Eine Methode zur Konstruktion zuverlässiger Software-Systeme für den Einsatz im Produktionsbereich

Der Technischen Fakultät
der Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Dipl.-Inf. Anton Friedl

Erlangen 1984

DI (mas 84-85)

Eine Methode zur Konstruktion zuverlässiger Software -
Systeme für den Einsatz im Produktionsbereich

Der Technischen Fakultät der
Universität Erlangen Nürnberg

zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Dipl.-Inf. Anton Friedl

Erlangen 1984

Als Dissertation genehmigt von der
Technischen Fakultät der
Universität Erlangen - Nürnberg

Tag der Einreichung	29.05.1984
Tag der Promotion	15.11.1984
Dekan:	Prof. Dipl.-Ing. G. Vetter
Berichterstatter:	Prof. Dr.-Ing. K. Feldmann
	Priv.-Doz. Dr.-Ing. S. Keramidis

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Fertigungsautomatisierung und Produktionssystematik der Universität Erlangen-Nürnberg.

Herrn Professor Dr.-Ing. K. Feldmann, dem Inhaber des Lehrstuhls für Fertigungsautomatisierung und Produktionssystematik danke ich für seine wohlwollende Unterstützung und großzügige Förderung bei der Durchführung dieser Arbeit.

Herr Privatdozent Dr.-Ing. habil. S. Keramidis hat diese Arbeit angeregt und ihren Fortgang stets mit großem Interesse verfolgt. Für seinen Einsatz und seine konstruktive Kritik möchte ich ihm an dieser Stelle herzlich danken.

Ferner gilt mein Dank allen Mitarbeiterinnen und Mitarbeitern des Lehrstuhls für Fertigungsautomatisierung und Produktionssystematik für ihre stete Hilfsbereitschaft. Insbesondere danke ich Frau A. Donhauser und Herrn cand. ing. H. Pirner für ihre Hilfe bei der Erstellung der Zeichnungen und Herrn cand. inf. H Spintzik, der mir im Rahmen seiner Diplomarbeit beim Entwurf und der Realisierung des Anwendungsbeispiels behilflich war.

Nicht zuletzt möchte ich mich an dieser Stelle bei meiner Frau Petra bedanken, die durch ihr Verständnis und den Verzicht auf manche gemeinsame Stunde erst die Durchführung dieser Arbeit ermöglichte.

Anton Friedl

INHALTSVERZEICHNIS Seite

1. Einleitung und Überblick

Durch die zunehmende Ausbreitung der elektronischen Datenverarbeitung, den laufenden Preisverfall bei der Hardware und die Lösung immer komplexerer Probleme durch Programmsysteme gewann die Software im letzten Jahrzehnt immer mehr an Bedeutung. So verschob sich in diesem Zeitraum das Verhältnis der Software- zu Hardwarekosten bei DV-Installationen kontinuierlich zugunsten der Software, und der Anteil der Software liegt heute bei komplexeren Anwendungen in Größenordnungen von 90 %.

Erstaunlicherweise stand dieser wachsenden Bedeutung der Software bis vor wenigen Jahren ein Vorgehen bei der Erstellung von Software gegenüber, das weder als methodisch, noch als rationell zu bezeichnen war. Die Konsequenz daraus war die vielzitierte SW-Krise mit qualitativ schlechten Programmen und oftmals erschreckend hohen Kosten für deren Erstellung. Die realen Kosten für die Software lagen häufig um bis zu 100 % höher als die kalkulierten.

Einen großen Anteil an diesen Kostensteigerungen machten hier Arbeiten aus, die erst nach der eigentlichen Fertigstellung der Programme, vor allem in der Test- und Inbetriebnahmephase anfielen. Selbst konzeptionelle Fehler wurden noch in diesen Phasen entdeckt. Besonders katastrophal wirkten sich auch notwendige Korrekturen, Änderungen oder Ergänzungen von SW-Komponenten aus, da große Systeme oftmals nicht überblickbar waren. Insbesondere war das Zusammenspiel der Einzelkomponenten und das Verhalten des Systems in Extremsituationen nicht vorhersehbar.

Vor dem Hintergrund dieser Entwicklungen entstanden dann ab Anfang der 70er Jahre unter dem Begriff "Software-Engineering" erste Ansätze zur Lösung dieser Probleme. Man hatte erkannt, daß nur ein ingenieurmäßiges, d.h. methodisches und rationelles Vorgehen bei der Erstellung von Software Verbesserungen bringen konnte. Besonders für die Phasen vor der eigentlichen Programmierung mußten Hilfs-

mittel zur Verfügung gestellt werden, die es ermöglichten, die Anforderungen an ein SW-System exakt zu beschreiben, diese Anforderungsbeschreibung in eine verarbeitungsorientierte Form umzusetzen, und den SW-Entwurfsvorgang ausreichend zu dokumentieren. Bis heute entstanden so zahlreiche Werkzeuge und Methoden zur Unterstützung des SW-Entwurfs, die vor allem für den Einsatz bei kommerziellen Anwendungen, die durch eine rein sequentielle Verarbeitung gekennzeichnet sind, konzipiert waren.

Betrachtet man dagegen nichtsequentielle Systeme, so sind die meisten der entstandenen Entwurfsmethoden wegen der wesentlich komplexeren Anforderungen nicht mehr einsetzbar. So erfordern z.B. Systeme zur Prozeßdatenverarbeitung, Automatisierungssoftware oder Dialogsysteme Mechanismen zur Synchronisation und prioritätengesteuerten Verarbeitung der einzelnen, parallel ablaufenden Systemkomponenten. Ein weiterer wichtiger Gesichtspunkt ist der Schutz gemeinsam genutzter Ressourcen oder Daten vor unerlaubtem Zugriff, Zerstörung oder Modifikation. Hierzu kommt bei einer Verarbeitung personenbezogener Daten, wie sie z.B. bei medizinischen Anwendungen oder im Personalwesen eines Fertigungsbetriebs vorkommt, noch ein Schutz vor unerlaubter Enthüllung.

Gerade für diese Einsatzfälle sind nun aber Methoden, die den Entwurf zuverlässiger, möglichst verifizierbarer Software ermöglichen, von besonderem Interesse. Zum einen ist die Konstruktion nichtsequentieller Systeme ein im allgemeinen sehr schwieriges Gebiet, so daß die Fehlermöglichkeiten gegenüber kommerziellen Anwendungen wesentlich größer sind und Fehler wesentlich schwieriger zu erkennen und zu beheben sind, da sie oftmals nur bei bestimmten Systemkonstellationen auftreten und nicht reproduzierbar sind. Zum anderen wirken sich gerade hier Softwarefehler häufig katastrophal aus. Man denke beispielsweise an die Steuerung und Überwachung von Produktionslinien, eines flexiblen Fertigungssystems oder gar eines Kernkraftwerks.

