (z.B. Betriebssystem oder Datenbank oder Laufzeitsystem einer Programmierumgebung) statt. Eine *konkrete* Beziehung zwischen Instanzen ist eine Beziehung, die von den beteiligten *Individuen* (Identitäten) abhängt. Eine *Beziehung* ist die notwendige Kenntnis von (nichttrivialer) Information über eine andere Identität. *Referenzen* sind spezielle Beziehungen zwischen Instanzen, wobei eine Instanz mindestens Kenntnis von der Identität der anderen hat oder leicht erlangen kann.

Im allgemeinen braucht die Verwaltungs-Instanz kein konkretes Objekt wie z.B. eine Baustein-Instanz zu sein, sondern kann z.B. auch als Aspekt der Aspektorientierten Programmierung (AOP) modelliert werden (vgl. Abschnitt 6.3).

Ein Beispiel[1] für Instanz-Orientierung ist die in Abschnitt 4.2.2 vorgestellte Methodik auf der Strategie-Ebene. Verdrahtungs-Leitungen spielen die Rolle von Referenzen zwischen Baustein-Instanzen, und zwar die *einzige* Form von Referenzen. Eine Umgehung dieser Art von Referenzen zwischen den Instanzen (z.B. in Form von Pointern) ist nicht vorgesehen bzw wird als Fehler behandelt. Die Verdrahtungs-Leitungen werden von `control`-Instanzen verwaltet und auf Einhaltung der hierarchischen Struktur überwacht. Eine Instanz hat keine notwendige Kenntnis über die mit ihr verbundenen fremden Identitäten. Die Strategie-Ebene enthält *Transformator-Bausteine*, mit deren Hilfe mehrere Sichten auf die Verdrahtungs-Graph-Strukturen herstellbar sind.

Die gesamte Verantwortung für das Konstruieren der Instanzen und für das Einrichten von Beziehungen wird separat gekapselt. Ein Beispiel hierfür sind separate Kommando-Kanäle für `control`, in die `strategy_*`-Instanzen eingeschleift werden, wobei die verwalteten Baustein-Instanzen selbst keinen *direkten* Einfluss darauf haben, wann sie konstruiert oder destruiert werden oder mit wem wie aus welchen Gründen verbunden werden; in manchen Einsatz-Szenarien kann dieser Einfluss auch vollständig unterbunden sein[2]. Die `strategy_*`-Instanzen übernehmen hierbei einen Teil der Funktionalität des Gesamtsystems, beispielsweise durch Bereitstellen von höherwertigen virtuellen Konstruktor-Operationen.

Grundsätzlich ist die Instanz-Orientierung davon unabhängig, welches Kommunikationsparadigma verwendet wird; zwar ist die hier verwendete (direktionale) verbindungsorientierte Kommunikation besonders geeignet, es sind aber auch verbindungslose Realisierungen möglich. Auch die Art der verwendeten Generizität ist prinzipiell davon unabhängig; kompositorische Generizität verbesser jedoch im Regelfall die Anwendungsmöglichkeiten der Instanz-Orientierung stärker als die Erweiterungs-Generizität.

7.1. Historischer Rückblick

Im historischen Rückblick ist es interessant zu ergründen, wie sich die Objektorientierung und ihre Anwendung in Betriebssystemen entwickelt hat. Der als Erfinder der Objektorientierung zu honorierende Dahl betrachtet in [DDH72, Kapitel III] in bezeichnender Weise „hierarchical *program* structures“, wählt also als Betrachtungsebene den *statischen* Programmtext. Noch heute wird in der Hauptströmung der Objektorientierung mehr kritische Aufmerksamkeit den Strukturbeziehungen auf Klassen-Ebene geschenkt als denjenigen auf Objekt-Ebene (vgl. schwache Kontrolle von Referenzen, Abschnitt 6.2.2). Die interessanterwei-

[1] Ein Beispiel für *Ansätze* von Instanz-Basierung sind solche Signal-Slot-Mechanismen [sig] in objektorientierten Systemen, bei denen die Slots den einzelnen Objekt-Instanzen zugeordnet sind und trotzdem die Registrierung von Handlern durch eine separate Laufzeit-Instanz vorgenommen und auf Fehler überprüft wird. Signal-Slot-Mechanismen werden in objektorientierten Entwürfen meistens nicht durchgehend als *alleiniges* Strukturierungsmittel für Instanz-Beziehungen eingesetzt; damit sind zwar die Mechanismen vorhanden, lassen sich aber umgehen (z.B. durch Einsatz von Pointern) und werden dadurch ihrer spezifischen Vorteile für die Sicherheit und Zuverlässigkeit beraubt.

[2] Der Einfluss der verwalteten Instanzen auf ihre Existenz und ihre Beziehungen muss lediglich *unterbindbar* sein, nicht unbedingt in allen Fällen unterbunden werden. Ein Unterbinden hat große Vorteile in sicherheitskritischen Anwendungen; die Möglichkeit dazu unterscheidet sich grundlegend von *reflektiven Architekturen* [Mae87], bei denen die *Selbst*-Beeinflussung und -Abänderung als wesentliche Motivation gesehen wird.

se von Dahl noch als „program concatenation“ bezeichnete Vererbung wird in [DDH72, Kapitel III] zum Aufbau einer total-geordneten *statischen* Klassen-Hierarchie auf dem Programmtext verwendet, die sich meines Erachtens in einem Betriebssystem nur zur *einmaligen* Instantiierung eines „hierarchisch gegliederten *Programms*“ eignet, da sie auf statischen Beziehungen des erstellen *Programm*codes fußt und nicht auf Laufzeit-Beziehungen zwischen Objekt-Instanzen.

Parnas weist in [Par74] auf verschiedene Arten von hierarchischer Gliederung hin. Er unterscheidet die Programm-Hierarchie, die Habermann-Hierarchie, Ressourcen-Zuteilungs-Hierarchien (bei denen er die Inflexibilität bei der Ressourcen-Zuteilung bemängelt; zur Lösung vgl. Abschnitt 5.2), und Schutz-Hierarchien. Parnas macht bewusst, dass die statische Programm-Hierarchie unabhängig von der Habermann-Hierarchie ist, die die Laufzeit-Auftrags-Beziehungen „gives work to“ zwischen den *Prozessen* (bzw. *virtuellen Maschinen* [HFC76]) in eine hierarchische Struktur zu bringen sucht. Weder Parnas [Par74] noch Habermann [HFC76][3] beschäftigen sich ausdrücklich mit dem Problem, wie die Habermann-Hierarchie in einem *dynamischen* System einzuhalten ist, in dem jederzeit neue Prozesse bzw. virtuelle Maschinen hinzu kommen können (bzw. *Schichten* wie in [Dij68, Lag75]; vgl. Begriff der Meta-Hierarchie in [YTT89]); vgl. die Problematik dynamischer Umverdrahtungen in Abschnitt 6.4. In den 1970er Jahren wurde offenbar von einer konstanten oder zumindest a priori beschränkten Hierarchie-Tiefe ausgegangen (vgl. statische Zuteilung von Prozess-Instanzen in [Han77]).

Im hier vorgestellten Baukasten-Ansatz können dynamische Hierarchien mit potentiell unbeschränkter Tiefe vorkommen. Im Unterschied zu den von Parnas genannten Hierarchie-Kriterien werden hier Baustein-Instantiierungen als *eigenständiges Grundkonzept*[4] mit einer eigenständigen *Verdrahtungs-Hierarchie* angesehen, wobei Verdrahtungen hierarchische Relationen explizit modellieren und *rekursive* Instantiierungen möglich sind. Separat kontrollierte *hierarchische* Beziehungen zwischen Instanzen stellen einen Spezialfall der Instanz-Orientierung dar.

Welche *Sorte* von hierarchischer Beziehung durch eine Verdrahtung ausgedrückt wird, ist ein *freier Parameter* bei den Entwurfsentscheidungen. Im Sinne von Parnas leistet eine Baustein-Instantiierung mit anschließender Verdrahtung je nach konkretem Fall möglicherweise einen Beitrag zur Habermann-Hierarchie[5](da Prozess- oder Kontrollfluss-Instantiierungen als *Folgeeffekte* entstehen können), oder zur Ressourcen-Hierarchie, oder je nach Baustein-Typ Beiträge zur Schutz-Hierarchie, oder zu mehreren dieser Hierarchie-Begriffe gleichzeitig. Eine Verdrahtungs-Hierarchie fungiert in dieser Sichtweise primär als *syntaktisches Konstrukt*, dem sich verschiedene *semantische* Hierarchie-Kriterien unterlegen lassen. Man kann die semantischen Hierarchien als *Vergröberung* der Verdrahtungs-Hierarchie ansehen, wenn ihre Struktur mit der syntaktischen Struktur *verträglich* ist (was z.B. bei der Schutz-Hierarchie nicht automatisch gegeben sein muss). Auf Programmcode- oder Modul-Ebene wird nur die Einhaltung des Lokalitätsprinzips durch strikte Trennung zwischen den Implementierungen verschiedener Baustein-Typen gefordert, jedoch nicht unbedingt eine Hierarchie (wodurch die Benutzung gemeinsamer Programm-Module oder -Bibliotheken nicht ausgeschlossen wird).

Wenn man die hier vorgestellte Architektur mit der Objektorientierung vergleicht, dann könnte man sie als Umsetzung von „hierarchical *object* structures“ (im Sinne von Hierarchie-Beziehungen *zwischen* Objekten $\cong$ Baustein-Instanzen) charakterisieren, während die Objektorientierung durch „hierarchical *class* structures“ zu charakterisieren wäre. Die Verallgemeinerung auf beliebige, aber zentral kontrollierte Instanz-Beziehungen wird daher mit dem Begriff „Instanz-Orientierung[6]“ umschrieben.

7.2. Übertragung nach OO und AOP

Tiefere Diskussionen der Instanz-Orientierung außerhalb von Betriebssystemen, insbesondere mögliche Implikationen für Programmiersprachen und Laufzeitsysteme, liegen außerhalb der Reichweite dieser Arbeit. Für zukünftige Forschungen und Untersuchungen sollen jedoch ein paar Bemerkungen angefügt werden.

Meiner Ansicht nach genügt es nicht, in OO-Programmiersprachen die Pointer oder Referenzen abzuschaffen und statt dessen ausschließlich mit Signal-Slot-Mechanismen [sig] oder anderen Formen anonymer Kommunikation zu arbeiten. Instanz-orientiertes Vorgehen muss beim Entwurf beginnen (oder noch besser schon bei der Anforderungs-Analyse bzw. bei der oftmals in der Praxis vernachlässigten *Problemanalyse*).

Weitere zu klärende Fragen betreffen die *Granularität*, mit der die Instanz-Orientierung vorteilhaft eingesetzt werden kann. Wenn man grundlegende Datenstrukturen wie lineare Listen oder Bäume mittels Instanz-Orientierung modelliert, könnte es leicht geschehen, dass man starke Performanz-Verluste durch den Einsatz der zentralen Kontroll-Instanz zu beklagen hat (sofern nicht etwa zukünftig besondere Hardware-Unterstützung für das Paradigma

[3] In [HFC76] wird zwischen „static address space“ (SAS) und „dynamic address space“ (DAS) unterschieden. Ein DAS entspricht jedoch konzeptuell einem Aktivierungsblock, da seine Lebensdauer auf die Dauer eines ASCALL-Aufrufes (bis zum ASRETURN) beschränkt ist. Auch der Begriff „instance“ wird im Sinne der Aktivierung eines Funktionsaufrufes verwendet. Dies ist etwas grundlegend anderes als eine Objekt-Instanz, deren Lebensdauer diejenige eines Funktionsaufrufes übersteigen kann.

[4] Methodisch gesehen wird die Verdrahtungs-Hierarchie ähnlich einem Axiomensystem einfach gegeben und vorausgesetzt. Andere Hierarchie-Beziehungen ergeben sich daraus als Folgeeffekt.

[5] Die Verdrahtungs-Hierarchie auf Bausteinen ist am ehesten mit der Habermann-Hierarchie zu vergleichen, da man beide als Ausprägung einer Relation „gives work to“ oder „may give work to“ sehen kann. Die Verdrahtungs-Hierarchie benutzt die Zusammenfassung von Programmcode und Daten zu „Objekten“ als Grundelemente, während die Habermann-Hierarchie auf „virtuellen Maschinen“ bzw auf dem klassischen Prozess-Begriff fußt, der zusätzlich (oder in manchen älteren Definitionen auch ausschließlich) einen Kontrollfluss enthält. Kontrollflüsse werden hier als unabhängig angesehen und nicht für Hierarchie-Beziehungen in Betracht gezogen.

[6] Rein vom Namen „*Objekt*-Orientierung“ her könnte man die hier vorgestelle Architektur mit diesem Begriff bezeichnen. Dann sollte man jedoch zur Abgrenzung die bisher als Objektorientierung bezeichnete Methodik in „Klassen-Orientierung“ (KO) umbenennen. Eine Umbenennung lange etablierter Begriffe dürfte jedoch zur Verwirrung führen und beträchtliche Akzeptanz-Probleme in der ganzen Welt mit sich bringen. Daher wurde der Name „Instanz-Orientierung“ (IO) gewählt.

der Instanz-Orientierung entwickelt und auf breiter Front eingesetzt wird). Die Instanz-Orientierung zeigt ihre Vorteile möglicherweise nur auf höherer Granularitäts-Ebene, etwa auf der Ebene von *Komponenten* (vgl. [Szy98]) oder leicht unterhalb. Man könnte die Instanz-Orientierung eventuell auch als Erweiterung von Szyperskis Komponenten-Begriff [Szy98] auf *instantiierbare* Komponenten[7] sehen, wobei die Instantiierung von einer Art „Middleware“ oder „Framework“ vorgenommen und überwacht wird (allerdings passen diese angestammten Begriffe nicht ganz).

Wenn man die Instanz-Orientierung als Konzept zur logischen Separation höher-granularer Einheiten ansieht, dann kann man *innerhalb* einer derartigen Instanz durchaus Pointer und Klassen-Hierarchien benutzen (wobei die Abgrenzungs-Problematik eingehend zu untersuchen wäre). Die Instanz-Orientierung wäre damit eine sinnvolle Ergänzung zur Objektorientierung, die sich unabhängig davon auch zusätzlich durch die Aspektorientierung (AOP) [K+97b] ergänzen lässt. Falls sich jedoch Mittel und Wege finden lassen sollten, um Referenzen mit für Anwendungen akzeptabler Performanz ausschließlich zentral verwalten zu können, wäre diese Lösung aus methodischen Gründen zu bevorzugen.

Obwohl man die separate Kontrolle von Instanz-Beziehungen als „crosscutting concern“ und damit als Aspekt in der AOP modellieren kann, verträgt sich dies nicht gut mit aktuellen Realisierungen von AOP wie AspectJ [K+01], bei denen Referenzen als Standard-Mittel zum Ausdruck von Beziehungen zwischen Objekten benutzt werden. Der Grad an Separation, der von der Instanz-Orientierung gefordert wird, lässt sich eher mit RPC zwischen getrennten Adressräumen bzw. Schutzbereichen vergleichen. Eine 1:1-Verknüpfung von Instanzen mit Schutzbereichen kann jedoch schwer wiegende Performanz-Nachteile mit sich bringen. Daher sollte man die Instanz-Orientierung nur zur *programmiersprachlichen Repräsentation* von Trennungs-Konzepten benutzen, und die Abbildung auf reale Verteilungs- und Schutz-Szenarien davon separieren. Die Granularität der Instanz-Orientierung darf dann auch unterhalb üblicher Middleware-Aufteilungs-Granularitäten liegen und erschließt damit einen größeren Anwendungsbereich.