Ziel dieser Arbeit ist deshalb die Entwicklung von Hilfsmitteln zur Konstruktion zuverlässiger SW-Systeme unter Berücksichtigung paralleler Abläufe und Schutzanforderungen. Die hier präsentierte Methodik bietet dabei einen einheitlichen Ansatz für die Spezifikation und Implementierung geschützter, asynchroner Systeme, wobei eine wesentliche Eigenschaft der gewählten Spezifikationskonstrukte eine einfache Ableitungsmöglichkeit der zugehörigen Implementierung ist. Zum Nachweis der Zuverlässigkeit können schließlich Verifikationsbedingungen angegeben werden.

Als Basis für die in dieser Arbeit angegebene Spezifikations- und Implementierungsmethodik für asynchrone, geschützte Systeme dient ein formales mathematisches Modell, das die Semantik der Methodik festlegt. Der Weg über ein formales Modell wurde dabei aus Gründen der Allgemeinheit des Ansatzes gewählt. Ein weiterer Vorteil, der sich durch die Zugrundelegung des formalen Modells ergibt, ist die Möglichkeit, anwendungsbezogene Fragen, wie z.B. die Verträglichkeit asynchroner Aktivitäten, bereits im formalen Modell zu formulieren bzw. zu klären. Schließlich wird durch die Zugrundelegung des Modells sowohl für die Spezifikation als auch Implementierung die Einheitlichkeit des Ansatzes, die einfache Ableitung der Implementierung aus der Spezifikation und die leichte Verifizierbarkeit der Korrektheit der Implementierung entscheidend gefördert.

An dieser Stelle sei erwähnt, daß im letzten Jahr am Institut für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg drei Dissertationen (<Mac83>, <Rei83>, <Web83>) entstanden, die sich mit Aufgabenstellungen befassen, die dem in dieser Arbeit gesteckten Ziel teilweise verwandt sind. Jede dieser Arbeiten betrachtet jedoch das breite Themengebiet der Entwicklung von Methoden zur Konstruktion zuverlässiger SW-Systeme mit anderen Schwerpunkten, so daß sich die vorliegende Arbeit und diese Dissertationen gegenseitig ergänzen: Während in <Web83> schwerpunktmäßig Recovery-Maßnahmen zur Zuverlässigkeitssteigerung untersucht werden, beschränkt sich Mackert

in <Mac83> auf asynchrone Systeme ohne Schutzanforderungen. In <Rei83> schließlich werden zwar Schutzaspekte berücksichtigt, jedoch ist der dort gewählte Ansatz rein sprachlich. Es fehlt also im Gegensatz zur vorliegenden Arbeit eine formale Modellierung als Basis des Spezifikations- und Implementierungskonzepts. Zudem berücksichtigt die dort gewählte Definition von Schutz weder die Dynamik von Berechnungen noch im Zusammenhang mit semantischer Abfrage und Veränderung sich ergebende Probleme. Im folgenden soll nun eine überblicksartige Darstellung der in dieser Arbeit behandelten Schwerpunkte gegeben werden.

Den Einstieg in die Arbeit bildet ein Überblick über die Spezifika von SW-Systemen für Echtzeitanwendungen in der Produktion. Ausgehend von einer Darstellung typischer Aufgabenstellungen werden die Anforderungen an Hardware und Software bei diesen Anwendungen erläutert. Grundlegendste Voraussetzung ist hierbei die Zuverlässigkeit der eingesetzten Komponenten und SW-Module. Für die SW-Erstellung bedeutet dies die Notwendigkeit des Einsatzes von Methoden zum SW-Engineering. Abschließend werden die erforderlichen Erweiterungen gegenüber existierenden Methoden erläutert.

Im Kapitel 3 der Arbeit werden dann Prinzipien und Methoden zur Konstruktion zuverlässiger SW-Systeme vorgestellt. Den Anfang bildet eine Darlegung der Charakteristika und Qualitätsmerkmale des Produkts "Software" - im Gegensatz zu anderen, herkömmlichen technischen Produkten. Die anschließende Erläuterung des "SW-Life-Cycle", d.h. der einzelnen Phasen innerhalb der SW-Entwicklung, leitet dann über zu einem Überblick über verschiedene SW-Entwurfsprinzipien. Hier werden u.a. die Begriffe Abstraktion, Modularisierung, Strukturierung, Lokalität, Spezifikation und Verifizierbarkeit erläutert. Die Verwirklichung dieser Prinzipien zeigt sich dann (mehr oder weniger) in den existierenden Methoden für das SW-Engineering, die kurz vorgestellt werden sollen.

Als geeignetste Methode für den Programmentwurf stellt sich hierbei das Konzept der Abstrakten Datentypen heraus, da es alle erläuterten Entwurfsprinzipien automatisch erfüllt. Es wird deshalb in dieser Arbeit als Basiskonzept für das zu entwickelnde formale Modell, die Spezifikationsmethode und die Implementierung verwendet. Abstrakte Datentypen stellen dem Designer eines Systems eine Basis für prozedurale und Datenabstraktion zur Verfügung und definieren per se ein konsistenzerhaltendes Modell für Objekte. Die darüber hinaus noch in das ADT-Konzept zu integrierenden Objektverwaltungsmechanismen, wie Synchronisation der Zugriffe auf Objekte und Schutzaspekte, werden abschließend kurz erläutert.

Im Kapitel 4 der Arbeit werden ausführlich die existierenden Methoden zur Realisierung von Schutz- und Sicherungsmechanismen vorgestellt. Die Notwendigkeit hierfür ergibt sich dabei aus der Forderung der Integration von Schutzmechanismen in das ADT-Konzept. Den Einstieg in das Kapitel bildet die Vorstellung reiner Zugriffsschutzmodelle, wie z.B. das Modell von Harrison oder die Take-Grant-Modelle. Anschließend werden Modelle für militärische Sicherheit, wie das Bell-LaPadula-Modell oder das Modell von Feiertag präsentiert. Bei Informationsflußmodellen werden, sowohl rein syntaktische Ansätze (Denning), als auch semantische (Cohen) berücksichtigt.

Alle diese Modelle beschränken sich auf einen ganz bestimmten Teilaspekt des Gesamtgebiets Schutz, sind also nicht vollständig. Dieser Mangel wird bei den Modellen von Goguen und Stoughton zu vermeiden versucht, da hier sowohl Zugriffsschutz- als auch Informationsflußaspekte berücksichtigt werden. Abschließend werden alle präsentierten Modelle miteinander verglichen und insbesondere an den Anforderungen, die in Kapitel 3 an ein formales Modell zur Objektverwaltung gestellt wurden, gemessen. Schließlich werden die Forderungen, die sich aus der Sicht, des Gebiets Schutz an unser allgemeines Objektverwaltungsmodell stellen, zusammenfassend dargelegt.

Gegenstand des Kapitels 5 ist dann die Erstellung des formalen Modells zur Objektverwaltung. Basis hierfür ist die sogenannte Betriebssystemmaschine <Ker82>, ein allgemeines formales Modell für Betriebssysteme. Dieses Modell wird hier in zwei Schritten erweitert:

Zunächst werden in die ursprüngliche Betriebssystemmaschine Schutzkonstrukte integriert und verschiedene Möglichkeiten der Definition von Abfrage und Veränderung einer Variablen durch eine Operation - als Basis für Schutz- und Synchronisationsfragen - diskutiert.

Anschließend wird das Konzept der Abstrakten Datentypen als Grundlage für prozedurale und Datenabstraktion in die erweiterte Betriebssystemmaschine integriert. Eine formale Definiton Abstrakter Datentypen und des zugehörigen Objektverwaltungsmechanismus, der sogenannten abstrakten Objektverwaltungsmaschine, bildet die Grundlage für die weiteren Untersuchungen. So werden ausführlich die Möglichkeiten zur Definition von Schutzstrategien (policies) im Modell diskutiert und der für die Zuverlässigkeit eines Systems grundlegende Begriff der Sicherheit eines ADT`s definiert.

Aufbauend auf diesem formalen Modell wird schließlich in Kapitel 6 der Arbeit ein Spezifikations- und Implementierungskonzept für synchronisierte, geschützte Abstrakte Datentypen präsentiert. Nach einer Einordnung des Konzepts und der Erläuterung des Zusammenhangs zwischen formalem Modell, Spezifikations- und Implementierungskonzept werden zuerst die Spezifikationskonstrukte angegeben. Hierbei wird die in <Ker82> definierte Syntax im wesentlichen übernommen und um Sprachkonstrukte für die eingeführten Schutzmechanismen erweitert. Für die Verifikation der Sicherheit synchronisierter, geschützter ADTs auf Spezifikationsebene und die korrekte Implementierung von Spezifikationen werden schließlich Verifikationsregeln und Beispiele angegeben.

Den Abschluß der Arbeit bildet in Kapitel 7 die Spezifikation eines größeren Beispiels aus dem Bereich der Automatisierung. Hier wird die Anwendbarkeit der präsentierten Sprachkonstrukte in der Praxis und auch die Notwendigkeit der Entwurfsunterstützung bei SW-Systemen für Echtzeitanwendungen in der Produktion noch einmal deutlich dargelegt.

2. SW-Systeme für Echtzeitanwendungen in der Produktion

Ziel dieses Kapitels ist, die Besonderheiten von Softwaresystemen für Echtzeitanwendungen in der Produktion - insbesondere die Sicherheits- und Zuverlässigkeitsanforderungen an solche Systeme - herauszustellen, und Ansätze zur Erfüllung dieser Anforderungen aufzuzeigen.

Zunächst werden exemplarisch einige typische Anwendungsfälle erarbeitet - von der Prozeßdatenverarbeitung über die Steuerung und Überwachung von Produktionseinrichtungen bis hin zu Dialogsystemen in der Produktion. Die besonderen Anforderungen, die solche Systeme an Rechenanlagen, Datenübertragungssysteme und das Betriebssystem stellen, werden dargelegt. Insbesondere der Forderung nach Sicherheit und Zuverlässigkeit wird dabei in der Diskussion ein breiter Raum zugestanden.

Nach einer Definition der beiden Begriffe "Sicherheit" und "Zuverlässigkeit" werden Maßnahmen zur Zuverlässigkeitssteigerung von Hardware und Software diskutiert. Für die Software werden hierbei zwei unterschiedliche Strategien unterschieden - die sogenannte Fehlertoleranzstrategie und vorbeugende Maßnahmen bei der Programmerstellung. Als entscheidende Voraussetzung zur Entwicklung zuverlässiger Programmsysteme stellt sich der Einsatz von SW-Engineering-Methoden heraus.

Aufgrund der Besonderheiten von SW-Systemen für Echtzeitanwendungen in der Produktion können konventionelle SW-Engineering-Methoden nicht direkt eingesetzt werden. Die zusätzlichen Anforderungen an Entwurfsmethoden werden deshalb vorgestellt und somit die Überleitung zum folgenden Kapitel geschaffen, in dem Prinzipien und Methoden des SW-Engineering präsentiert werden.

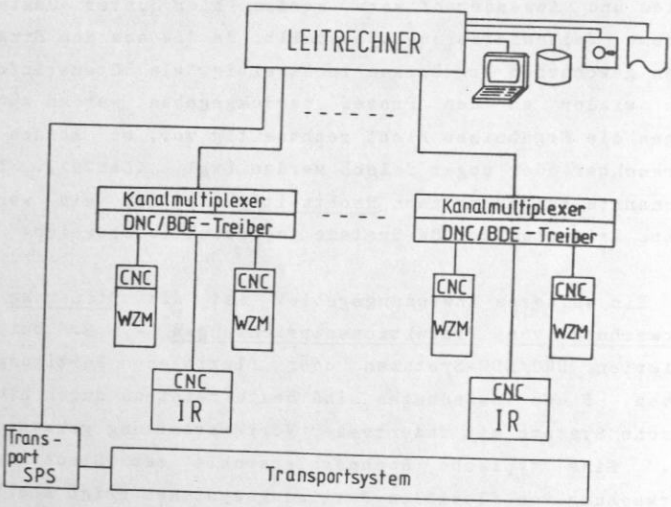
2.1 Typische Aufgabenstellungen

Das klassische Einsatzgebiet für Prozeßrechner und damit eine erste Aufgabenstellung für Echtzeit-SW-Systeme in der Produktion ist die Prozeßdatenverarbeitung, d.h. die Steuerung und Überwachung technischer Prozesse - zum Beispiel in der chemischen Industrie oder in Walzwerken. Charakteristisch für diese Anwendungen ist eine Erfassung und zeitgerechte Verarbeitung einer Fülle von Sensorsignalen in digitaler oder analoger Form. An Rechnerhardware, Betriebssystem und Anwendersoftware werden hier unter Umständen extreme Zeitanforderungen gestellt, da die aus den Eingabedaten gewonnenen Ergebnisse rechtzeitig als Steuerinformation wieder an den Prozeß zurückgegeben werden müssen. Liegen die Ergebnisse nicht rechtzeitig vor, so können sie unbrauchbar oder sogar falsch werden (vgl. <Lau76>). Diese sogenannte Forderung nach Rechtzeitigkeit ist ein wesentliches Kriterium für SW-Systeme im Produktionsbereich.