Die Modellierung von *abstrakten Schnittstellen* zwischen Instanzen ist ein besonders wichtiger Fokus der Instanz-Orientierung. In der hier vorgestellten Betriebssystem-Architektur wurde dieses Problem auf unterster Ebene durch einheitliche Verwendung der immer gleichen *universellen* Nest-Schnittstelle entschärft, jedoch auf höherer Ebene durch ein Laufzeit-Typkonzept bei generischen Operationen (Kapitel 6) nachgerüstet. Da sich i.A. eine gewisse Vielfalt unter den Schnittstellen nicht immer vermeiden lässt, sind ähnliche oder andere Konzepte bei Anwendungsprogrammen von immenser Wichtigkeit.

Eine mögliche Kombination von Instanz-Orientierung mit der Objektorientierung könnte folgendermaßen skizziert werden: eine Art Klassen-Modellierung findet überwiegend bei den Schnittstellen statt; diese werden als Schnittstellen-Klassen bezeichnet und entsprechen konzeptuell den Nestern, auch wenn sie ganz andere Funktionalitäten repräsentieren. Darüber hinaus gibt es eine weitere Art von Klassen, die Baustein- oder Objekt-Klassen genannt werden können, zwischen denen von der Schnittstellen-Klassen-Hierarchie unabhängige Beziehungen herrschen können. Über Baustein-Klassen ist von außen nichts außer den darin enthaltenen Schnittstellen-Instanzen bekannt. Es existiert keine unabhängigke Instantiierungs-Operation für Schnittstellen-Objekte, sondern immer nur in Kombination mit Baustein-Instanzen. Schnittstellen-Instanzen sind damit immer an die Existenz einer Baustein-Instanz gekoppelt, für die sie gelten sollen. Referenzen existieren nur zwischen derartigen Schnittstellen-Instanzen. Sie entsprechen konzeptuell den Verdrahtungs-Leitungen und werden gemäß dem Instanz-Orientierungs-Paradigma von einer weiteren unabhängigen Instanz verwaltet, so dass die Einführung mehrerer Sichten auf das System ermöglicht wird. Details sind weiteren Arbeiten überlassen.

Eine weitere Frage ist, wie man die Vorteile der universellen Generizität der Nest-Abstraktion auf allgemeine verteilte instanzorientierte Systeme übertragen kann. Nester ermöglichen die gemeinsame Nutzung von darin enthaltenen fein-granularen Objekten durch mehrere verteilte grob-granulare Instanzen. Konzeptuell ermöglichen Nester Referenz-Beziehungen auf *Mengen* von Objekten, sogar auf hierarchisch geschachtelten Mengen von Objekten. Möglicherweise wäre dies ein Feld, wo die in Betriebssystemen entwickelten Methoden und Verfahren vorteilhaft auf Anwendungsprogramme ausgedehnt werden könnten.

[7] Die Instanz-Orientierung verlangt im Unterschied zu Szyperski nicht zwingend, dass Instanzen *statusbehaftet* oder *statuslos* implementiert werden müssen.

8. Multiversion-Modelle

In diesem Kapitel sollen Überlegungen entwickelt werden, wie sich die Funktionalität von Transaktionen auf einfache generische Weise in die Nest-Schnittstelle integrieren lässt. Dadurch wird Transaktions-Fähigkeit und der gleichzeitige Umgang mit mehreren transaktionen Sichten in allen Schichten eines Betriebssystems ermöglicht; die Herstellung mehrerer transaktionaler `singleuser`-Sichten für Endbenutzer wird durch relativ einfache `adaptor`-Bausteine vorgenommen.

Ziel ist nicht die vollständige Abdeckung aller nur denkbarer `multiversion`-Modelle, sondern die exemplarische Auswahl und die Darstellung einiger (hoffentlich) für praktische Entwürfe geeigneter Modelle, die hinsichtlich der gängigsten Transaktions-Semantiken universelle Generizität bereit stellen sollten.

Historisch betrachtet hatten Transaktionen [BHG87, LS87, GR93, Elmbe] das Ziel, trotz paralleler Aktivitäten jedem Konsumenten ein `singleuser`-Modell bereit zu stellen, so dass sich Programmierer nicht um Parallelität zu kümmern brauchten. Nach ausführlicher Untersuchung von Synchronisation und Rücksetzen (recovery) wurde entdeckt, dass man trotz `singleuser`-Ausgangsbasis mit *Versionen* von Datenobjekten zu tun hatte; ohne Versionierung ist Rücksetzen nicht denkbar. Daraufhin wurden Multiversions-Varianten von Transaktionen [VGH93] untersucht, insbesondere Zeitstempel-Verfahren auf Basis von Versionen (time stamp ordering, time domain addressing). Im folgenden setze ich die Kenntnis der wesentlichen Grundkonzepte aus der Datenbank-Welt beim Leser voraus.

Für unsere Zwecke sind *abstrakte* generische Transaktions-*Mechanismen* an der Nest-Schnittstelle gefragt, mit denen möglichst universell verschiedene Transaktions-Modelle als *Strategie* durch Bausteine implementierbar sind.

8.1. Anforderungen

In Anwendungsbereichen wie z.B. Finanzbuchhaltung gibt es gesetzliche Anforderungen, die das spätere Nachvollziehen (Lesen) aller durchgeführten Aktionen (Operationen) *in der zeitlichen Reihenfolge der Durchführung* erzwingen (lückenlose zeitliche Dokumentation aller Operationen). Darüber hinaus existieren davon unabhängige *sachliche Kriterien* als zeitliche Ordnungsmerkmale, unter denen die durchgeführten Operationen ebenfalls nachträglich zugreifbar sein müssen. Ein typisches Beispiel aus der Finanzbuchhaltung ist der Unterschied zwischen dem Beleg-Datum eines Belegs und dem Verbuchungs-Datum: die Reihenfolge der Verbuchung (Eintragung) von Belegen muss nicht notwendigerweise in der Reihenfolge ihrer Geltung (Beleg-Datum) erfolgen. Im Journal einer Finanzbuchhaltung müssen alle Einträge mindestens in der Reihenfolge der Eingabe, in den Konto-Auszügen sollten sie zusätzlich in der Reihenfolge ihrer sachlichen Geltung abrufbar sein. Diese Anforderungen lassen sich auf folgende Weise verallgemeinern:

1. Jede jemals durchgeführte Operations-Instanz auf einem Nest muss anhand eines Merkmals (z.B. Zeitstempel) wieder eindeutig auffindbar sein
2. Es können mehrere von einander unabhängige Zeit-Bereiche notwendig sein

Punkt 1 bedeutet, dass ein „Elefantengedächtnis[1]“ implementierbar sein muss, da es Anwendungen gibt, die dieses aufgrund gesetzlicher Anforderungen verlangen. Falls bei anderen Anwendungen der Overhead stört, lassen sich eingeschränkte Modelle ableiten, bei denen nicht mehr benötigte Informationen gezielt oder automatisch „vergessen“ werden.

Punkt 2 scheint auf den ersten Blick zu bedeuten, dass aufgrund gesetzlicher oder sachlicher Anforderungen ein oder mehrere Totalordnungen auf ein oder mehreren Zeit-Bereichen zu implementieren wären. Dies wäre jedoch insbesondere in verteilten Systemen sehr hinderlich. Die folgenden Überlegungen befassen sich hauptsächlich mit diesem Problem.

8.2. Allgemeines Modell

Eine Möglichkeit wäre die Einführung von zeitlichen Halbordnungs-Konzepten (vgl. [Lam78]) in die Nest-Schnittstelle. Eine Totalordnung würde sich dann als Spezialfall dort ergeben, wo dies aufgrund von Anforderungen notwendig wäre. Diese Möglichkeit wird hier aufgrund ihrer hohen Komplexität nicht weiter untersucht[2]. Statt dessen schlage ich folgende *Betrachtungsweise* des Problems vor:

Die Idee der Festlegung einer *zeitlichen Totalordnung* zwischen Operations-Instanzen impliziert nicht notwendigerweise, dass der *Festlegungs-Vorgang* selbst in totalgeordneter zeitlicher Reihenfolge geschehen muss. Es wird lediglich verlangt, dass das *Endergebnis* eine zeitliche Totalordnung ergeben muss; *wann dieses Endergebnis jedoch berechnet wird*, ist eine davon unabhängige Frage.

Wir unterscheiden damit zwischen virtueller Zeit und realer Reihenfolge von Operations-Ausführungen. Weiterhin benutzen wir *lazy evaluation* bei der Festlegung der virtuellen zeitlichen Reihenfolge von Operations-Instanzen[3].

[1] Bekannte Realisierungen sind z.B. die Logs von Datenbanken, die gelegentlich vollständig aufbewahrt werden und als SQL-Tabelle dargestellt werden, so dass man in ihnen wie in anderen Tabellen suchen kann.

[2] Konzeptuell wäre dies die Verwendung Lamportscher Uhren [Lam78] auch in persistenten Speichern als dauerhaftes Modell der Darstellung zeitlicher Relationen zwischen Operationen.

[3] Beim Time-Warp [Jef85] wird zwar ebenfalls eine virtuelle Zeit-Achse verwendet, doch schreiten die Operations-Ausführungen auf dieser Achse monoton vorwärts und werden nur beim Rollback gelegentlich *intern* zurück gesetzt. Im Unterschied dazu erlaubt das hier propagierte Modell beliebige Rückdatierungen von Operationen, die unabhängig von evtl. vorkommenden Rollbacks sind; weiterhin wird die virtuelle Zeit-Achse mit der Orts-Achse zu einer zweidimensionalen Ebene verknüpft.

Solange eine Totalordnung noch nicht endgültig festgelegt worden ist, können (temporäre) Halbordnungen vorkommen. Einmal gemachte Festlegungen lassen sich jedoch nie mehr revidieren. Man kann dies auch als konsequente Anwendung des Persistenz-Gedankens auf die zeitlichen Ordnungsrelationen ansehen, wobei ein Nest in dieser Betrachtungsweise mit zeitlicher Verzögerung nach und nach persistent werden kann.

Der Endzustand eines Nestes ist erst dann erreicht, wenn nicht nur die Daten-Inhalte und Parameter aller ausgeführten Operations-Instanzen, sondern auch die Zeitstempel der Operations-Instanzen intern festgelegt sind. Konsumenten erhalten niemals Kenntnis von halb-geordneten Zwischenzuständen; spätestens die Abfrage zeitlicher Relationen an der Nest-Schnittstelle führt automatisch zu einer Festlegung (und damit eventuell zu einer Zustands-Änderung im strengen Sinn, selbst bei einer „simplen“ Abfrage).

Die nähere Festlegung vorher (teilweise) unbestimmter zeitlicher Relationen wird *Präzisierung* genannt; alle zeitlichen Relationen zwischen Operationen eines Nestes werden zusammen gefasst *Präzisierungs-Zustand* genannt. Präzisierungs-Zustände müssen *konsistent* sein, d.h. bei einem verteilten System dürfen zwei verschiedene Abfrager keine widersprüchlichen Antworten zum gleichen Sachverhalt bekommen. Die Menge aller ausgeführten Operationen samt deren Parametern, jedoch ohne die zeitlichen Relationen, wird *Daten-Zustand* eines Nestes genannt.

Parallelität von Operationen ist insbesondere dann möglich, wenn keine *Abfrage* zeitlicher Relationen des Präzisierungs-Zustandes erfolgt (etwa weil zeitliche Ordnungen für das zu lösende Problem keine Rolle spielen). Falls niemals eine Abfrage zeitlicher Relationen zwischen Operations-Instanzen statt findet, dann kann ein persistenter Speicher im Extremfall lauter unkorrelierte Operations-Instanzen enthalten, deren zeitliche Reihenfolge nicht fest gelegt ist (eine Festlegung *darf* in diesem Fall natürlich trotzdem erfolgen, z.B. im Hintergrund mit geringer Priorität).

Ob zwei Operations-Instanzen eine zeitliche Relation zu einander besitzen, könnte mit Hilfe eines *Benutzungs-Modells* implizit fest gelegt werden. Beispielsweise könnte ein bestimmtes Benutzungs-Modell vorschreiben, dass sämtliche Operationen, die sich auf gegenseitig überlappende Adressbereiche eines Nestes beziehen, in eine Totalordnung gebracht werden müssen (vgl. `linearize_local` in Abschnitt 3.2). Es sind jedoch Fälle denkbar, in denen selbst diese schwach erscheinende Forderung nicht erfüllt zu sein braucht (beispielsweise wenn irrelevante Operations-Durchführungen einfach ignoriert und niemals mehr betrachtet werden). Da weiterhin sehr viele verschiedene Benutzungs-Modelle denkbar sind, schlage ich vor, keinerlei Annahmen über implizite Abhängigkeiten zwischen Operations-Instanzen in die Nest-Schnittstelle aufzunehmen, sondern jegliche zeitliche Abhängigkeiten *explizit* anzugeben. Die Realisierung unterschiedlicher Benutzungs-Modelle oder Transaktions-Semantiken wird dadurch zur Aufgabe konkreter Baustein-*Implementierungen*.

Das allgemeinste hier vorgeschlagene `multiversion`-Modell sieht so aus, dass zu *jeder* durchzuführenden Operations-Instanz in irgend einer Form angegeben werden muss, von welchen anderen („früheren“) Operations-Instanzen sie abhängt. Die Spezifikation der „früheren“ Operations-Instanzen braucht keine individuelle Bekanntschaft aller in Frage kommenden Operations-Instanzen vorauszusetzen, sondern kann durch verschiedene mengen- oder prädikatenorientierte Spezifikationsmechanismen erfolgen.

Aus der Vielzahl von Spezifikationsmechanismen für zeitliche Ordnungen wähle ich im folgenden eine bestimmte aus und überlasse die Untersuchung alternativer Mechanismen weiteren Forschungsarbeiten.

8.3. Zeitintervall-Modell

Die Grundidee besteht darin, als Zeitbasis eine *physikalische*[4] Lamportsche Uhr [Lam78] mit genügender Genauigkeit zu verwenden, jedoch den einzelnen Operations-Instanzen keine festen Zeit-*Punkte*, sondern Zeit-*Intervalle* (vgl. Begriff der *Linearisierbarkeit* in [HW90]) zuzuordnen. Im Unterschied zu fest gefügten Konsistenzmodellen wie der linearen Konsistenz, die Halbordnungen auch im Endergebnis zulassen, können Zeitstempel-Intervalle zu jedem beliebigen späteren Zeitpunkt nachträglich *verkleinert* (Spezialfall einer Präzisierung) werden: ein ursprünglicher Zeitstempel $[t_1, t_2]$ mit $t_1 \leq t_2$ lässt sich jederzeit in $[t_3, t_4]$ mit $t_1 \leq t_3$, $t_3 \leq t_4$ und $t_4 \leq t_2$ umwandeln.

Eine solche Verkleinerung *muss* immer dann durchgeführt werden, wenn sich beim *Vergleich* der Zeitstempel zweier Operations-Instanzen op_1 und op_2, geschrieben $t(op_1) = [t_1, t_2]$ und $t(op_2) = [t_3, t_4]$, eine gegenseitige Überlappung (d.h. $t_3 \leq t_2 \wedge t_1 \leq t_4$ oder $t_1 \leq t_4 \wedge t_3 \leq t_2$) ergibt. Durch die erzwungene Verkleinerungs-Operation wird sicher gestellt, dass anschließend entweder $t_2 < t_3$ oder $t_4 < t_1$ gilt[5], d.h. eine Kommutation der beiden Operations-Instanzen ist anschließend nicht mehr möglich. Solange die Zeitstempel der beiden Operations-Instanzen nicht durch eine implizit oder explizit ausgeführte Vergleichs-Operation miteinander verglichen werden, dürfen sich die Intervalle überschneiden, d.h. es braucht keine endgültige Festlegung der Reihenfolge der beiden Operationen zu erfolgen.

Für den Benutzer dieses Modells sieht alles so aus, als gäbe es eine totalgeordnete globale Zeit. Die interne Realisierung verwendet jedoch (temporäre) Halbordnungen. Die Präsentation als Totalordnung nach aussen unterscheidet dieses Modell von explizit auf Halbordnungen aufgebauten Modellen.

8.4. Container-Operationen: Locks

Bereits im `multiuser`-Modell (Abschnitt 3.3.5) dienten Locks zur Klammerung einer Menge von Zugriffs-

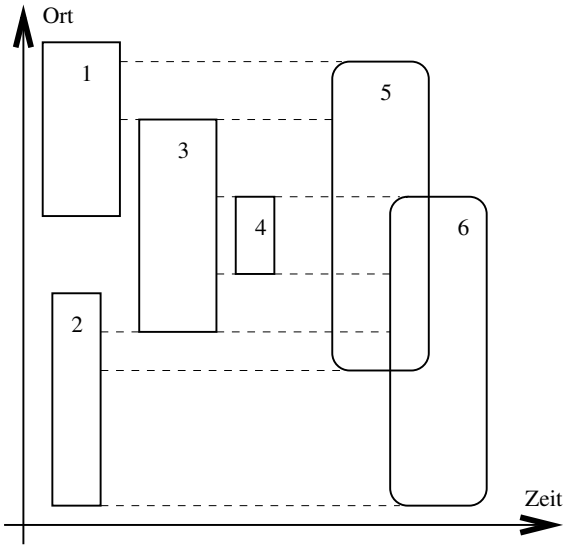
[4] Es geht auch mit logischen Lamportschen Uhren, nur muss der Konsument dann die „Bedeutung“ von Zeit-Werten in der Real-Zeit auf mühsame Weise selbst bestimmen.

[5] Bei dieser Darstellung wird still-schweigend voraus gesetzt, dass stets eindeutige Zeitstempel in einem verteilten System generiert werden. Eine wohlbekannte Methode hierfür ist das Anhängen von Knoten-Kennzeichen an den abgelesenen Wert einer Rechner-Uhr, so dass eventuell gleichlautende Uhrzeiten, die durch zufälliges „gleichzeitiges“ Ablesen von Uhren entstehen könnten, stets zu eindeutigen Zeit-Werten führt.

Operationen. In `multiversion`-Modellen werden Lock-Operationen analog dazu als zeitliche und räumliche *Container* aufgefasst, die eine *Menge* von Zugriffs-Operationen *beherbergen* können. Die Idee besteht darin, nur noch diese Container als Einheiten für Beziehungen zwischen Zugriffs-Operationen zu verwenden, und von den Details der im Container beherbergten Operationen und Operations-Arten zu abstrahieren.

Lock-Operations-Instanzen werden als persistente oder quasi-persistente[6] Objekte aufgefasst, die eine zeitliche und eine örtliche Dimension besitzen und einem Mandat als Eigentümer zugeordnet sind; verschiedene Transaktionen müssen unter verschiedenen Mandaten laufen. Die örtliche Ausdehnung ist der Bereich des virtuellen Adressraums des Nestes, auf den sich der Lock bezieht. Die zeitliche Ausdehnung entspricht den soeben vorgestellten Intervallen auf einer *virtuellen Zeitachse*, die unabhängig von der tatsächlichen Ausführungs-Reihenfolge besteht. Während die örtliche Ausdehnung weit gehend fest gelegt ist (nachdem sie z.B. einmal als spekulative Locks ausgehandelt wurden, vgl. Abschnitt 5.3), kann die zeitliche Ausdehnung nachträglich jederzeit verkleinert werden (Präzisierung). Ein Container darf nur solche Zugriffs-Operationen beherbergen, die sich auf einen Teil-Bereich der räumlichen und zeitlichen Dimension beziehen. Im Regelfall *erben* die von einem Container beherbergten Zugriffs-Operationen ihre zeitliche Dimension vom umgebenden Container.

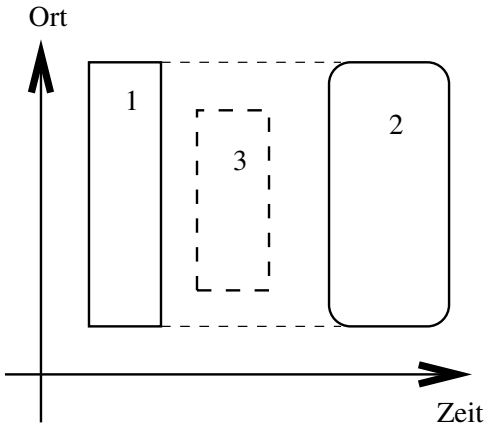
Im folgenden Bild werden Write-Lock-Instanzen durch Rechtecke mit spitzen Ecken, Read-Lock-Instanzen dagegen durch abgerundete Ecken dargestellt:



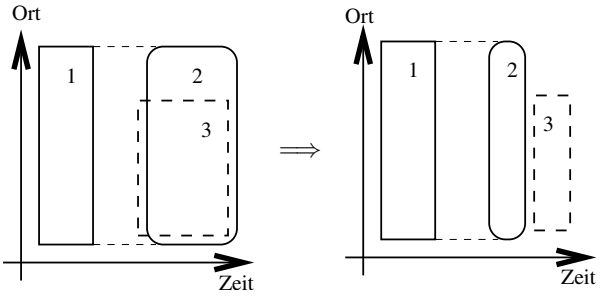
Verschiedene Write-Lock-Instanzen dürfen sich niemals gegenseitig überlappen, d.h. nicht die gleiche Fläche in der zeitlichen und örtlichen Dimension einnehmen. Im Konfliktfall lassen sich Write-Locks jedoch nachträglich immer so verkleinern, dass sie auf der Zeit-Achse nebeneinander passen (aufgrund der eindeutigen Zeitstempel, siehe Abschnitt 8.3). Read-Lock-Instanzen verschiedener Mandate dürfen sich dagegen gegenseitig überlappen.

Die Verhältnisse zwischen Read- und Write-Lock-Instanzen werden im folgenden Bild näher untersucht:

[6]Mit Quasi-Persistenz ist gemeint, dass grundsätzlich die Persistenz zwar angestrebt wird, jedoch später die gespeicherte Information durch gezieltes Vergessen vergröbert werden kann.



Die in den Kästchen eingezeichneten Nummern sollen die Reihenfolge[7] andeuten, in denen die jeweiligen Locks gesetzt werden sollen. Das Setzen des gestrichelt gezeichneten Locks 3 ist nicht möglich, weil er die durch gestrichelte Linien angedeutete „Sicht“ des vorher dagewesenen Locks 2 auf Lock 1 teilweise „verdecken“ würde. Falls die Reihenfolge von Lock 2 und 3 vertauscht würde, dann wäre ein Setzen aller drei Locks möglich, jedoch würde dann die Read-Lock-Instanz eine Sicht auf die kleinere Write-Lock-Instanz erhalten. Im folgenden Sonderfall ist eine Konfliktlösung durch nachträgliches Verkleinern möglich:



Es gilt folgender Grundsatz: Eine einmal gewährte (bzw. in Anspruch genommene[8]) Sicht eines Locks auf andere Locks darf nachträglich nicht verändert werden.

Die Frage ist, was man unter einer Sicht auf andere Locks verstehen soll. Intuitiv geht es um die bekannte Semantik von Speichern, nach der eine Lese-Operation bzw. ein Read-Lock den Zustand der *zeitlich letzten* davor liegenden Änderung (Write-Lock bzw. Schreib-Operationen) sehen sollte. Der Unterschied zur klassischen Semantik besteht lediglich darin, dass die Zeitachse ein virtuelles Gebilde ist, auf dem die Relationen „vorher“ und „nachher“ unabhängig von der tatsächlichen Ausführungsreihenfolge festgestellt werden (sofern eine global eindeutige Ausführungsreihenfolge überhaupt existiert).

[7]Das Konzept einer „globalen Reihenfolge“ von Operations-Ausführungen sollte in einem verteilten System nicht verwendet werden. Die Darstellung durch Reihenfolge-Nummern dient hier nur zur Illustration.

[8]Es sind allgemeinere Modelle denkbar, bei denen es nicht auf die Sichten zwischen Locks, sondern zwischen den darin enthaltenen Zugriffs-Operationen ankommt. Solange im Beispiel nur Lock 2 gesetzt, aber nicht gelesen wurde, könnte man Lock 3 noch gewähren. Das sich daraus ergebende kompliziertere Modell wird in dieser Arbeit nicht untersucht.

8.5. Kausale Abhängigkeiten

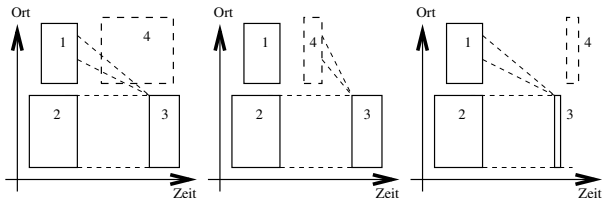
Die Verallgemeinerung der Idee einer Sicht einer Lock-Instanz auf andere Lock-Instanzen wird *direkte kausale Abhängigkeit* genannt. Die Verallgemeinerung besteht darin, dass nicht nur Abhängigkeiten wegen der (möglichen) Sicht auf andere Lock-Instanzen gleicher Adressbereiche als kausale Abhängigkeit gewertet werden, sondern darüber hinaus noch beliebige weitere kausale Abhängigkeiten in das (quasi-)persistente Präzisierungs-Zustands-Modell auf explizite Anforderung aufgenommen werden können.

Eine typische Anwendung für die Aufnahme extern spezifizierter kausaler Abhängigkeiten ist die Emulation der Herbrand-Semantik klassischer Transaktionen [VGH93]. Klassische „an der Syntax“ orientierte Transaktionen betrachten *jede* Schreiboperation als kausal abhängig von *allen* vorher durch die selbe Transaktion gelesenen Objekten. So genannte „semantische“ Transaktions-Modelle [Elmbe] versuchen dagegen, durch Kenntnisse über den inneren Aufbau der Transaktionen nur diejenigen kausalen Abhängigkeiten als Konsistenz-Kriterium in Betracht zu ziehen, die auch tatsächlich vorhanden sind und benutzt werden.

Wir stellen universelle Generizität bezüglich verschiedener Transaktions-Modelle dadurch her, dass wir die explizite Angabe kausaler Abhängigkeiten in der Nest-Schnittstelle verlangen. Der Sonderfall vollständiger kausaler Abhängigkeiten von allen je von einer Transaktion angefassten Objekte lässt sich relativ leicht als Strategie durch einen geeigneten Anpassungs-Baustein implementieren.

Konkret lässt sich dies so gestalten: Read-Lock-Instanzen erhalten automatisch eine direkte kausale Abhängigkeit von allen Write-Lock-Instanzen, die sich mit ihrem Ausschnitt aus dem Adressbereich überschneiden und die in der Zeitachse davor am nächsten liegen. Dies sind genau diejenigen Write-Lock-Instanzen, deren Versionen von der Read-Lock-Instanz aus sichtbar sind.

Bei Write-Lock-Instanzen muss der Aufrufer der Write-Lock-Operation eine Menge von Adressbereichen angeben. Aus dieser Menge von Adressbereichen wird die Menge der zeitlich nächstliegenden Write-Lock-Instanzen bestimmt, von denen eine direkte kausale Abhängigkeit angenommen werden soll. Diese Art der Abhängigkeit dient lediglich als Informationsübergabe für zusätzliche Bedingungen oder Eigenschaften wie beispielsweise die *Serialisierbarkeit*, deren Einhaltung eine Frage konkreter Implementierungs-Strategien ist. Die folgende Situation lässt sich beispielsweise auf verschiedene Arten lösen:



In der Ausgangs-Situation wurde Lock 3 als kausal abhängig von Lock 1 und Lock 2 definiert. Der neu hinzukommende Lock 4 überschneidet nur den Adressbereich von Lock 1, nicht jedoch den von Lock 3. Da er nicht gleichzeitig die Adressbereiche von Lock 3 und 2 überscheidet, ist die Zulässigkeit dieser Operation eine Frage der konkreten Strategie. Ich sehe folgende Möglichkeiten:

Man kann die Ausgangssituation (erstes Bild) zulassen. Dann muss jedoch persistent gemerkt werden, von welcher Version (nämlich Lock 1) die konkrete kausale Abhängigkeit besteht. Andernfalls würde eine Verkleinerung von Lock 4 (zweites Bild) in der klassischen Speichersemantik zu einem unbeabsichtigen Wechsel der Version führen, zu der nahcträglich eine ander (unbeabsichtigte) kausale Abhängigkeit besteht. Dieses Problem könnte auch dadurch verhindert werden, dass derartige Locks nicht gesetzt werden dürfen. Die spezielle Ausgangssituation lässt hierfür zwar wegen der zeitlichen Überschneidung von Lock 4 mit Lock 3 auch die spezielle Lösung aus dem dritten Bild zu, doch führt dies im allgemeinen zu einer Behinderung.

Die einfachste Lösung scheint mir zu sein, „überspringende“ kausale Abhängigkeiten[9] wie im Bild 1 im allgemeinsten Fall zuzulassen, indem sie auf den jeweiligen Lock-*Instanzen* (einschließlich zeitlicher Platzierung) und nicht ausschließlich auf den Adressbereichen aufgebaut sind. Es können jedoch weitere Zusatzbedingungen bzw Restriktionen für die Zulässigkeit solcher „überspringender“ kausaler Abhängigkeiten als besondere Strategie in Baustein-Implementierungen eingeführt werden; hierzu zählt insbesondere die Einhaltung der Serialisierbarkeit von Transaktionen.

8.6. Aktualitätsgrade

Der aufmerksame Leser wird sich bereits gefragt haben, wie `unlock`-Operationen in einem Modell funktionieren sollen, das alle Lock-Instanzen als (quasi-)persistente Objekte auffasst und sie im Extremfall für immer aufbewahren kann (womit z.B. in militärischen oder geheimdienstlichen Anwendungen jederzeit nachvollziehbar wäre, wer wann welche Read-Locks gesetzt und die zugehörigen Daten inspiziert hat).

Die als Container für Zugriffs-Operationen wirkenden Lock-Instanzen müssen irgendwann *abgeschlossen* werden. Abschließen bedeutet, dass der Container keine (nachträglich angelieferten) Operationen mehr für die Beherbergung akzeptiert.

Beim Anlegen wird jedem Container ein Mandat oder eine Menge von Mandaten als „Eigentümer“ des Containers zugeordnet. Der *Zustand* eines Containers wird beim Anlegen ebenfalls fest gelegt, kann jedoch später geändert werden und ist ein Wert aus der folgenden hierarchisch geordneten Aufzählung von Aktualitätsgraden:

`act_invalid` Der Container beherbergt ungültige (d.h. zu ignorierende) Operationen.

`act_destroyable` Der Container beherbergt ein vorläufiges Berechnungsergebnis, das jederzeit (evtl. asynchron) auch durch fremde Mandate (nicht nur durch das Eigentümer-Mandat) für ungültig erklärt werden kann, ohne dass dadurch ein irreparabler Schaden entsteht.

`act_optimistic` Der Container beherbergt eine vorläufige Version, deren Gültigkeit noch nicht

[9] Dies ist ähnlich zu nicht-determinanten Zustands-Abhängigkeiten aus Abschnitt 3.2, bezieht sich jedoch nicht auf die Zustände selbst, sondern auf die in den Zuständen gespeicherten kausalen Abhängigkeiten.

feststeht. Eine Ungültigkeits-Erklärung kann jedoch nur durch das Eigentümer-Mandat erfolgen (vgl. optimistische Strategien in der Datenbank-Literatur [VGH93]).