Ein weiteres Anwendungsgebiet ist die Steuerung und Überwachung von Produktionseinrichtungen - z.B. bei kombinierten DNC/BDE-Systemen oder flexiblen Fertigungssystemen. Diese Anwendungen sind heute meistens durch hierarchische Systeme mit dezentraler Vorverarbeitung gekennzeichnet. Eine typische Rechnerhierarchie zur Steuerung und Überwachung von flexiblen Fertigungssystemen zeigt Abbildung 2-1.

Die Steuerungs- und Überwachungsfunktionen sind hier auf drei Ebenen verteilt. Auf der untersten Ebene der CNC-Steuerungen laufen die extrem zeitkritischen Anforderungen, wie die Versorgung der Werkzeugmaschinen mit geometrischen und technologischen Daten, die direkte Steuerung der Industrieroboter oder des Transportsystems ab. In der Leitrechnerebene werden die mehr organisatorischen und dispositiven Aufgaben wie die DNC-Organisation, die organisatorische Steuerung des Materialflusses, die Verarbeitung der erfaßten Betriebsdaten oder die Ermittlung einer optimalen Maschinenbelegung durchgeführt. Auch die Kommunikation mit

übergeordneten CAD/CAP- oder PPS-Systemen fällt in den Aufgabenbereich des CAM-Leitrechners. Die Zwischenebene dient neben einer Schnittstellenvervielfachung und der entsprechenden Verteilung der Steuerdaten zur dezentralen Vorverarbeitung der erfaßten Istdaten und damit zu einer weiteren zeitlichen Entkopplung des Leitrechners.



CNC .. Computerized Numerical Control
WZM .. Werkzeugmaschine
IR ... Industrieroboter
SPS .. Speicherprogrammierbare Steuerung

Abb. 2-1: Rechnerhierarchie zur Steuerung und Überwachung von flexiblen Fertigungssystemen

Trotz dieser Entlastung des Leitrechners von extrem zeitkritischen Aufgaben werden bei diesem Anwendungsfall an Hardware und Software natürlich noch Echtzeitanforderungen gestellt. So laufen im Leitrechner die Steuerungs- und Überwachungspfade der Werkzeugmaschinen, Industrieroboter und des Transportsystems zusammen, d.h. insbesondere die organisatorische Steuerung der Handhabungs- und Transportvorgänge erfordert eine zeitgerechte Verknüpfung der jeweiligen Systemkomponenten. Beispielsweise muß nach der Bearbeitung eines Werkstücks auf einer Werkzeugmaschine ein Werkstück- und u.U. auch Werkzeugwechsel erfolgen. Dies erfordert den Anstoß entsprechender Handhabungs- und Transportvorgänge. Auch hier ist naturgemäß die Forderung nach Rechtzeitigkeit zu stellen.

Wegen der oftmals sehr hohen Anzahl an zu steuernden und zu überwachenden Systemkomponenten und dem daraus resultierenden gleichzeitigen Anfall von Anforderungen an den Rechner, muß dieser in der Lage sein, diese Anforderungen quasi simultan zu bearbeiten. Diese Forderung nach Gleichzeitigkeit ist ein weiteres Charakteristikum für SW-Systeme im Produktionsbereich (<Lau76>).

Als letzter Anwendungsfall für Echtzeitsysteme sei hier der Dialogbetrieb für interaktives Arbeiten am Rechner präsentiert. Obwohl hier weder Sensordaten verarbeitet, noch Prozesse oder Produktionsanlagen gesteuert werden, handelt es sich um eine zeitkritische Anwendung. Die erforderliche Systemantwortzeit auf die Eingabe eines Benutzers entspricht der für den Benutzer maximal zumutbaren Wartezeit und liegt in der Größenordnung von einigen Sekunden. Daß trotz dieser für ein Rechnersystem an und für sich sehr hohen Antwortzeit unter Umständen erhebliche Zeitprobleme bei Dialogsystemen entstehen können, liegt nun entweder an den komplexen Algorithmen, die ausgeführt werden, an extrem großen Datenmengen, die verarbeitet werden, oder an der hohen Zahl der angeschlossenen Terminals und damit der gleichzeitig anfragenden Benutzer.

Typische Beispiele für dialogorientierte Systeme im Produktionsbereich sind CAD-Systeme oder Systeme zur Lagerverwaltung und kurzfristigen Fertigungssteuerung. Weitere charakteristische Anwendungsbeispiele sind Buchungssysteme bei Banken und Fluggesellschaften oder Datenbanksysteme.

Anhand der gezeigten typischen Aufgabenstellungen wurden die für SW-Systeme im Produktionsbereich wesentlichen Forderungen nach Rechtzeitigkeit und Gleichzeitigkeit begründet. Im folgenden Abschnitt sollen nun die zur Erfüllung dieser beiden Forderungen an Hardware und Betriebssystem zu stellenden Anforderungen dargelegt werden.

2.2 Besondere Anforderungen an HW und Betriebssystem bei Echtzeitanwendungen

Die Forderung nach Rechtzeitigkeit und Gleichzeitigkeit für SW-Systeme im Produktionsbereich läßt sich direkt auf den Ablauf der einzelnen Programme des Systems übertragen. Als Verfahren zur Erfüllung dieser Anforderungen hat sich die Methode der asynchronen Programmierung oder Parallelprogrammierung (<Lau76>) durchgesetzt.

Grundsätzlich werden bei dieser Methode keinerlei Voraussetzungen über den jeweiligen Zeitpunkt der Ausführung von Programmen gemacht, d.h.

- die einzelnen Programme können asynchron, d.h. zu beliebigen Zeitpunkten ablaufen.
- die zeitliche Reihenfolge, in der die Programme ablaufen, wird nicht fest vorgegeben.
- für "Konfliktfälle", in denen mehrere Anforderungen zur Ausführung parallel ablaufender Programme zeitlich zusammentreffen, wird eine Strategie festgelegt, wie zu verfahren ist.

Nicht jede Rechenanlage (bzw. Betriebssystem) ist allerdings für den Ablauf asynchroner Systeme und die sonstigen Erfordernisse bei Echtzeitanwendungen, wie z.B. die Erfassung von Sensorsignalen, geeignet. Die besonderen Anforderungen, die an HW und Betriebssystem bei Echtzeitanwendungen zu stellen sind, sollen deshalb hier kurz dargelegt werden.