`act_tmp` Es handelt sich um einen vorläufigen (noch nicht abgeschlossenen) Container, zu dem jederzeit neue Operationen zur Beherbergung hinzukommen können. Ein Entzug bereits beherbergter Operationen ist im Normalfall nicht vorgesehen (einzige Ausnahme: Systemfehler oder andere katastrophale Ereignisse).

`act_close` Der Container ist geschlossen, d.h. es können keine neuen Operationen mehr beherbergt werden. Die Operationen bzw. deren Effekte brauchen nicht unbedingt persistent gespeichert zu sein. Ein erneutes Öffnen, d.h. der Wechsel in einen kleineren Zustand mittels `lock` ist dann (und nur dann) zulässig, wenn nirgendwo kausale Abhängigkeiten von diesem Container bestehen.

`act_freeze` Wie `act_close`, jedoch ist ein Wechsel in kleinere Zustände nicht mehr möglich.

`act_safe` Wie vorher, der Container samt beherbergtem Container-Inhalt ist jedoch mindestens 1 Mal persistent gespeichert und daher auch nach einfachen Systemfehlern wie Stromausfall noch erreichbar.

`act_multisafe` Der Container(-Inhalt) ist mehrfach redundant gegen schwerere Systemfehler gesichert.

Für die Zustände gelten folgende Regeln: Anlegen eines Containers ist nur mit den Aktualitätsgraden `act_destroyable` bis `act_tmp` möglich. Eine `unlock`-Operation setzt den Zustand aller von ihrem Adressbereich überstrichenen und im Mandats-Eigentum befindlichen Container auf einen wählbaren anderen Aktualitätsgrad; Zustands-Änderungen geschehen durch `unlock` mit Ausnahme von `act_invalid` nur in aufsteigender Richtung der Aktualitätsgrade. Der Zustand `act_invalid` ist von `act_destroyable` bis `act_close` aus erreichbar (also insbesondere ab `act_freeze` nicht mehr erreichbar); ein einmal erreichter `act_invalid`-Zustand kann nie mehr verlassen werden.

Weiterhin sollte beachtet werden, dass Zustandswechsel in höhere Aktualitätsgrade nur dann sinnvoll sind, wenn alle kausal vorangehenden Container mindestens den gleichen Aktualitätsgrad erreicht haben. Falls ein kausal vorangehender Container auf `act_invalid` gesetzt wird, müssen alle transitiv kausal abhängigen Container ebenfalls auf `act_invalid` gesetzt werden. Dies entspricht den *kaskadierenden Rollbacks* in klassischen Transaktions-Modellen.

Die `unlock`-Operation kann damit sowohl die klassische Funktionalität von Rollback-Operationen von Transaktionen als auch von Commit-Operationen ausführen. Ein Rollback entspricht dem Zustand `act_invalid`, beim Commit wird je nach geforderter Fehlertoleranz ein Aktualitätsgrad ab `act_freeze` angesteuert.

8.7. Schnittstellen von Lock-Operationen

Dieser Abschnitt stellt einen Versuch eines Entwurfes einer universell generischen Nest-Schnittstelle zur Behandlung möglichst vieler verschiedener Transaktions-Semantiken und Konsistenz-Modelle dar. Der hier vorgestellte Vorschlag soll lediglich als Diskussiongrundlage dienen, auf deren Basis weiter verallgemeinerte oder weiter spezialisierte bzw. revidierte Schnittstellen-Varianten entworfen werden können. Insbesondere Fragen nach der Praktikabilität sehr allgemeiner und universeller Transaktions-Semantiken sind teilweise auch in der Literatur noch nicht ausreichend geklärt oder werden in einigen Fällen zumindest für bestimmte Klassen von Anwendungen verneint; in der Praxis wird nicht ohne Grund nur ein geringer Teil der der hier abgedeckten Semantiken eingesetzt. Daher ist damit zu rechnen, dass einige konkrete Implementierung der hier *ermöglichten* Semantiken nicht praktikabel oder nur für bestimmte Nischen-Anwendungen praktikabel sind. Ziel dieses Vorschlags ist, eine möglichst hohe Bandbreite an verschiedenen Untermodellen und Implementierungen durch eine uniforme Schnittstelle zu unterstützen und zu weiteren Experimenten und Forschungen anzuregen.

8.7.1. Suchintervalle

Eine Nest-Instanz lässt sich im `multiversion`-Modell als zweidimensionaler Raum auffassen, der eine örtliche und eine zeitliche Dimension hat. Prinzipiell kann man jederzeit auf alle Teile dieses Raumes zugreifen. Dazu gibt es Suchoperationen, mit denen sich Lock-Container im Nest aufspüren lassen. Zur Spezifikation von Suchintervallen schlage ich folgende Konventionen vor:

Ein *Suchintervall* ist ein Paar `(start,delta)`, wobei `start` den Beginn und `start + delta` das Ende des Such-Bereiches angeben. Ein Suchintervall kann sich entweder auf die zeitliche oder auf die örtliche Dimension beziehen. Bei positivem `delta` wird in aufsteigender Richtung gesucht, d.h. falls sich mehrere Lock-Container mit dem Suchintervall in der jeweiligen Dimension schneiden, wird nach dem Durchführen von Vergleichsoperationen und eventuell davon ausgelöster Verkleinerung (nur bei der zeitlichen Dimension) diejenige Lock-Instanz ausgeliefert, die den kleinsten Wert in der jeweiligen Dimension hat und sich mit dem Suchintervall überschneidet. Bei negativem `delta` wird in umgekehrter Richtung gesucht.

Bei gleichzeitiger Suche nach Suchintervallen `adr_search` und `time_search` werden diejenigen Lock-Instanzen ausgeliefert, die sich in beiden Dimensionen mit den jeweiligen Suchintervallen überschneiden. Auf diese Weise ist eine Suche in einem rechteckigen Bereich des Ort-Zeit-Raumes möglich.

Als Sonderfall kann weiterhin `delta`$= +\infty$ oder `delta`$= -\infty$ zur unbeschränkten Suche zugelassen werden, ebenso `start`$= +\infty$.

8.7.2. Lock-Operationen im Multiversion-Modell

Die `lock`-Operation aus Abschnitt 3.3.5 wird im hier vorgestellten Erweiterungs-Vorschlag auf `multiversion`-

Modelle aus methodischen Gründen in die beiden Operations-Varianten `read_lock` und `write_lock` aufgesplittet, da sie eine leicht unterschiedliche Parameter-Versorgung benötigen. Es sind jedoch auch geringfügig andere Gestaltungen der Schnittstelle möglich, die die Uniformität der Schnittstelle beider Lock-Arten beibehalten.

```
read_lock(nest,mandate,
       speculative_req,adr_req,time_req,
act,min_act,stable,direct,aim,action) →
                  (adr_result,time_result)
```

Die Parameter `speculative_req` und `adr_req` übernehmen die Funktion von `try_address`, `try_len`, `log_address` und `len` aus Abschnitt 3.3.5; der Unterschied besteht in der Verwendung eines geeigneten Intervall-Typs. Zusätzlich wird ein Vorschlag für das Zeitstempel-Intervall `time_req` übergeben. In den Ergebnis-Intervallen `adr_result` und `time_result` werden die tatsächlich möglichen Orts- und Zeit-Intervalle zurück geliefert (sofern das Setzen des Locks möglich ist). Das Orts-Intervall muss auf jeden Fall mindestens `adr_req` umfassen, das Zeit-Intervall kann gegenüber `time_req` auch verkleinert ausfallen.

Der Parameter `act` gibt den Aktualitätsgrad des zu erzeugenden Read-Locks vor.

Der Parameter `min_act` spezifiziert hingegen einen Aktualitätsgrad, den sämtliche *direkt* kausal vorangehenden Write-Locks *mindestens* erfüllen müssen. Falls sich in der direkten kausalen Abhängigkeit Write-Locks mit zu geringem Aktualitätsgrad befinden, dann wird im Regelfall so lange *gewartet*, bis diese den geforderten Aktualitätsgrad erreicht haben. Damit lassen sich verschiedene Wartesemantiken emulieren:

1. Wählt man `min_act = act_close`, dann ergibt sich die Wartesemantik klassischer Locks, da die Aufwertung der konkurrierenden Lock mindestens bis zum Aktualitätsgrad `act_close` (oder höher) der Freigabe klassischer Locks entspricht.

2. Wählt man `act_destroyable` oder `act_optimistic`, dann wird im Extremfall überhaupt nicht auf konkurrierende Locks im klassischen Sinne gewartet[10]. Damit lassen sich optimistische Strategien von Transaktions-Implementierungen emulieren, wo die Korrektheit von gelesenen Versionen und kausalen Abhängigkeiten erst ganz zum Schluss bei der versuchten Aufwertung des aktuellen Containers nach `act_close` oder höher überprüft werden und ggf. eine Zwangs-Umwidmung nach `act_invalid` vorgenommen wird.

Je kleine Aktualitätsgrade man für `min_act` vorgibt, desto mehr wird die mögliche Parallelität gesteigert (vgl. „dirty read“ in Datenbanken), gleichzeitig steigt jedoch das Risiko kaskadierender Rollbacks und auch das Risko von inkonsistentem Lesen durch neu hinzukommende Beherbergungen in kausal vorgehenden Containern. Bei strengen Anforderungen an einzuhaltende Konsistenz-Bedingungen kann damit u.U. die Notwendigkeit von Rollbacks impliziert werden (vgl. *Fehlersicherheit von Schedules* [VGH93]).

Die kausalen Abhängigkeiten werden im allgemeinsten `multiversion`-Modell persistent aufbewahrt, um die Speicher-Semantik auch bei später nachgelieferten beliebig „vordatierten“ Locks erfüllen zu können. Der boolsche Parameter `stable` gibt vor, ob die kausalen Abhängigkeiten „stabil“ sein sollen, oder ob sich spätere Locks „dazwischen schieben“ dürfen. Bei `stable = true` dürfen später im Zeit-Intervall zwischen dem neu gesetzten Lock und dem kausal abhängigen Vorgänger keine solchen Write-Locks mehr nachträglich eingebaut werden, die zu *irgend einer* Veränderung des Suchergebnisses bei der (erneuten) Bestimmung der funktionalen Abhängigkeiten aller möglicherweise betroffenen Lock-Instanzen führen *können*[11]. Dadurch bestimmt der *Eigentümer* eines Locks (d.h. das Mandat oder eine Menge von Mandaten), ob andere Locks mit ihm verträglich sind oder nicht. Wenn man diese Eigenschaft auch transitiv für alle Vorgänger der kausalen Abhängigkeiten ermöglicht (z.B. durch einen Aufzählungstyp mit `stable = transitive`; die Bestimmung gilt unabhängig vom ursprünglichen `stable`-Zustand der transitiv vorangehenden kausalen Abhängigkeiten), dann lassen sich verschiedene Transaktions- oder Konsistenz-Modelle beliebig miteinander mischen, ohne die jeweils intendierte Semantik zu verletzen.

Der boolsche Parameter `direct` bestimmt, ob die erstellte kausale Abhängigkeit unbedingt vom *ersten* tatsächlich vorhandenen Write-Lock in Suchrichtung abhängen muss (dies ist in klassischen ACID-Transaktionen unbedingt erforderlich), oder ob (z.B. in verteilten Systemen) *veraltete* Versionen geduldet werden. Letztere Möglichkeit verbessert die Performanz zu Lasten weniger strikter Konsistenz-Modelle und kann zu „überspringenden“ kausalen Abhängigkeiten ähnlich der PRAM-Konsistenz[12] führen.

Der Parameter `aim` dient als Ersatz für den früheren Parameter `kind` aus Abschnitt 3.3.5. Er gibt einen der Werte `aim_address`, `aim_data` oder `aim_both` an. Damit wird spezifiziert, ob nur die Adress-Abbildungs-Funktion des Nestes, oder die Daten-Abbildungs-Funktion, oder beide gegenüber konkurrierenden Zugriffen gesperrt werden sollen (weiterhin kann damit auch die Zulässigkeit der beherbergten Operations-Arten geprüft werden, was hier nicht näher detailliert wird). Adressmodifizierende Operationen wie `move` lassen sich dadurch von datenmodifizierenden Operationen wie `transfer` entkoppeln. Um diese Fähigkeit besser auszunutzen, kann man beispielsweise einen weitere Parameter `adr_time` einführen, der den zu verwendenden Zeitstempel für die Adressabbildung unabhän-

[10] Streng genommen ist der Begriff „Lock“ in diesem Fall kein angemessener Begriff, da u.U. kein Warten mehr stattfindet. Ein mit min_act = act_destroyable gesetzter Read- oder Write-„Lock“ entspricht jedem einem Mitglied eines klassischen Read- oder Write-Sets, das die von einer Transaktion berührten Datenelemente repräsentiert. Ein allgemein anerkannter Überbegriff, der sowohl klassische Locks als auch für Read- oder Write-Sets umfasst, ist noch nicht gebildet worden.

[11] Die Bestimmung aller möglichweise betroffenen Lock-Instanzen muss nicht bei jedem Setz-Versuch neu erfolgen, wenn das Prinzip des *dynamic programming* angewandt wird. Weiterhin darf eine Implementierung das spätere Zwischenschieben von Locks auch dann zurückweisen, wenn dies laut `stable`-Angaben möglich wäre. Dadurch lassen sich Heuristiken einführen, die z.B. relativ grosse Bereiche der Raum-Zeit-Ebene gegen das Setzen neuer Write-Locks sperren.

[12] Weitere Konsistenzmodelle, insbesondere kausal halbgeordnete (z.B. [BSS91]), lassen sich eventuell im `direct`-Parameter durch Ersatz des Typs `boolean` durch einen Aufzählungtyp spezifizieren, der Zwischenstufen zwischen Ordnungslosigkeit und Totalordnung kausaler Abhängigkeiten ausdrücken soll.

gig von der Datenabbildung vorgibt. Damit können insbesondere solche Mandate einen ungestörten Zugriff auf die Nest-Adressen durchführen, die selbst keine adressmodifizierenden Operationen ausführen; diesen Mandaten erscheint dann die Adress-Abbildung als „fest genagelt“, obwohl sie von anderen Mandaten auf der virtuellen Zeitachse geändert worden sein kann.

Für die in den Containern beherbergten Zugriffsoperationen muss der passende Lock-Container in der zeitlichen Dimension für ein bestimmtes Mandat und eine bestimmte örtliche Adresse stets eindeutig bestimmt sein. Um dies zu erreichen, müssen wir verhindern, dass unter dem gleichen Mandat und unter der gleichen Adresse mehrere auf Dauer nicht abgeschlossene Lock-Container in der virtuellen Zeitachse gleichzeitig angelegt werden. Hierzu gibt es folgende Möglichkeiten:

1. Man weist jegliche Versuche zurück, mehrere zeitlich verschiedene Container für das gleiche Mandat und die gleiche Adresse anzulegen.

2. Man schliesst automatisch den vorher angelegten Container auf mindestens den Aktualiätsgrad `act_close` ab.

3. Man setzt automatisch den Aktualitätsgrad des früheren Containers auf `act_invalid`; hierbei kann es jedoch auch zu einer transitiven Invalidierung des gerade angeforderten neuen Containers bzw. zu einem Rollback des anfordernden Mandats kommen.

4. Man belässt den früheren Container in seinem aktuellen Zustand und richtet neue Zugriffsoperationen nur noch an den neuen Container; beim `unlock` werden hingegen sämtliche zeitlich verschiedenen aber örtlich überlappenden Lock-Container unter dem gleichen Mandat abgeschlossen.

Welche dieser Möglichkeiten sinnvoll ist, kann vom Anwendungsfall abhängen. Die letzte Möglichkeit dürfte jedoch in fast allen Fällen am sinnvollsten sein, da sie die geringsten Folge-Effekte hervorruft. Durch Einführen eines weiteren Parameters könnte man auch eine dynamische Auswahl unter allen 4 Möglichkeiten zulassen.

```
write_lock(nest,mandate,
       speculative_req,adr_req,time_req,
                act,causal,aim,action) →
                (adr_result,time_result)
```

Gegenüber `read_lock` kommt hier die direkte Angabe einer Menge (bzw. Liste oder Array) von Quadrupeln `causal` hinzu. Ein Quadrupel hat die Form `(adr_search, min_act, stable, direct)`. Der Wert von `adr_search` spezifiziert den Ort, an dem die kausal vorgehenden Read- oder Write-Locks liegen. Durch die Aufnahme von `min_act`, `stable` und `direct` in die Quadrupel lassen sich diese Eigenschaften für die funktionalen Abhängigkeiten in jedem Adressbereich einzeln einstellen[13].

Falls im Suchbereich bereits ein Lock-Container mit Aktualitätsgrad `act_close` existiert, dessen bereits vorhandenen kausalen Abhängigkeiten eine kompatible Obermenge zu den geforderten darstellen und der selbst nicht an späteren kausalen Abhängigkeiten teilnimmt, dann *darf* dieser Container an Stelle einer Neuerzeugung eines Containers erneut geöffnet werden. Diese Optimierung braucht nicht in jeder Implementierung unterstützt zu werden; `act_close` darf beim unlock auch als `act_freeze` behandelt werden.

Das *Aufwerten* von vorher gesetzten Read-Locks in Write-Locks findet dann statt, wenn sich in der Orts-Zeit-Fläche des neuen Write-Containers ein Read-Container mit gleichem oder geringerem Aktualitätsgrad und gleichem Eigentümer-Mandat befindet. Hierbei werden nachträgliche kausale Abhängigkeiten hinzugefügt (deren Verträglichkeit mit den speziellen implementierten Konsistenzbedingungen zu prüfen ist). Nach der Aufwertung wird der bisherige Read-Container so behandelt, als wäre er schon immer ein Write-Container gewesen.

```
unlock(nest,mandate,adr_search,act,action)
```

Gegenüber der früheren Version aus Abschnitt 3.3.5 kommt hier der Aktualitätsgrad `act` hinzu, auf den die freizugebenden Locks gesetzt werden sollen. Es ist möglich, Locks mehrmals schrittweise auf immer höhere Aktualitätsgrade zu setzen, so dass eine genaue Klammerung von `lock` / `unlock`-Paaren nicht unbedingt notwendig ist. Falls ein Lock-Container den mindestens geforderten Aktualitätsgrad bereits erreicht oder überschritten hat, geschieht nichts. Weiterhin kann durch `action = try` ein parallel laufender aynchroner Abschluss der betroffenen Container angestoßen werden, so dass weitere Aktivitäten wie das sofortige Starten neuer optimistischer Transaktionen ohne Verzögerung durchführbar sind (was jedoch auch die Wahrscheinlichkeit kaskadierender Rollbacks erhöhen kann).

```
notify_lock(nest,mandate,adr_try,adr_need,
       time_need,aim,urgency) → success
```

Fordert einen (asynchrone) Abschluss von bisher nicht-abgeschlossenen Lock-Containern an. Neben der Rückgabe spekulativ vergrößerter Adressbereiche lässt sich über `time_need` auch die Verkleinerung von Zeit-Intervallen in verteilten Systemen konsistent durchführen.

8.8. Emulation einiger bekannter Transaktions-Strategien

Verschiedene Varianten klassischer ACID-Transaktionen werden durch die soeben vorgestellten Lock-Operationen auf folgende Weise in einem `adaptor`-Baustein emuliert, der das `multiversion`-Modell in mehrere Instanzen von `singleuser`-Nestern transformiert[14], die jeweils ei-

[13] Falls diese indivuelle Einstellmöglichkeit nicht benötigt wird, kann man sie auch in die Parameter von `lock` verschieben und dadurch die Schnittstelle vereinfachen. Welche Semantik sich durch individuelle Einstellungen ergeben soll und ob diese im Sinne eines bestimmten semantischen Modells überhaupt gültig ist oder zurück gewiesen werden muss, kann auch (teilweise) in der konkreten Bausteinn Implementierung der Bearbeiter-Instanz festgelegt werden. Ansonsten hat der Aufrufer die Auswahl aus einer ungeheueren Vielzahl von Semantiken, die er durch seine Parameter-Wahl vorgeben kann. Als sehr einfaches Beispiel lässt sich durch eine leere Menge von Qudrupeln spezifizieren, dass keine funktionalen Abhängigkeiten bestehen; dies kann bei der Initialisierung oder beim Löschen / unbedingten Überschreiben des Dateninhaltes durchaus vorkommen und zusammen mit der Rückdatierungs-Möglichkeit auf der virtuellen Zeitachse die Parallelität dieses evtl. lange dauernden Vorgangs mit anderen Aktivitäten steigern.

[14] Dieser `adaptor`-Baustein fungiert als eine Art „Zwischenschicht“, die die klassische `singleuser`-Isolation aus einem generischen

ner klassischen Transaktions-ID mit den bekannten ACID-Eigenschaften entsprechen:

Zu jeder einzelnen Schreib- und Lese-Operation, die sich auf einen bisher von der jeweiligen `singleuser`-Sicht noch nicht angefassten Adressbereich bezieht, werden Locks zusätzlich generiert und vor dem Durchreichen an das `multiversion`-Modell vorangestellt. Read-Locks werden bei Bedarf dadurch in Write-Locks aufgewertet, dass auf dem gleichen Adressbereich die andere Lock-Art, jedoch mit dem gleichen Zeit-Intervall angefordert wird. Ansonsten werden einmal angeforderte Locks bis zum Transaktions-Ende nicht mehr freigegeben (2-Phasen-Locking).

Die kausalen Abhängigkeiten werden nach der Grundidee der Herbrand-Semantik syntaktischer Transaktions-Modelle [VGH93] stets so gesetzt, dass *sämtliche* von einer Transaktion jemals angefassten Adressbereiche bei den Write-Locks angegeben werden müssen.

Beim Commit werden alle gesetzen Locks atomar durch `unlock` auf dem gesamten Adressraum (d.h. man wählt `adr_search = (0,∞)`) mit mindestens `act = act_safe` abgeschlossen (sofern die D-Eigenschaft von ACID ausdrücklich bei jedem *einzelnen* Commit gefordert wird); beim Rollback ist statt dessen `act = act_invalid` zu wählen.

Bei den folgenden Emulationen konventioneller Transaktionen wird stets `stable = true` und `direct = true` gewählt (da man ansonsten nicht die korrekte Semantik der klassischen vollen Isolation sicherstellen kann). Die einzelnen Transaktions-Varianten unterscheiden sich dadurch, welche Zeit-Intervalle und `min_act`-Werte jeweils beim Setzen der Locks gewählt werden.

8.8.1. Timestamp-Ordering-Protokolle

Bei der Simulation von Multiversion-Timestamp-Ordering (MVTO) aus der Datenbank-Literatur [VGH93] wird die Serialisierbarkeit dadurch sicher gestellt, dass alle Operationen einer Transaktion als mit dem gleichen Zeitstempel durchgeführt *gelten*; dadurch entsteht eine virtuell zeitliche Totalordnung aller Operationen, die genau der Serialisierungs-Reihenfolge entspricht. Datenbanken benötigen hierzu ebenfalls die Fähigkeit zur Verwaltung mehrerer Versionen von Objekten (manchmal auch Time-Domain-Addressing genannt [GR93]). Der Zeitstempel wird bei Beginn der Transaktion oder beim ersten Zugriff einmal so bestimmt, dass jede Transaktion einen eindeutigen Zeitstempel besitzt, der nie mehr geändert wird.

Die Simulation von Schreib-Lese-Transaktionen durch `multiversion`-Nester ist relativ einfach: die Locks werden mit `time_req = (start,0)` und `act = act_tmp` gesetzt, wobei `start` den einmal bei Beginn der Transaktion eindeutig vergebenen Zeitstempel bedeutet.

Nur-Lese-Transaktionen lassen sich dadurch simulieren, dass bei Beginn der Transaktion abweichend von der oben beschriebenen allgemeinen Strategie ein einziger Read-Lock auf dem gesamten Adressbereich `adr_req = (0,∞)` gesetzt wird, so dass die anschließenden Lese-Operationen nicht mehr einzeln durch Read-Locks abgesichert werden müssen. Ein derartiger Read-Lock, der den gesamten Adressbereich umfasst, erzeugt einen *Schnappschuss*[15] des gesamten Nestes.

Da auch bei Schreib-Lese-Transaktionen sämtliche Read-Locks mit dem gleichen Zeitstempel gesetzt werden, lassen sich auch hier die individuellen Read-Locks durch einen Schnappschuss-Lock ersetzen; beim Schreiben werden dann einzelne Teile des Schnappschuss-Locks durchlöchert und zu Write-Locks aufgewertet.

Die *Korrektheit* bestimmt sich danach, ob das Setzen von Locks möglich ist, ohne die kausalen Abhängigkeiten fremder Mandate (mit `stable=true`) zu verletzen.

8.8.2. Striktes 2-Phasen-Locking

Bei der Simulation klassischer 2-Phasen-Locks [VGH93] wird das `multiversion`-Verhalten gar nicht ausgenutzt. Read- und Write-Locks werden auf dem Zeitintervall `time_req = (now,∞)` gesetzt, wobei `now` der streng monoton fortschreitenden aktuellen realen Uhrzeit beim Aufruf entspricht. Beim Gewähren des Locks wird dieses Zeitintervall auf den Gewährungs-Zeitpunkt verkleinert, so dass nachfolgende Write-Locks stets auf der Zeitachse Platz finden können. In den Quadrupeln der Write-Locks wird durch `min_act = act_freeze` sicher gestellt, dass zu jedem Zeitpunkt höchstens ein einziger nicht abgeschlossener Write-Lock auf überlappenden Adressen existiert (klassische Wartesemantik von Lock-Operationen).

8.9. Weitere Ideen

8.9.1. Weitere Arten von Serialisierbarkeit bzw. Konsistenzmodellen

In einem noch allgemeinenere Modell könnte man auch den in einem Lock-Container beherbergten Zugriffsoperationen eine zeitliche Dimension zuordnen. Somit dürfen unter dem gleichen Mandat auch mehrere örtlich überlappende nicht abgeschlossene Write-Locks zum gleichen Real-Zeitpunkt vorhanden sein, wenn sie sich nicht auf der virtuellen Zeitachse überschneiden und keine Verletzungen der kausalen Abhängigkeiten auftreten. Man könnte dieses Modell als Multiversions-Locking bezeichnen. Die Möglichkeit, örtlich aber nicht zeitlich überlappende Adressbereiche auch zum gleichen Real-Zeitpunkt sperren zu können, geht über den Begriff „multiversion concurrency control[16]“ [HR01] bzw. den Begriff der Multiversions-Konflikt-Serialisierbarkeit [VGH93] hinaus. Eventuell wäre Time-Domain-Locking eine passende Bezeichnung für den englischen Sprachraum.

Der verallgemeinerte Serialisierungs-Begriff aus [Vid87], der sowohl die Konflikt- als auch die MVTO-Serialisierbarkeit als Spezialfall enthält, dürfte sich durch die hier vorgestellte Nest-Schnittstelle ebenfalls simulieren

`multiversion`-Nest erzeugt.

[15] Im Unterschied zu [CL85] lässt sich ein Schnappschuss auch *rückwirkend* in der virtuellen Zeitachse machen.

[16] Bisher wurde Versionierung in Datenbanken offenbar nur zur Entkoppelung zwischen Lesern und Schreibern eingesetzt. Das hier vorgestellte Modell ermöglicht dagegen prinzipiell auch die Entkoppelung zwischen parallel aktiven Schreibern; Einschränkungen ergeben sich lediglich aus den kausalen Abhängigkeiten.

lassen. Dies gilt ebenso für Protokoll-Familien wie MVS-GT (Multiversions-Serialisationsgraphen-Tester), die sich als konkrete Strategien in Bausteinen realisieren lassen.

Geschachtelte Transaktionen lassen sich durch Bausteine realisieren, die Gruppen von Änderungs-Operationen nach aussen hin durch gesammelte Lock-Operationen weiter geben und dabei ggf. die Aktualitätsgrade in andere Werte transformieren.

Durch „überspringende“ kausale Abhängigkeiten, Aufweichen der Herbrand-Semantik und Variation der `min_act-` , `stable-` und `direct`-Parameter dürften sich weitere bisher unbekannte Serialisierbarkeits-Begriffe bilden lassen, deren systematische Untersuchung zukünftigen Arbeiten überlassen bleiben soll. Ein großes Forschungs-Feld stellen Optimierungen bei sich widersprechenden Zielen dar, beispielsweise wenn spekulativ vergrößerte Lock-Bereiche nur mit „überspringenden“ kausalen Abhängigkeiten oder geringerem Aktualitätsgrad erkauft werden können[17] u.v.m.

Durch gezieltes Freigeben einiger Locks vor dem Transaktions-Ende[18] lassen sich beliebige Misch-Formen zwischen vollkommener transaktionaler Isolation und dem wechselseitigen Ausschluss realisieren[19]. Der bisher in Betriebssystemen verwendete wechselseitige Ausschluss wird durch das Konzept der Aktualitätsgrade so orthogonal ergänzt, dass ein nachträgliches Rücksetzen schief gegangener Berechnungen (die mit Berechnungen anderer Mandate nicht-serialisierbar verwoben sein können) möglich ist.

8.9.2. Automatische Re-Evaluation

Die Idee besteht darin, in den Containern (gelegentlich oder teilweise auch in den gespeicherten kausalen Abhängigkeiten) nicht (nur) die Ergebnisse der von den Konsumenten generierten Zugriffs-Operationen zu beherbergen, sondern (zusätzlich) ihre *Berechnungsvorschrift*.

Dadurch kann man die beherbergten Zugriffs-Operationen neu berechnen, wenn nachträgliche Änderungen bei einem der kausal vorgehenden Container oder nachträgliches Einschieben von Write-Locks in stabile kausale Abhängigkeits-Beziehungen zugelassen werden, wodurch die kausalen Abhängigkeiten auf die neuen Versionen abgeändert werden. Bei solchen Änderungen wird lediglich eine Neuberechnung des Container-*Inhaltes* aller transitiv kausal abhängigen Container angestossen; die Container selbst und ihre Position in der Raum-Zeit-Ebene bleiben erhalten.

Eine Spezifikation der Berechnungsvorschrift kann durch Einführen eines geeigneten *Interpreters* und einer *Interpreter-Sprache* erfolgen, von der Programmstücke mittels einer Operation `download` bekannt gemacht werden (z.B. zur Addition von Zahlen-Werten oder Berechnung einer Summe). Das Herunterladen von ausführbarem Maschinencode ist prinzipiell ebenfalls möglich, hat jedoch Nachteile bei der Sicherheit (zu deren Lösung kann man spezielle Konventionen für die während der Evaluation aktiven Schutzbereiche einführen).

In beiden Fällen sollten möglichst *alle* Konsumenten ihre `transfer`-Operationen zum Ändern der Daten- und/oder Adressabbildung von Nestern durch `download`-Operationen ersetzen, damit nicht-reevaluierbare Container dieses Verfahren nicht stoppen können oder inkorrekt werden lassen.

Weitere Untersuchungen dieser Idee sind zukünftigen Forschungen vorbehalten.

[17] Die klassischen Flaschenhals-Probleme verteilter Systeme, insbesondere die durch Latenzen verursachten, lassen sich ebenfalls unter dem Aspekt sich widersprechener Optimierungs-Ziele (Tradeoffs) betrachten. Interessant wäre eine Vereinheitlichung der Darstellung dieses Gebiets mit den hier aufgeworfenen Tradeoffs.