Rechenanlagen

Die Besonderheiten bei Rechenanlagen für Echtzeitanwendungen betreffen zunächst die Kommunikation mit der angeschlossenen Peripherie und die Reaktion auf externe Ereignisse. So wird neben einem schnellen Reaktionsvermögen auf Unterbrechungssignale ein schneller Datenverkehr mit den angeschlossenen Geräten gefordert. Weitere Voraussetzungen für die Rechnerhardware sind eine große Anzahl anschließbarer Geräte, die u.U. über Schnittstellenvervielfacher (Multiplexer) betrieben werden, und eine flexible E/A-Steuerung, die den Anschluß unterschiedlichster Peripherie ermöglicht. Dies wird besonders deutlich, wenn man das breite Spektrum der anzuschließenden Geräte - vom Prozeßelement über Numerische und Speicherprogrammierbare Steuerungen bis hin zu BDE-Terminals und normalen Bildschirmterminals - betrachtet.

Zur Ermöglichung des Ablaufs asynchroner Aktivitäten muß die Rechnerhardware dem Betriebssystem Instruktionstypen zur Verfügung stellen, die eine Koordinierung paralleler Aufgaben durch das Betriebssystem gestatten. Die oben angesprochene Strategie für Konfliktfälle wird meistens mit Hilfe von Prioritäten, die an die einzelnen parallel ablaufenden Programme vergeben werden, definiert. Als Basis zur Durchsetzung dieser Strategien müssen in der Hardware Mechanismen zur Definition von Prioritätsebenen existieren.

Datenübertragungssysteme

Systeme zur Datenübertragung spielen bei Echtzeitanwendungen eine bedeutende Rolle wegen der erforderlichen Kommunikation zwischen Rechner und peripheren Geräten bzw. bei hierarchischen Systemen zwischen den einzelnen Rechnern untereinander. Zu beachten ist, daß oftmals die Entfernungen zu den Peripheriegeräten in Größenordnungen von vielen hundert Metern liegen.

Für die Datenübertragung zwischen Rechner und den angeschlossenen Geräten unterscheidet man zwischen verschiedenen Organisationsformen (Abbildung 2-2).

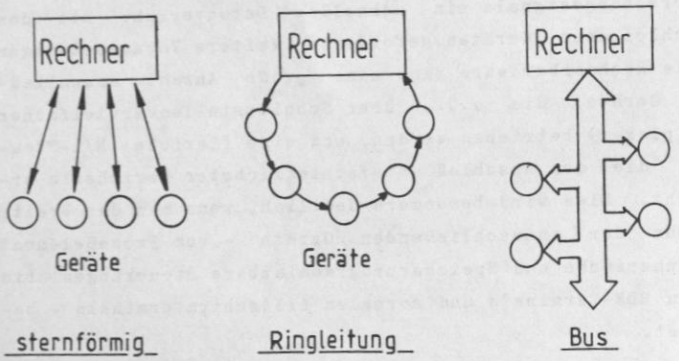


Abb. 2-2: Organisationsformen für die Datenübertragung

Bei der sternförmigen Anordnung verfügt jedes einzelne Gerät über eine eigene Datenleitung zum Rechner. Der relativ sicheren Datenübertragung - bei Ausfall eines Geräts oder eines Übertragungswegs bleiben die anderen unbeein-

flußt - stehen im allgemeinen sehr hohe Verkabelungskosten gegenüber. Gerade umgekehrt verhält sich die Ringleitung, die heute kaum mehr zum Einsatz kommt. Dagegen werden Bussysteme, die einen gemeinsamen Datenpfad für alle Geräte benutzen, immer häufiger für die Kommunikation zwischen Rechner und peripheren Geräten oder zwischen verschiedenen Rechnern verwendet. Hierfür hat sich in den letzten Jahren der Begriff des "Local Area Network (LAN)" herausgebildet.

Neben der Organisationsform der Datenübertragung bildet die Übertragungsart ein weiteres Unterscheidungskriterium für Datenübertragungssysteme. Man kann hier zwischen paralleler, byte-serieller und bit-serieller Übertragung unterscheiden. Während bei paralleler Übertragung die gesamte Wortlänge zwischen Rechner und Peripherie parallel übertragen wird, werden im byte-seriellen Fall die Worte in Bytes zerlegt, die Bytes einzeln übertragen und am Empfangsort wieder zum ursprünglichen Wert zusammengesetzt. Bei bit-serieller Übertragung wird völlig analog verfahren, nur daß die zu übertragenden Daten hier vollständig serialisiert werden.

Als geeignete Kombinationen von Organisationsform und Übertragungsart haben sich in der Praxis verschiedene Lösungen etabliert. So kommt bei der rechnerinternen Datenübertragung z.B. zwischen Hauptspeicher und Zentraleinheit i.a. ein schneller Bus mit paralleler Übertragung zum Einsatz. Für die Übertragung zur Peripherie werden entweder langsamere Busse mit byte-serieller Übertragung (z.B. IEC-Bus, PDV-Bus) verwendet oder bit-serielle Übertragungsverfahren über serielle Schnittstellen wie die V24- oder 20mA-Schnittstelle. Die Anordnung ist hier meist sternförmig.

Betriebssysteme

Hauptaufgabe von Echtzeit-Betriebssystemen ist die Gewährleistung der rechtzeitigen und gleichzeitigen Ausführung einer Vielzahl von Automatisierungsprogrammen, d.h. die asynchrone Ausführung dieser Programme muß ermöglicht und organisiert werden. Insbesondere ist in den bereits angesprochenen Konfliktfällen, in denen mehrere Rechenprozesse gleichzeitig auf dem Prozessor ablaufen sollen, gemäß der angegebenen Strategie die Reihenfolge des Ablaufs der Rechnerprozesse zu organisieren.

Weitere Konfliktfälle treten auf, wenn mehrere Rechenprozesse gleichzeitig auf bestimmte Ressourcen (z.B. ein Peripheriegerät oder gemeinsame Daten) zugreifen wollen. Die Aufgabe des Betriebssystems ist in diesen Fällen die Synchronisation dieser Zugriffswünsche. Das Betriebssystem muß also Mechanismen zur Verwaltung der Zugriffe auf Ressourcen (bzw. Betriebsmittel) zur Verfügung stellen.

In der Praxis werden diese Zugriffs-Verwaltungsaufgaben durch das Anlegen von Warteschlangen vor den Betriebsmitteln realisiert. Diese Warteschlangen sind dann vom Betriebssystem gemäß der gewählten Strategie abzuarbeiten. Wie bereits erwähnt, wird diese Strategie im allgemeinen durch die Vergabe von Prioritäten an die verschiedenen Prozesse festgelegt. Im Betriebssystem müssen deshalb unterschiedliche Prioritäten für Anwenderprogramme definierbar sein.