[18] Die klassische Zwei-Phasen-Eigenschaft der Gesamtfreigabe am Ende einer Transaktion ist *allen* Transaktions-Varianten gemeinsam, die sich durch das hier vorgestellte `multiversion`-Modell simulieren lassen.

[19] Eine griffige Formel zur Charakterisierung beider Extreme könnte auf dem Hintergrund der Betrachtung von Locks als Container auf der virtuellen Zeitache folgendermassen lauten: Bei transaktionaler Semantik werden Container so lange wie möglich offen gehalten (um sie ggf. für invalide erklären zu können), beim wechselseitigen Ausschluss werden Container dagegen tendenziell so früh wie möglich geschlossen, um parallelen Zugreifern „eine Chance zu geben“.

9. Schlussbemerkungen

In dieser Arbeit wurde dargelegt, wie sich ein vollwertiges Betriebssystem prinzipiell durch ein Baukasten-System mit nur wenigen universellen Grund-Abstraktionen aufbauen lässt.

Auf dem Papier bzw. im Kopf entwickelte Konzepte bergen immer die Gefahr von Unstimmigkeiten und übersehenen Problemen, die ohne Rückmeldung aus der Praxis schwer zu finden und zu korrigieren sind. Die Implementierung der hier vorgestellten Architektur durch das Athomux-Projekt [Ath] hat zwar bereits viele Rückmeldungen und Bestätigungen ergeben, dürfte aber bei weitem nicht ausreichen. Es sind immer noch sehr viele Fragen offen, zu deren Beantwortung die Kräfte einer größeren Gemeinschaft notwendig sein werden.

In dieser Arbeit konnte nur ein Aufriss einiger wesentlicher Probleme einer LEGO-artigen Nest-Baustein-Architektur gegeben werden. Viele Detailprobleme harren noch ihrer Lösung. Einige Beispiele für noch zu untersuchende Aufgabenstellungen:

- Welche weiteren Baustein-Arten sind notwendig oder sinnvoll?
- Gibt es eine „bessere“ Zerlegung der zu implementierenden Funktionalität in Bausteine als hier vorgeschlagen?
- Welche Instanz-Konfigurationen sind bei welchen Anwendungs-Szenarien vorteilhaft?
- Wie verhält sich die erzielbare Performanz eines Baukastens im Vergleich zu konventionellen „festverdrahteten“ Betriebssystem-Architekturen?
- Gibt es eine andere, vorteilhafte Zerlegung der Nest-Funktionalität in Elementaroperationen?
- Welche Alternativen gibt es für Namen für Baustein-Instanzen und Drähte? Wie kann ein Verdrahtungs-Netzwerk auf bessere Weise extern repräsentiert bzw spezifiziert werden, wie könnten Instantiierungs- bzw Konstruktor-Operationen besser realisiert werden?
- Sollte man Konstruktor/Destruktor- bzw Instantiierungs/De-Intantiierungs-Operationen besser in eine eigene Grund-Abstraktion packen, oder wie in der Athomux-Implementierung zu den Nest-Elementaroperationen hinzu nehmen, oder wie hier ursprünglich vorgeschlagen ausschließlich als generische Operationen realisieren?
- Durch welche Arten von `strategy_*`-Bausteinen sollte man die (automatische / virtuelle) Konstruktor-Funktionalität am besten realisieren?
- Sind die hier vorgeschlagenen Modelle `singleuser`, `multiuser` und `multiversion` ausreichend und angemessen im Verhältnis Aufwand zu Nutzen? Gibt es „bessere“ (Unter-)Modelle? Welche Modelle eignen sich für welche Anwendungen und Szenarien?
- Wie sind Sicherheit und Zugriffsschutz in einer dynamischen Nest-Baustein-Architektur am besten zu gewährleisten? Welche Sicherheits-Modelle passen eventuell besser als bisher betrachtete, die auf konventionellen festen Annahmen über Betriebssystem-Architekturen basierten?
- Welche Typsysteme für generische Operationen bieten den besten Kompromiss zwischen Aufwand und Nutzen in möglichst vielen Anwendungs-Szenarien?
- Wie sollte eine möglichst universelle Schnittstelle zwischen „Anwendungsprogrammen“ und einem Nest-Baustein-Betriebssystem aussehen? Was sollte publiziert, was verborgen werden? Wo sind Anpassungen an die Bedürfnisse der Anwender und der Anwendungs-Programmierer notwendig?
- Welche weiteren im Athomux-Projekt noch nicht untersuchten Alternativen gibt es, um konventionelle Betriebssystem-Schnittstellen (z.B. Posix oder die „Linux“-Fortentwicklung) mit geringem Aufwand und guter Performanz nachzubilden?
- Wie schafft man Interoperabilität zwischen verschiedenen „Personalities“ wie z.B. Posix versus native Schnittstelle?
- In welchen Fällen ist die Erschließung von bereits im Quelltext vorhandenen Gerätetreibern für Linux oder *BSD mittels Kapselungs-Bausteinen sinnvoll und ausreichend, ab wann ist die Erstellung eigener Treiber und Treiberhierarchien notwendig und sinnvoll? Wie verhindert man „Intrusivitäten“ durch „Fremdanleihen“? Wie hält man mit der zeitweise stürmischen Entwicklung von Fremd-Treibern und/oder der zugehörigen Hardware Schritt?
- Wie können Netzwerk-Protokolle am besten mittels einer Nest-Baustein-Architektur realisiert werden?
- Wie geht man sinnvoll mit Änderungen von Infrastruktur-Belangen (z.B. Nest-Schnittstelle) und Schnittstellen nach außen um? Wie verwaltet man *Versionen* von Bausteinen und/oder Schnittstellen am besten?
- Ist der hier vorgeschlagene Weg der Integration von Datenbanken und Betriebssystemen gangbar? Was ist dabei zu beachten? Welche weiteren Lösungen gibt es, welche sind zu bevorzugen?
- Welche Vor- und Nachteile bringen welche Untermodelle von `multiversion` in welchen Bausteinen und in welchen Anwendungs-Szenarien? Welche Probleme sind bei einer Durchdringung von

`multiversion` durch fast die gesamte Baustein-Infrastruktur zu lösen? Wie beherrscht man diese Probleme? Was bringt / kostet `multiversion` in verteilten Systemen?

- Wie kann man die besonderen Merkmale von Nestern, insbesondere die `move`-Operation, auch Anwendungsprogrammen in virtuellen Adressräumen verfügbar machen? Wie kann man dynamisch verschiebliche Arrays sinnvoll auf Programmiersprachen-Ebene ansprechen und verwalten?

- Welche Art von Hardware-Unterstützung ist zur Unterstützung von Verschiebeorperationen in virtuellen Benutzer-Adressräumen wünschenswert und mit gutem Verhältnis von Aufwand zu Nutzen zu implementieren? Wie kann man feinere Granularitäten als Speicherseiten mit geringem Aufwand unterstützen? Was sollte besser in Hardware, was besser in Software realisiert werden?

- Welche Auswirkungen auf Programmierstile, Sprachen und Laufzeitsysteme haben dynamisch verschiebliche Arrays (z.B. Ersatz von verketteten Listen durch dynamische Arrays)? Wie sollten Typkonzepte für solche Konstrukte aussehen?

- Wie sollten Typkonzepte für relative Adressierung mittels Offsets aussehen (z.B. Offset-Pointer-Typen)? Wie lassen sie sich nahtlos in konventionelle Programmiersprachen integrieren?

- Wie kann man auf einfache und effiziente Weise Pointer-Werte oder Offset-Pointer-Werte an Verschiebungen anpassen?

- Welche theoretischen Konsequenzen ergeben sich aus einem Speichermodell für Registermaschinen (RAM oder PRAM), wenn man uniforme Verschiebeoperationen einführt (beispielsweise geht Sortieren durch Einfügen mit $O(n \log n)$ statt $O(n^2)$)?

- Welche Vor- und Nachteile bringt die Instanz-Orientierung im Software-Engineering? Wie sollten passende Entwurfsmethoden aussehen?

A. Simulation von Vererbung durch Generizität

Zum Nachweis, dass sich Erweiterungs-Generizität mit Hilfe von Parametrischer Generizität ausdrücken lässt, simulieren wir die Kernidee der objektorientierten Vererbung durch Präprozessor-Makros der Sprache C:

```
// Allgemeine Deklarationen
#define class struct

// List iterators
#define decl_method(base,type,name) base##_##name##_head((*base##_##name),class type);
#define def_method(base,type,name) \
  static base##_##name##_head(base##_##name,class type) \
  base##_##name##_body(class type)
#define init_item(base,type,name) self->type##_##name = type##_##name;

// Hilfs-Makros
#define init_head(type) void type##_init(class type * self)
#define declarations(type) type##_attribs type##_list(decl_method,type,type)
#define inherit(base,type) base##_attribs base##_list(decl_method,base,type)
#define inherit_init(base,type) base##_init((class base*)self);

// Haupt-Makros
#define base_class(type)           \
class type;                        \
                                   \
type##_list(def_method,type,type)  \
                                   \
class type {                       \
  declarations(type)               \
};                                 \
                                   \
init_head(type)                    \
{                                  \
  type##_list(init_item,type,type) \
}

#define sub_class(base,type)       \
class type;                        \
                                   \
type##_list(def_method,type,type)  \
                                   \
class type {                       \
  inherit(base,type)               \
  declarations(type)               \
};                                 \
                                   \
init_head(type)                    \
{                                  \
  inherit_init(base,type)          \
  type##_list(init_item,base,type) \
}
```

Auf diese allgemeinen Deklarationen hin folgt nun die einmalige Angabe zweier Klassen A und B mittels Makro-Definitionen:

```
// spezifisch fuer A

#define A_attribs                \
  int member1;                   \
  int member2;

#define A_method1_head(name,type) \
void name(type * self)

#define A_method1_body(type)      \
{                                 \
   /*...*/                        \
```

```
}

#define A_method2_head(name,type) \
void name(type * self)

#define A_method2_body(type)      \
{                                 \
   /*...*/                        \
}

#define A_list(op,b,t)     op(b,t,method1) op(b,t,method2)

// Anwendung A

base_class(A)

// spezifisch fuer B

#define B_attribs                 \
  int member3;                    \
  int member4;

#define B_method3_head(name,type) \
void name(type * self)

#define B_method3_body(type)      \
{                                 \
   /*...*/                        \
}

#define B_list(op,b,t)     op(b,t,method3)

// Anwendung B

sub_class(A,B)
```

Mit Hilfe der Unix-Kommandos `gcc -E vererbung.c | grep -v '^#' | indent -bad -bap | indent -kr` entsteht hieraus der folgende expandierte Quelltext:

```
struct A;
static void A_method1(struct A *self)
{
}
static void A_method2(struct A *self)
{
}
struct A {
    int member1;
    int member2;
    void (*A_method1) (struct A * self);
    void (*A_method2) (struct A * self);
};
void A_init(struct A *self)
{
    self->A_method1 = A_method1;
    self->A_method2 = A_method2;
}
struct B;
static void B_method3(struct B *self)
{
}
struct B {
    int member1;
    int member2;
    void (*A_method1) (struct B * self);
    void (*A_method2) (struct B * self);
    int member3;
    int member4;
    void (*B_method3) (struct B * self);
};
void B_init(struct B *self)
{
    A_init((struct A *) self);
    self->B_method3 = B_method3;
}
```

B. Beispiel-Problem der Objektorientierung

Ein Beispiel, bei dem die Mitvererbung von Datenstrukturen bzw. Repräsentationen zu einer kombinatorischen Explosion der Anzahl von Klassen führen kann:

Die Erweiterung einer abstrakten Basisklasse G solle auf zwei verschiedene voneinander unabhängige Arten stattfinden. Dies lässt sich relativ einfach durch zwei verschiedene abgeleitete abstrakte Basisklassen A und B durchführen. Aus jeder der beiden abstrakten Basisklassen seien verschiedene konkrete Klassen $A_1, \ldots, A_n$ und $B_1, \ldots B_m$ abgeleitet, deren Funktionalität jeweils benötigt werde. Nun komme die Anforderung hinzu, dass die Vereinigung beider Schnittstellen ebenfalls benötigt werde. Auf der Ebene der abstrakten Basisklassen ist dies sehr einfach durch eine weitere Basisklasse C zu bewerkstelligen, die ihre Schnittstelle aus A und B durch Mehrfachvererbung erhält. Dies bedeutet für die davon abgeleiteten konkreten Klassen C_{ij} jedoch, dass nun das gesamte kartesische Produkt aus $n \cdot m$ abgeleiteten Klassen in Frage kommt und ggf. zu implementieren ist. Selbst wenn dabei keine neue Funktionalität mittels überschriebener Methoden-Implementierungen hinzu kommen sollte, führt dies zu einer kombinatorischen Explosion der Klassen-Namen und zu einer Aufblähung des gesamten Systems. Nach meinem Dafürhalten taucht dieses Problem oft in der Praxis auf, ohne erkannt zu werden.

Für dieses Problem gibt es eine oft funktionierende Lösung, ohne das objektorientierte Paradigma verlassen zu müssen. Die Implementierung einer abgeleiteten Klasse C_{ij} sollte nicht durch Mehrfachvererbung erfolgen, sondern durch Container-Instanzen $c_{ij} \in C_{ij}$, die jeweils zwei Instanzen $a_i \in A_i$ und $b_j \in B_j$ enthalten (in Form einer has-a-Beziehung anstatt einer is-a-Beziehung). Man kann C_{ij} auch als Proxy-Klasse bezeichnen. Dieses Vorgehen bringt jedoch folgende Schwierigkeiten mit sich:

1. Die bisherigen Attribute waren Mitglieder der alten Objektinstanz-Repräsentation; von der neuen Repräsentation werden sie entweder indirekt adressiert oder sie werden nochmals eingebettet. Dadurch wird Platz verschwendet.

2. Die gemeinsame Oberklasse G kann Attribute enthalten, die bei der Mehrfachvererbung nur einmal gemeinsam instantiiert werden sollen.

3. Der Namensraum wird nach wie vor durch $n \cdot m$ verschiedene Klassennamen verschmutzt.

Das erste Problem ist lästig, gefährdet aber die Korrektheit nicht. Zur Lösung des zweiten Problems kann man eine dritte Komponente $g \in G$ in die Container-Instanz einführen. Das dritte Problem ist bei dieser Lösung nicht zu umgehen und belastet im günstigsten Fall nur den Compiler.

Es gibt jedoch noch eine weitere Lösung: man führt eine Klasse C' ein, die im wesentlichen die Summe der Schnittstellen enthält, jedoch bei den Konstruktoren weitere Parameter i und j einführt. Ansonsten fungiert C' nur als trivialer Container, der die eigentliche Implementierung in G, A und B aufruft (was den Schnittstellen-Aufwand etwa verdoppelt). Damit betreibt man in der Objektorientierung jedoch letztlich die gleiche Art von Einbettung von Unterobjekt-Instanzen, wie sie in „konventionellen“ Programmierparadigmen schon immer gehandhabt wurde; es wird lediglich ein Zusatz-Aufwand mit Duplizierung von Schnittstellen getrieben, um das objektorientierte Paradigma mittels Mehrfachvererbung an der *Schnittstelle* retten zu können. Es stellt sich die Frage, ob die konzeptuell durch Mehrfachvererbung entstandene Schnittstelle C', deren Implementierung allerdings keine (Mehrfach-)Vererbung benutzt, wirklich benötigt wird, oder ob die durchgehend gleichzeitige Benutzung von A und B als Ersatz für C' nicht unter dem Strich einfacher ist. Unabhängig davon sehe ich das für die Objektorientierung wesentliche Paradigma der Vererbung nicht mehr als tatsächlich benutzt, und ich stelle die Frage, ob man es in diesem Beispiel nicht gleich von vornherein hätte weglassen können.

Dies bedeutet nach meiner Ansicht nicht, dass die automatisierte Ableitung von objektorientierten Datenschemata aus Klassen-Schemata generell nutzlos ist, denn es gibt sehr wohl Anwendungsfelder, wo die automatische Ableitung immer wiederkehrende Programmiervorgänge automatisiert.

Ich bin der Ansicht, dass die Objektorientierung ein oftmals nützliches Denkschema zum Entwurf von Schnittstellen ist; dass aber eine unreflektierte Umsetzung dieses Schemas in konkrete Datenstrukturen kostspielig oder sogar gefährlich werden kann, da die üblicherweise angebotenen Daten-Repräsentationen nicht für jede Aufgabenstellung optimal zu sein brauchen.

Auf methodischer Ebene scheint mir das geschilderte Problem auch darin zu liegen, dass objektorientierte Entwurfs-Denkweise oftmals auf Biegen und Brechen versucht, alles mittels Erweiterungs-Generizität zu lösen. Die hier vorgeführte Problemstellung passt jedoch viel besser zur kompositorischen Generizität.

Dieses Beispiel dient daher als weiteres Argument für den von mir propagierten Vorrang kompositorischer Generizität vor Erweiterungs-Generizität.

C. Ansätze zur Vermeidung von Wettrennen

(Ergänzungen zur `notify_*`-Problematik aus Kapitel 5)

1. Man sorgt dafür, dass stets genügend Ressourcen vorhanden sind und ein Entzug durch `notify_*` niemals notwendig wird. Leider sind in der Realität oft nicht genügend Ressourcen zur Deckung des Bedarfs vorhanden. Wenn der Bedarf limitiert ist oder wenn man bereits bei der Ausgabe von Ressourcen Limits einführt (z.B. Quota), kann man u.U. auf Rückforderungsmechanismen verzichten. Ebenfalls in diese Kategorie fällt z.B. der Tausch von gleichwertigen Ressourcen, bei dem ebenfalls eine Rückforderung in Zukunft ausgeschlossen werden kann. Im Sinne des Eigentum / Besitz-Modells (Abschnitt 5.1) bedeutet dies, dass echtes Eigentum (nicht nur bloßer Besitz) übertragen wird, das nie mehr zurückgefordert werden kann.

2. Man verlangt, dass Besitz grundsätzlich nur an hierarchisch tiefer stehende Instanzen und nur für die Dauer der Bearbeitung einer Operation weiter gegeben werden darf (physischer IO); damit würde das Konzept des logischen IO nicht mehr benutzt und das Modell eingeschränkt.

3. Man erlaubt nur die Rückvergabe von solchem Unterbesitz an höhere Instanzen, der bereits einmal von der gleichen höheren Instanz als Hauptbesitz an die niedrigere vergeben wurde (transitive Rücklieferung von Unterobjekten). Damit gehört der Unterbesitz in Wirklichkeit stets der höheren Instanz; eine Rückforderung durch den Hauptbesitzer wird als *unberechtigt* klassifiziert und nicht ausgeführt bzw. gar nicht erst versucht. Diese Konstruktion erweitert das Modell gegenüber 2., ohne gravierende Nachteile einzuführen; daher gebe ich ihm den Vorzug. Allerdings sehe ich es immer noch als relativ stark eingeschränkt an, insbesondere für verteilte Systeme und Netzwerk-Betriebssysteme. Auf der konzeptuellen Ebene scheint mir lediglich eine Vertauschung von Etiketten stattzufinden, da in Wirklichkeit keine echte Delegation von Macht über den Besitz stattfindet.

4. Man vergibt Besitz an höhere Instanzen (ggf. auch an niedrigere Instanzen) grundsätzlich nur auf begrenzte (Real-)Zeit, innerhalb deren die Ressource entweder zurückgegeben oder die Verleihzeit verlängert werden muss; die Zeitschranke wird so bemessen, dass ein Verfall des Leihdatums im Normalbetrieb nicht stattfindet. Nach diesem Modell ist es die Pflicht eines jedes Ausleihers, entweder für rechtzeitige Rückgabe oder für Fristverlängerung zu sorgen (analog zum Ausleih-Verfahren einer Universitäts-Bibliothek). Ein Überschreiten der Zeitschranke wird als *Fehlverhalten* interpretiert, und die Ressourcen können z.B. automatisch entzogen werden. Dieser Entzug hat jedoch wieder den Charakter eines asynchron auftretenden Signals, d.h. es ändert nichts am Prinzip der Rückgabe, nur am Auslöser. Dieses Modell eignet sich insbesondere für verteilte Systeme, da eine nicht rechtzeitig erfolgte Rückgabe bzw. Verlängerung auch als *Ausfall* einer Instanz interpretiert werden kann, auf den dann z.B. mit automatischen Entzug des Ressourcen-Besitzes und Neuverteilung *reagiert* wird; falls die Instanz nicht tatsächlich ausgefallen war, sondern nur die Kommunikationsverbindung, dann „weiß“ die betroffene Instanz nach Ablauf der Frist, dass sie die Verlängerung nicht mehr rechtzeitig geschafft hat und dass sie daher nicht mehr „rechtmäßiger“ Besitzer ist.

5. Die Besitzvergabe an höhere Instanzen geschieht ausschließlich unter der vollen Kontrolle der niedrigeren Instanz, die vollständig bestimmen kann, was damit geschieht. Damit wird jedoch das Hierarchie-Modell auch im regulären Betrieb auf den Kopf gestellt; es würde sich die Frage stellen, ob es noch eine Berechtigung hat. Auch die Eignung für verteilte Systeme scheint mir zweifelhaft.

Keiner dieser Ansätze scheint mir geeignet, das Wettrenn-Problem (Kapitel 5) so aus der Welt zu schaffen, dass dadurch keine neuen Probleme entstehen und dass Anforderungen von verteilten Systemen ausreichend berücksichtigt werden. Eine Bewertung, welche dieser Probleme als gewichtiger zu betrachten ist, kann je nach Standpunkt des Bewerters und nach Anforderungen an das System unterschiedlich ausfallen (beispielsweise sehe ich bei extrem sicherheitskritischen Anwendungen Vorteile bei Modell 3). Es ist also grundsätzlich möglich, eine konkrete Baustein-Hierarchie innerhalb der hier vorgestellten Baustein-Architektur so zu gestalten, dass `notify_*`-Operationen und deren Wettrennen vermieden werden; dies sehe ich dort als erstrebenswert an, wo eine Vermeidung ohne weiteres möglich ist und nur geringe Kosten verursacht. Ich sehe jedoch geringe Chancen, Wettrennen in einem verteilten System so zu vermeiden, dass keine Performanz-Nachteile oder Abhängigkeiten von der Funktionsfähigkeit zentraler Instanzen entstehen.

Danksagung

Das Zustandekommen dieser Arbeit verdanke ich Klaus Lagally. Ohne sein Vorbild in der Denk- und Anschauungsweise, und ohne die beinahe grenzenlose Freiheit, die er mir zur eigenständigen Forschung gelassen hat, hätte es diese Arbeit nicht gegeben. Er hat mich nicht mit anderen Aufgaben überschüttet oder mich zu irgend etwas gedrängt, obwohl es Phasen gab, in denen es so aussah, als ob bei meiner Arbeit nichts Vernünftiges herauskommen würde. Seine zutiefst von humanistischen Idealen durchtränkte Grundhaltung hat die Atmosphäre geschaffen, in der etwas gewachsen ist, was lange Zeit nur im Verborgenen lag und kaum von außen sichtbar war.

Ein besonderer Dank gebührt Johannes Elsner, der mir dabei geholfen hat, zu meinen Stärken zu finden.

Weiterer Dank an Stefan Staiger, der mir als glühender Verfechter der Objektorientierung geholfen hat, die hier vorgestellten Konzepte besser von denjenigen der Objektorientierung abzugrenzen und Unterschiede zu erkennen, die mir vorher nicht vollkommen bewusst waren.

Schließlich ein Dank an Florian Niebling, Jens-Christian Korth, Hardy Kahl und Roland Niese, die mich bei der Implementierung von Athomux unterstützt haben und zu Verbesserungen und Vereinfachungen der Schnittstelle beigetragen haben.

Literaturverzeichnis

[A+91] ANDERSEN, THOMAS E. und OTHERS: *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*. Symposium on Operating System Principles, Seiten 95–109, 1991.

[AB86] ARCHIBALD, JAMES und JEAN-LOUP BAER: *Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model*. Transactions on Computer Systems, 4(4):273–298, 1986.

[Ant90] ANTONOV, VADIM G.: *A Regular Architecture for Operating Systems*. Operating System Reviews, 24(3):22–39, 1990.

[Ass96] ASSENMACHER, HOLGER: *Ein Architekturkonzept zum Entwurf flexibler Betriebssysteme*. Dissertation Universität Kaiserslautern, 1996.

[Ath] *The Athomux operating system project*. `http://www.athomux.net`.

[AW94] ATTIYA, HAGIT und JENNIFER L. WELCH: *Sequential Consistency versus Linearizability*. Transactions on Computer Systems, 12(2):91–122, 1994.

[B+90] BERSHAD, BRIAN N. und OTHERS: *Lightweight Remote Procedure Call*. Transactions on Computer Systems, 8(1):37–5, 1990.

[B+95] BERSHAD, BRIAN N. und OTHERS: *SPIN – An Extensible Microkernel for Application-specific Operating System Services*. Operating System Reviews, 29(1):74–77, 1995.

[Bac86] BACH, MAURICE J.: *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[BHG87] BERNSTEIN, PILIP A., VASSOS HADZILACOS und NATHAN GOODMAN: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BN84] BIRRELL, ANDREW D. und BRUCE JAY NELSON: *Implementing Remote Procedure Calls*. Transactions on Computer Systems, 2(1):39–56, 1984.

[BPS81] BELADY, L. A., R. P. PARMELEE und C. A. SCALZI: *The IBM History of Memory Management Technology*. IBM Journal of Research and Development, 25(5):491–503, 1981.

[BR76] BLEVINS, PARKER R. und C. V. RAMAMOORTHY: *Aspects of a Dynamically Adaptive Operating System*. Transactions on Computers, 25(7):713–725, 1976.

[Bri64] BRIGHT, HERBERT S.: *A Philco Multiprocessing System*. AFIPS Conference, 26(2):97–141, 1964.

[BS75] BERNSTEIN, ARTHUR J. und PAUL SIEGEL: *A Computer Architecture for Level Structured Systems*. Transactions on Computers, 24(8):785–893, 1975.

[BS96] BRUSTOLONI, JOSE CARLOS und PETER STEENKISTE: *Effects of Buffering Semantics on I/O Performance*. OSDI, Seiten 277–291, 1996.

[BS02] BROY, MANFRED und JOHANNES SIEDERSLEBEN: *Objektorientierte Programmierung und Softwareentwicklung*. Informatik Spektrum, 25(1):3–11, 2002.

[BSS91] BIRMAN, KENNETH, ANDRE SCHIPER und PAT STEPHENSON: *Lightweight Causal and Atomic Group Multicast*. Transactions on Computer Systems, 9(3):272–314, 1991.

[C+94] CHASE, JEFFREY S. und OTHERS: *Sharing and Protection in a Single-Address-Space Operating System*. Transactions on Computer Systems, 12(4):271–307, 1994.

[Che87] CHERITON, DAVID R.: *UIO: A Uniform I/O System Interface for Distributed Systems*. Transactions on Computer Systems, 5(1):12–46, 1987.

[CJ75] COHEN, ELLIS und DAVID JEFFERSON: *Protection in the Hydra Operating System*. Symposium on Operating System Principles, Seiten 141–160, 1975.

[CL85] CHANDY, K. MANI und LESLIE LAMPORT: *Distributed Snapshots: Determining Global States of Distributed Systems*. Transactions on Computer Systems, 3(1):63–75, 1985.

[Cla85] CLARK, DAVID D.: *The Structuring of Systems Using Upcalls*. Symposium on Operating System Principles, Seiten 171–180, 1985.

[CLG+93] CHEN, PETER M., EDWARD K. LEE, GARTH GIBSON, RANDY H. KATZ und DAVID A. PATTERSON: *RAID: High-Performance, Reliable Secondary Storage*. Computing Surveys, 26(2):145–185, 1993.

[Cre81] CREASY, R. J.: *The Origin of hte VM/370 Time-Sharing System*. IBM Journal of Research and Development, 25(5):483–490, 1981.

[CW85] CARDELLI, LUCA und PETER WEGNER: *On Understanding Types, Data Abstraction, and Polymorphism*. Computing Surveys, 17(4):471–522, 1985.

[Dat95] DATE, C. J.: *An Introduction to Database Systems*. Addison Wesley, 1995.

[DDH72] DAHL, O.-J., E. W. DIJKSTRA und C. A. R. HOARE: *Structured Programming*. Academic Press, 1972.

[Den68] DENNING, PETER J.: *The Working Set Model for Program Behavior*. CACM, 11(5):323–333, 1968.

[Den71] DENNING, PETER J.: *Virtual Memory*. Computing Surveys, 2(3):153–189, 1971.

[DGMS85] DAVIDSON, SUSAN B., HECTOR GARCIA-MOLINA und DALE SKEEN: *Consistency in Partitioned Networks*. Computing Surveys, 17(3):341–370, 1985.

[DH66] DENNIS, JACK B. und EARL C. VAN HORN: *Programming Semantics for Multiprogrammed Computations*. CACM, 9(3):143–155, 1966.

[Dij65] DIJKSTRA, E. W.: *Solution of a Problem in Concurrent Programming Control*. CACM, 8(9):569, 1965.

[Dij68] DIJKSTRA, EDSGER W.: *The Structure of the "THE" Multiprogramming System*. CACM, 11(5):341–346, 1968.

[Dij71] DIJKSTRA, E. W.: *Hierarchical Ordering of Sequential Processes*. Acta Informatica, 1:115–138, 1971.

[dJ93] JONGE, WIEBREN DE: *The Logical Disk: A New Approach to Improving File Systems*. Symposium on Operating System Principles, Seiten 15–28, 1993.

[Doe96] DOEPPNER, THOMAS W.: *Distributed File Systems and Distributed Memory*. Computing Surveys, 28(1):229–231, 1996.

[DS72] DENNING, PETER J. und STUART C. SCHWARTZ: *Properties of the Working-Set Model*. CACM, 15(3):191–198, 1972.

[EKO95] ENGLER, DAWSON R., M. FRANS KAASHOEK und JAMES O'TOOLE: *Exokernel: An Operating System Architecture for Application-Level Resource Management*. Symposium on Operating System Principles, Seiten 251–266, 1995.

[Elmbe] ELMAGARMID, AHMED K.: *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, ohne Jahresangabe.

[EN95] ELMASRI, RAMEZ und SHAMKANT B. NAVATHE: *Fundamentals of Database Systems*. Addison Wesley, 1995.

[Esk96] ESKICIOGLU, M. RASIT: *A Comprehensive Bibliography of Distributed Shared Memory*. Operating System Reviews, 30(1):71–96, 1996.

[Fie88] FIELD, ANTHONY J.: *Functional Programming*. Addison-Wesley, 1988.

[Fot61] FOTHERINGHAM, JOHN: *Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of Backing Store*. CACM, 4(10):435–436, 1961.

[GC94] GOODHEART, BERNY und JAMES COX: *The Magic Garden Explained – The Internals of UNIX System V Release 4*. Prentice Hall, 1994.

[GR93] GRAY, JIM und ANDREAS REUTER: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra68] GRAHAM, ROBERT M.: *Protection in an Information Processing Utility*. CACM, 11(5):365–369, 1968.