Neben der Organisation von asynchron ablaufenden Aktivitäten muß ein Echtzeit-Betriebssystem in der Lage sein, auf HW-Unterbrechungssignale sofort zu reagieren. Ziel dieser Unterbrechungssignale ist nämlich, die momentan ablaufenden Prozesse zugunsten von Aufgaben höherer Priorität - die durch das Unterbrechungssignal angestoßen werden sollen - zu unterbrechen. Wesentliche Forderung dabei ist natürlich die störungsfreie Fortsetzung des unterbrochenen Prozesses nach Beendigung der dringlicheren Aufgabe.

Die Forderung nach Anschlußmöglichkeiten für unterschiedlichste Peripheriegeräte setzt beim Betriebssystem die Möglichkeit voraus, verschiedene Treiberprogramme zur Überwachung der E/A-Operationen zu integrieren. Auch eine Kommunikation mit anderen Rechnern muß über Standardschnittstellen möglich sein.

2.3 Sicherheits- und Zuverlässigkeitsanforderungen an Hardware und Software

Naturgemäß ist eine der wesentlichsten Anforderungen an ein automatisiertes System dessen Sicherheit und Zuverlässigkeit. In diesem Abschnitt soll deshalb der Diskussion dieser beiden Eigenschaften ein ihrer Bedeutung entsprechend breiter Raum eingeräumt werden. Im Vordergrund stehen dabei nach einer Begriffsklärung Maßnahmen zur Steigerung der Zuverlässigkeit von Hardware, Betriebssystem und Anwendersoftware für Echtzeitanwendungen in der Produktion.

Unter dem Begriff Sicherheit soll bei einem automatisierten System die Eigenschaft, daß eine Gefährdung von Menschen durch das System ausgeschlossen ist, verstanden werden (<Lau81>). Man kann in diesem Zusammenhang noch unterscheiden zwischen Systemen, bei denen sich die Gefährdung nur auf Systembediener bezieht (z.B. Produktionsanlagen, Roboter etc.), und solchen, bei denen auch eine Gefährdung Unbeteiligter möglich ist, wie beispielsweise Kernkraftwerke (vgl. <Kap79>).

Für den Begriff Zuverlässigkeit existieren bei technischen Systemen bereits DIN-Normen (DIN 31051, 40041, 40042), die jedoch den Sachverhalt relativ komplex beschreiben. Einfacher ausgedrückt, wird unter Zuverlässigkeit eines Systems im allgemeinen die Eigenschaft verstanden, daß es die gestellten technischen Anforderungen erfüllt, oder "das tut, was es tun soll".

Der Zusammenhang zwischen Sicherheit und Zuverlässigkeit ist nun dadurch gegeben, daß durch eine Verbesserung der Zuverlässigkeit eine Erhöhung der Sicherheit erreicht wird. Anders ausgedrückt heißt dies, daß ein absolut zuverlässiges System (eine nicht erreichbare Idealvorstellung!) auch absolut sicher ist.

Bei Systemen für Echtzeitanwendungen in der Produktion bezieht sich die Forderung nach Zuverlässigkeit auf alle Systemkomponenten, d.h. auf HW, Betriebssystem und Anwendersoftware. Bei der Hardware muß dabei der Ausfall von Komponenten oder der gesamten Rechenanlage vermieden werden. Es handelt sich hier um Fehler, die aufgrund physikalischer oder chemischer Ausfälle auftreten. Die Fehler, die bei der Software auftreten können, sind dagegen auf menschliches Fehlverhalten bei der Erstellung der SW zurückzuführen. Die Konsequenz solcher Fehler sind dann beim Betriebssystem im Extremfall Systemabstürze und bei der Anwendersoftware die Nicht-Erfüllung der spezifizierten Anforderungen.

Die Vermeidung solcher Fehler und damit die Erhöhung der Zuverlässigkeit von HW und SW ist nun eine wesentliche Forderung bei Echtzeitanwendungen in der Produktion. Welche Maßnahmen hierfür ergriffen werden können, soll im folgenden dargelegt werden.

Grundsätzlich lassen sich zwei verschiedene Strategien zur Erhöhung der Zuverlässigkeit von Systemen angeben, die sogenannte Perfektionsstrategie und die Fehlertoleranzstrategie (<Lau81>):

- Bei der Perfektionsstrategie werden Maßnahmen ergriffen, um die Ursachen der Fehler und Ausfälle zu beseitigen, mit dem Ziel, zu einem perfekten System zu kommen.

- Die Fehlertoleranzstrategie toleriert die Tatsache, daß in komplexen technischen Systemen unvermeidlich Fehler auftreten und versucht, deren Auswirkungen z.B. durch Redundanzmaßnahmen zu unterbinden.

Beide Strategien sind für die verschiedenen Anwendungsgebiete Rechenanlagen, Kommunikationswege und Software in unterschiedlicher Weise einsetzbar. Im einzelnen ergeben sich damit eine Vielzahl verschiedener Maßnahmen zur Zuverlässigkeitssteigerung, die nun präsentiert werden sollen.

Rechenanlagen

Hier kommt im wesentlichen die Fehlertoleranzstrategie zum Einsatz, da die durch altersbedingten Ausfall von Einzelkomponenten entstehenden Fehler nicht vermeidbar sind.

An erster Stelle sind hier Redundanzmaßnahmen zu nennen, die in einer Mehrfachauslegung von Komponenten bestehen. Vor allem die kritischen Systemkomponenten wie Zentraleinheit und Plattenlaufwerke werden hier i.a. mehrfach eingesetzt. Man kann dabei zwischen echten Duplexsystemen und Stand-by-Betrieb unterscheiden (siehe Abbildung 2-3).

Bei echten Duplexsystemen werden alle zentralen Komponenten doppelt angeschafft. Beide Prozessoren verarbeiten gleichzeitig die anfallenden Daten der Peripherie und vergleichen ständig die Rechenergebnisse. Beim Ausfall eines Prozesses kann der andere ohne Zeitverzögerung die zu erledigenden Aufgaben alleine lösen.

Duplexsysteme werden vor allem bei sehr kritischen Prozessen verwendet. Sie sind gekennzeichnet durch hohe Anschaffungskosten und durch wesentlich höhere Programmierkosten als bei einem einfachen Rechnersystem. Oftmals be-

gnügt man sich deshalb mit einer doppelten Auslegung des Plattenspeichers, um das schwächste Glied des Gesamtsystems abzusichern. Hier werden dann die Daten einfach auf beiden Platten doppelt gehalten, um bei Ausfall eines Speichers ein weiterhin funktionsfähiges System garantieren zu können.

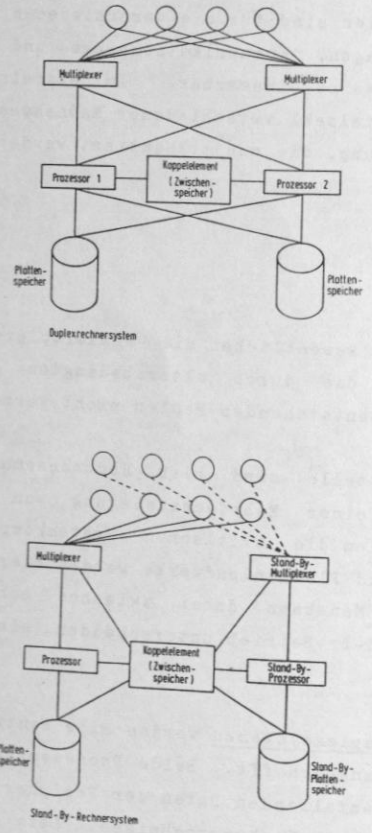


Abb. 2-3: Duplex- und Stand-by-Rechnersysteme

Seit einigen Jahren gibt es auf dem Markt Rechnersysteme, bei denen standardmäßig alle zentralen Komponenten doppelt oder sogar mehrfach ausgelegt sind. Diese Systeme besitzen ein an die spezielle Architektur angepaßtes Betriebssystem, das die erforderlichen Abstimmungen zwischen den mehrfachen Rechnerkomponenten automatisch durchführt, so daß sich die oben erwähnte komplexere und mit höheren Kosten verbundene Programmierung erübrigt.

Wesentlich kostengünstiger als das Duplexverfahren ist der Stand-by-Betrieb. Hier übernimmt der Stand-by-Rechner erst nach Ausfall des Hauptsystems die Steuerung. Unter normalen Bedingungen können deshalb am Stand-by-Rechner andere Aufgaben, wie z.B. Programmentwicklung, durchgeführt werden.

Ein "Mitfahren", d.h. der ständige Austausch von Nachrichten zwischen beiden Systemen ist hier natürlich notwendig. So muß die Stand-by-Maschine den Status der kritischen Programme des anderen Rechners kennen, um so schnell wie möglich die unterbrochenen Programme fortsetzen zu können.

Neben diesen Redundanzmaßnahmen kann bei Rechenanlagen eine Steigerung der Zuverlässigkeit durch Dezentralisierung erreicht werden. Durch Verlagerung von Aufgaben weg vom zentralen Rechner und dezentrale Steuerung bzw. Vorverarbeitung wird zum einen der Rechner entlastet, zum anderen ein wirkungsvoller Schutz bei Rechnerausfall erreicht.

Als Beispiel sei hier die im Abschnitt 2.1 skizzierte Steuerungshierarchie bei flexiblen Fertigungssystemen angeführt. Hier kann durch Zwischenspeicherung eines NC-Programms in den CNC-Steuerungen bei Ausfall des Leitrechners für eine gewisse Zeit ohne Einschränkungen weitergearbeitet werden, so daß hier ein Stand-by-Betrieb auf Leitrechnerebene völlig ausreicht. Eine noch höhere Zuverlässigkeit des Gesamtsystems erreicht man durch Einbeziehung einer weiteren Steuerungsebene für flexible Fertigungszellen. Die

Steuerung für eine flexible Fertigungszelle steuert und überwacht dann die Werkzeugmaschinen der Fertigungszelle und den zelleninternen Transport völlig autonom, so daß hier eine weitgehende Unabhängigkeit vom Leitrechner erreicht wird.

Kommunikationswege

Bei Kommunikationswegen kommen neben Fehlertoleranzmaßnahmen auch vorbeugende Maßnahmen zur Steigerung der Zuverlässigkeit zum Einsatz. Hier ist vor allem der Schutz vor elektrischen Störungen auf den Übertragungsleitungen zu nennen. Durch die Verwendung abgeschirmter, verdrillter Kabel oder die Verwendung von Potentialtrennungsverfahren kann die Übertragungszuverlässigkeit erhöht werden.

Im Gegensatz zu Rechenanlagen kommt als Fehlertoleranzmaßnahme eine Mehrfachauslegung der Kommunikationswege zur Peripherie im allgemeinen aus Kostengründen nicht in Frage. Einzig bei der Rechner-Rechner Kommunikation kann durch eine vermaschte Netzwerk-Konfiguration die Ausfallsicherheit der Übertragungswege erhöht werden.

Als wirkungsvollste Maßnahme zur Zuverlässigkeitssteigerung bieten sich stattdessen Datensicherungsverfahren für die Übertragung an. Hier ist im wesentlichen zwischen zeichenweiser und blockweiser Datensicherung zu unterscheiden.

Bei der zeichenweisen Datensicherung wird jedem zu übertragenden Zeichen ein Parity-Bit hinzugefügt. Mit Hilfe des Parity-Bits wird die Summe der "1"-Bits des Zeichens je nach Vereinbarung auf gerade oder ungerade Parität ergänzt (sog. Querparität). Durch die zeichenweise Sicherung kann von der Empfangsstation ein Übertragungsfehler erkannt werden.

Bei blockweiser Sicherung wird eine sogenannte Längsparitätskontrolle durchgeführt. Hier wird die Sicherungsinformation für einen ganzen Block gebildet und anschließend an diesen übertragen.

Die beiden Verfahren lassen sich auch miteinander kombinieren. Es entsteht dann die Kreuzsicherung, die aus einer Blocksicherung mit Längs- und Querparität besteht.

Die angesprochenen Datensicherungsmaßnahmen für die Übertragung ermöglichen zwar das Erkennen eines Fehlers, aber nicht dessen Korrektur; hierfür werden gesicherte Datenübertragungsprozeduren eingesetzt.

Unter einer Datenübertragungsprozedur versteht man dabei das festgelegte Verhalten des Senders und Empfängers und den zeitlichen Ablauf bei der Datenübertragung. Bei gesicherten Prozeduren wird jeder Block, bei dessen Übertragung ein Fehler auftrat, automatisch wiederholt. Standardmäßig können für eine gesicherte Datenübertragung die sogenannten Basic-Mode-Prozeduren LSV1, LSV2, MSV1, MSV2 oder HDLC-Prozeduren eingesetzt werden (vgl. <Hof78>).