[Hab69] HABERMANN, A. N.: *Prevention of System Deadlocks*. CACM, 12(7):373–385, 1969.

[Haj00] HAJJI, FARID: *Perl – Einführung, Anwendungen, Referenz*. Addison Wesley, 2000.

[Han70] HANSEN, PER BRINCH: *The Nucleus of a Multiprogramming System*. CACM, 13(4):238–250, 1970.

[Han73] HANSEN, PER BRINCH: *Concurrent Programming Concepts*. Computing Surveys, 5(4):223–245, 1973.

[Han77] HANSEN, PER BRINCH: *The Architecture of Concurrent Programs*. Prentice Hall, 1977.

[HFC76] HABERMANN, A. N., LAWRENCE FLON und LEE COOPRIDER: *Modularization and Hierarchy in a Family of Operating Systems*. CACM, 19(5):266–272, 1976.

[Hoa74] HOARE, C. A. R.: *Monitors: An Operating System Structuring Concept*. CACM, 17(10):549–557, 1974.

[Hoa78] HOARE, C. A. R.: *Communicating Sequential Processes*. CACM, 21(8):666–677, 1978.

[Hot74] HOTZ, GÜNTER: *Schaltkreistheorie*. De Gruyter, 1974.

[HP94] HEIDEMANN, JOHN S. und GERALD J. POPEK: *File-System Development with Stackable Layers*. Transactions on Computer Systems, 12(1):58–89, 1994.

[HP95] HEIDEMANN, JOHN und GERALD POPEK: *Performance of Cache Coherence in Stackable Filing*. Symposium on Operating System Principles, Seiten 127–142, 1995.

[HR83] HÄRDER, THEO und ANDREAS REUTER: *Principles of Transaction-Oriented Database Recovery*. Computing Surveys, 15(4):287–317, 1983.

[HR01] HÄRDER, THEO und ERHARD RAHM: *Datenbanksysteme – Konzepte und Techniken der Implementierung*. Springer-Verlag, 2. Auflage Auflage, 2001.

[Hur] *The GNU Hurd – GNU Project – Free Software Foundation (FSF)*. `http://www.gnu.org/software/hurd/hurd.html`.

[HW90] HERLIHY, MAURICE P. und JEANNETTE M. WING: *Linearizability: A Correctness Condition for Concurrent Objects*. Transactions on Programming Languages and Systems, 12(3):463–492, 1990.

[IEC] *IEC Standard 61131-3*. `http://www.holobloc.com/stds/iec/sc65bwg7tf3/html/news.htm`.

[Jef85] JEFFERSON, DAVID R.: *Virtual Time*. Transactions on Programming Languages and Systems, 7(3):404–425, 1985.

[JH02] JÄHNICHEN, STEFAN und STEPHAN HERRMANN: *Was, bitte, bedeutet Objektorientierung?* Informatik Spektrum, 25(4):266–275, 2002.

[Jon80] JONES, ANITA K.: *Capability Architecture Revisited*. Operating System Reviews, 14(3):33–35, 1980.

[Jür73] JÜRGENS, JÜRN: *Synchronisation paralleler Prozesse anhand von Zuständen*. Doktorarbeit, Technische Universität München, 1973.

[K+81] KAHN, KEVIN C. und OTHERS: *iMAX: A Multiprocessor Operating System for an Object-Based Computer*. Symposium on Operating System Principles, Seiten 127–136, 1981.

[K+91] KARLIN, A. und OTHERS: *Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor*. Symposium on Operating System Principles, Seiten 41–55, 1991.

[K+97a] KAASHOEK, M. FRANS und OTHERS: *Application Performance and Flexibility on Exokernel Systems*. Symposium on Operating System Principles, Seiten 52–65, 1997.

[K+97b] KICZALES, GREGOR und OTHERS: *Aspect-Oriented Programming*. In: *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer-Verlag, 1997. `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP9%7/for-web.pdf`.

[K+01] KICZALES, GREGOR und OTHERS: *An Overview of AspectJ*. In: *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer-Verlag, 2001. `http://aspectj.org/documentation/papersAndSlides/ECOOP2001-Overview.pdf%`.

[KE95] KLEIMAN, STEVE und JOE EYKHOLT: *Interrupts as Threads*. Operating System Reviews, 29(2):21–26, 1995.

[Knu94] KNUTH, DONALD E.: *The TEXbook*. Addison Wesley, 1994.

[LA93] LIM, BENG-HONG und ANANT AGARWAL: *Waiting Algorithms for Synchronization in Large-Scale Multiprocessors*. Transactions on Computer Systems, 11(3):253–294, 1993.

[Lag75] LAGALLY, KLAUS: *Das Projekt Betriebssystem BSM*. Technischer Bericht LRZ-Bericht 7502/1, gleichzeitig TUM-Bericht 7509, Leibnitz-Rechenzentrum München, 1975.

[Lag78] LAGALLY, K.: *Synchronization in a Layered System*. In: FLYNN, M. J. und OTHERS (Herausgeber): *Operating Systems, An Advanced Course*, Band 60 der Reihe *Lecture Notes in Computer Science*, Seiten 253–281. Springer-Verlag, 1978.

[Lam74] LAMPORT, LESLIE: *A New Solution of Dijkstra's Concurrent Programming Problem*. CACM, 17(8):453–455, 1974.

[Lam78] LAMPORT, LESLIE: *Time, Clocks, and the Ordering of Events in a Distributed System*. CACM, 21(7):558–565, 1978.

[Lam84] LAMPORT, LESLIE: *Using Time Instead of Timeout for Fault-Tolerant Distributed Systems*. Transactions on Programming Languages and Systems, 6(2):254–280, 1984.

[Lan81] LANDWEHR, CARL E.: *Formal Models for Computer Security*. Computing Surveys, 13(3):247–278, 1981.

[LCC+75] LEVIN, R., E. COHEN, W. CORWIN, F. POLLACK und W. WULF: *Policy/Mechanism Separation in Hydra*. Symposium on Operating System Principles, Seiten 132–140, 1975.

[LH89] LI, KAI und PAUL HUDAK: *Memory Coherence in Shared Virtual Memory Systems*. Transactions on Computer Systems, 7(4):321–359, 1989.

[Lie95a] LIEDTKE, JOCHEN: *Address Space Sparsity and Fine Granularity*. Operating System Reviews, 29(1):87–90, 1995.

[Lie95b] LIEDTKE, JOCHEN: *On μ-Kernel-Construction*. Symposium on Operating System Principles, Seiten 237–250, 1995.

[Lie95c] LIEDTKE, JOCHEN: *A Short Note on Small Virtually-Addressed Control Blocks*. Operating System Reviews, 29(3):31–34, 1995.

[Lio96] LIONS, JOHN: *Lions' Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, 1996.

[Lis72] LISKOV, BARBARA H.: *The Design of the Venus Operating System*. CACM, 15(3):144–149, 1972.

[LS87] LOCKEMANN, P. C. und J. W. SCHMIDT: *Datenbank-Handbuch*. Springer-Verlag, 1987.

[LS94] LIN, TEIN-HSIANG und KANG G. SHIN: *An Optimal Retry Policy Based on Fault Classification*. Transactions on Computers, 43(9):1014–1025, 1994.

[Mae87] MAES, PATTIE: *Concepts and experiments in computational reflection*. In: *Conference on Object Oriented Programming Systems Languages and Applications*, Seiten 147–155. ACM SIGPLAN, 1987. `http://doi.acm.org/10.1145/38765.38821`.

[McK96] MCKUSICK, MARSHALL KIRK: *Secondary Storage and Filesystems*. Computing Surveys, 28(1):217–219, 1996.

[Mey88] MEYER, BERTRAND: *Objektorientierte Software-Entwicklung*. Prentice Hall, 1988.

[MJLF84] MCKUSICK, MARSHALL K., WILLIAM N. JOY, SAMUEL J. LEFFLER und ROBERT S. FABRY: *A Fast File System for UNIX*. Transactions on Computer Systems, 2(3):181–197, 1984.

[MP81] MILLER, BARTON und DAVID PRESOTTO: *XOS: An Operating System for the X-Tree Architecture*. Operating System Reviews, 15(2):21–32, 1981.

[NS01] NEHMER, JÜRGEN und PETER STURM: *Systemsoftware, Grundlagen moderner Betriebssysteme*. dpunkt.verlag, 2001.

[NW74] NEEDHAM, R. M. und M. V. WILKES: *Domains of protection and the management of processes*. Computer Journal, 17(2):117–120, 1974.

[NW77] NEEDHAM, R. M. und R. D. H. WALKER: *The Cambridge CAP Computer and its protection system*. Symposium on Operating System Principles, Seiten 1–10, 1977.

[OpL] *Opportunistic Locks*. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%/base/opportunistic_locks.asp`.

[Opl65] OPLER, ASCHER: *Procedure-Oriented Language Statements to Facilitate Parallel Processing*. CACM, 8(5):306–307, 1965.

[Org72] ORGANICK, ELLIOT I.: *The Multics System: An Examination of Its Structure*. MIT Press, 1972.

[Pag81] PAGAN, FRANK G.: *Formal specification of programming languages: a panoramic primer*. Prentice Hall, 1981.

[Par72] PARNAS, D. L.: *On the Criteria To Be Used in Decomposing Systems into Modules*. CACM, 15(12):1053–1058, 1972.

[Par74] PARNAS, DAVID: *On a 'Buzzword': Hierarchical Structure*. Information Processing Conference (IPIP), Seiten 336–339, 1974.

[Par78] PARNAS, DAVID L.: *The Non-Problem of Nested Monitor Calls*. Operating System Reviews, 12(1):12–18, 1978.

[PC75] PRUITT, J. L. und W. W. CASE: *Architecture of a Real Time Operating System*. Symposium on Operating System Principles, Seiten 51–59, 1975.

[PDZ99] PAI, VIVEK S., PETER DRUSCHEL und WILLY ZWAENEPOEL: *IO-Lite: A Unified I/O Buffering and Caching System*. OSDI, Seiten 15–28, 1999.

[PDZ00] PAI, VIVEK S., PETER DRUSCHEL und WILLY ZWAENEPOEL: *IO-Lite: A Unified I/O Buffering and Caching System*. Transactions on Computer Systems, 18(1):37–6, 2000.

[Rei97] REISER, HANS: *Trees are Fast*. `http://www.namesys.com/`, 1997. (original version dated 27.6.1997 has been superseeded by a newer version).

[RK79] REED, DAVID P. und RAJENDRA K. KANODIA: *Synchronization with Eventcounts and Sequencers*. CACM, 22(2):115–123, 1979.

[RO91] ROSENBLUM, MENDEL und JOHN K. OUSTERHOUT: *The Design and the Implementation of a Log-Structured File System*. Symposium on Operating System Principles, Seiten 1–15, 1991.

[Ros94] ROSCOE, TIMOTHY: *Linkage in the Memesis Single Address Space Operating System*. Operating System Reviews, 28(4):48–55, 1994.

[RT74] RITCHIE, DENNIS M. und KEN THOMPSON: *The UNIX Time-Sharing System*. CACM, 17(7):365–375, 1974.

[Sch99] SCHMIDER, CHRISTOPH: *Untersuchung eines Fuzzy Hashing Verfahrens*. Studienarbeit, Universität Stuttgart, Fakultät Informatik, 1999.

[SG96] SHAW, MARY und DAVID GARLAN: *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[sig] *Signals and Slots*. `http://www.trolltech.com/products/embedded/whitepaper/qt-embedded-white%paper-5.html`.

[Smi88] SMITH, JONATHAN M.: *A Survey of Process Migration Mechanisms*. Operating System Reviews, 22(3):28–40, 1988.

[SPvE93] SLEEP, M. R., M. J. PLASMEIJER und M. C. J. D. VAN EEKELEN (Herausgeber): *Term Graph Rewriting: Theory and Practice*. Wiley, 1993.

[ST03a] SCHÖBEL-THEUER, THOMAS: *On Variants of Genericity*. In: *Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2003)*, Seiten 359–365. Knowlegde Systems Institute, 2003.

[ST03b] SCHÖBEL-THEUER, THOMAS: *A Way for Seamless Integration of Databases and Operating Systems*. In: *Proceedings of the International Conference on Computer Science and its Applications (ICCSA-2003)*, Seiten 82–89. National University, 2003.

[ST04a] SCHÖBEL-THEUER, THOMAS: *Generalized Optional Locking in Distributed Systems*. In: *to appear in PDCS 2004*. IASTED conference proceedings, Cambridge, MA, November 2004.

[ST04b] SCHÖBEL-THEUER, THOMAS: *Instance Orientation: a Programming Methodology*. In: *to appear in SEA 2004*. IASTED conference proceedings, Cambridge, MA, November 2004.

[ST04c] SCHÖBEL-THEUER, THOMAS: *Speculative Prefetching of Optional Locks in Distributed Systems*. In: *PDCN 2004, Innsbruck*. IASTED conference proceedings, 2004.

[Ste97] STEVENS, W. RICHARD: *Advanced Programming in the Unix Environment*. Addison-Wesley, 1997.

[Str78] STROUSTRUP, BJARNE: *On Unifying Module Interfaces*. Operating System Reviews, 12(1):90–98, 1978.

[Szy98] SZYPERSKI, CLEMENS: *Component Software*. Addison-Wesley, 1998.

[T+90] TANENBAUM, ANDREW S. und OTHERS: *Experiences with the Amoeba Distributed Operating System*. CACM, 33(12):47–63, 1990.

[TA90] TAY, B. H. und A. L. ANANDA: *A Survey of Remote Procedure Calls*. Operating System Reviews, 24(3):68–79, 1990.

[Tho96] THOMPSON, SIMON: *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[TL93] THEKKATH, CHANDRAMOHAN A. und HENRY M. LEVY: *Limits to Low-Latenct Communication on High-Speed Networks*. Transactions on Computer Systems, 11(2):179–203, 1993.

[TSF90] TAM, MING-CHIT, JONATHAN M. SMITH und DAVID J. FARBER: *A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems*. Operating System Reviews, 24(3):40–67, 1990.

[Vah96] VAHALIA, URESH: *UNIX Internals*. Prentice Hall, 1996.

[VGH93] VOSSEN, GOTTFRIED und MARGRET GROSSHARDT: *Grundlagen der Transaktionsverarbeitung*. Addison-Wesley, 1993.

[Vid87] VIDYASANKAR, K.: *Generalized Theory of Serializability*. Acta Informatica, 24:105–119, 1987.

[Y+90] YOKOTE, YASUHIKO und OTHERS: *The Muse Object Architecture: A New Operating Systems Structuring Concept*. Operating System Reviews, 25(2):22–46, 1990.

[Yok92] YOKOTE, YASUHIKO: *The Apertos Reflective Operating System: The Concept and Its Implementation*. In: *OOPSLA'92 Proceedings*, Seiten 414–434. ACM, 1992. Sony CSL technical report SCSL-TR-92-014, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/92/SCSL-TR-92-014.ps.Z`.

[YTT89] YOKOTE, YASUHIKO, FUMIO TERAOKA und MARIO TOKORO: *A Reflective Architecture for an Object-Oriented Distributed Operating System*. In: *European Conference on Object-Oriented Programming (ECOOP)*, 1989. Sony CSL technical report SCSL-TR-89-001, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/89/SCSL-TR-89-001.ps.Z`.

[YTY+89] YOKOTE, YASUHIKO, FUMIO TERAOKA, MASAKI YAMADA, HIROSHI TEZUKA und MARIO TOKORO: *The Design and Implementation of the Muse Object-Oriented Distibuted Operating System*. In: *Conference on Technology of Object-Oriented Languages and Systems*, 1989. Sony CSL technical report SCSL-TR-89-010, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/89/SCSL-TR-89-010.ps.Z`.