Software

Wesentlichen Anteil an der Zuverlässigkeit des Gesamtsystems hat die Software, da ja hier erst die anwenderspezifischen Erfordernisse realisiert werden. Die Zuverlässigkeit der Hardware ist dabei gewissermaßen als Basis für die Zuverlässigkeit des Gesamtsystems anzusehen. Zuverlässigkeit der Software ist nun sowohl für das Betriebssystem als auch für die Anwenderprogramme zu fordern. Somit ergeben sich für das Gesamtsystem die drei Ebenen Hardware, Betriebssystem und Anwendersoftware, die jeweils aufeinander aufsetzen. Bezogen auf die Eigenschaft der Zuverlässigkeit bedeutet dies, daß eine wesentliche Voraussetzung für den zuverlässigen Ablauf der Anwenderprogramme die Zuverlässig-

keit des Betriebssystems ist, und für die Zuverlässigkeit des Betriebssystems die der zugrundeliegenden Hardware.

Die hier zu diskutierenden Maßnahmen zur Zuverlässigkeitssteigerung von Software sind nun sowohl für das Betriebssystem als auch für Anwenderprogramme einsetzbar. Hierbei sind die beiden grundsätzlichen Strategien - Fehlertoleranz- und Perfektionsstrategie - anwendbar.

Die Fehlertoleranzstrategie bei Software ist unter dem Namen Softwareredundanz bekannt. Im Gegensatz zur Hardware ist hier eine echte Redundanz, d.h. eine Mehrfachverwendung gleicher SW-Komponenten, natürlich nicht sinnvoll. Das Verfahren der "Recovery Blocks" (<Ran75>) basiert stattdessen auf dem Grundgedanken, das Ergebnis eines Programmoduls während der Laufzeit anhand eines Annahmekriteriums zu prüfen und bei Ablehnung eine u.U. mehrfache Neuberechnung nach alternativen Lösungswegen durchzuführen.

Programmsprachlich läßt sich das Recovery Blockkonzept folgendermaßen beschreiben:

```
ENSURE <acceptance test>

    BY <procedure 1>
    .......
    BY <procedure n>

    ELSE error

END
```

Abb. 2-4: Recovery Blockkonzept

Wichtig dabei ist, daß die Prozeduren zur alternativen Berechnung möglichst verschieden konstruiert sind. Problematisch an dieser Methode ist die Tatsache, daß im Annahme-

kriterium selbst wieder Fehler stecken können, so daß eigentlich eine Verifikation dieses Annahmetests notwendig ist. Ein weiterer Nachteil besteht gerade für Echtzeitanforderungen darin, daß durch den Annahmetest und eine eventuelle mehrfache Nachberechnung zusätzliche Programmlaufzeiten entstehen.

Die Perfektionsstrategie ist bei Software durch vorbeugende Maßnahmen bei der Programmerstellung charakterisiert. Ursprünglich waren dies einfach Programmtests zur Fehlererkennung vor der Inbetriebnahme. Bei dieser Vorgehensweise ist es allerdings sehr schwierig, alle möglichen Fälle zu erfassen und somit alle noch vorhandenen Fehler zu entdecken. Ist diese Methode schon bei komplexen Systemen nur sehr begrenzt einsetzbar, so ist sie für parallele Systeme überhaupt nicht geeignet, da hier allein durch Testen nicht alle Fälle reproduziert werden können.

Die ersten Gedanken, die Zuverlässigkeit von Software durch eine übersichtliche Struktur der Programme zu erhöhen, standen unter dem Schlagwort Strukturierte Programmierung bzw. "goto"-lose Programmierung. Durch alleinige Verwendung der Grundbausteine Sequenz, Bedingte Anweisung und Schleife (Abb. 2-5) und die Eliminierung der goto-Anweisung wird die Übersichtlichkeit der Programme wesentlich erhöht.

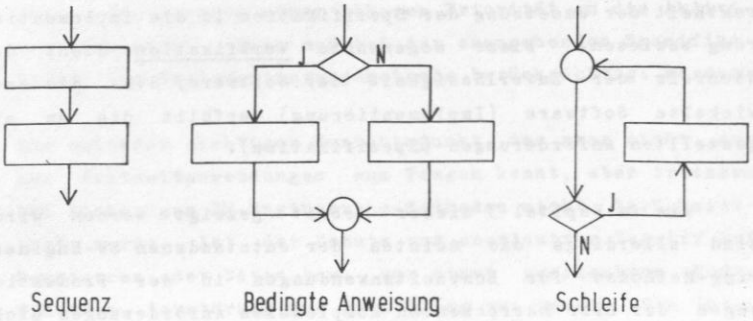


Abb. 2-5: Grundbausteine der Strukturierten Programmierung

Die strukturierte Programmierung ist allerdings ein rein implementierungstechnisches Hilfsmittel und kann keine konzeptionellen Fehler beseitigen. Es besteht außerdem keine Möglichkeit, direkt nachzuprüfen, ob die an das Programm gestellten Anforderungen erfüllt werden, da diese im Rahmen der Methode nicht spezifiziert werden können.

Als Konsequenz daraus stellte sich die Forderung, nicht erst bei der eigentlichen Programmierung, sondern schon beim Entwurf der Programme Hilfsmittel zur Zuverlässigkeitssteigerung einzusetzen. Wie bereits einleitend erwähnt, entstanden die ersten Ansätze hierzu Anfang der 70er Jahre unter dem Begriff "Software-Engineering". Da die in diesem Zusammenhang entstandenen Prinzipien und Methoden in Kapitel 3 dieser Arbeit detailliert vorgestellt werden, soll hier nur ein kurzer Abriß über die prinzipielle Vorgehensweise gegeben werden und im folgenden Abschnitt die besonderen Anforderungen von Echtzeitanwendungen an SW-Engineering-Methoden präsentiert werden.

Prinzipiell liegt den Methoden zur Konstruktion zuverlässiger Software eine Vorgehensweise in drei Schritten zugrunde. Im ersten Schritt, der Spezifikation, werden die an das zu entwickelnde Programm zu stellenden Anforderungen exakt beschrieben. Die Umsetzung dieser Anforderungen in Programmcode geschieht dann im zweiten Schritt, der Implementierung. Schließlich wird im dritten Schritt die Korrektheit der Umsetzung der Spezifikation in die Implementierung bewiesen. Diese sogenannte Verifikation dient dem Nachweis der Zuverlässigkeit der Software, d.h. die entwickelte Software (Implementierung) erfüllt die an sie gestellten Anforderungen (Spezifikation).

Wie im Kapitel 3 dieser Arbeit gezeigt werden wird, sind allerdings die meisten der entstandenen SW-Engineering-Methoden für Echtzeitanwendungen in der Produktion wegen der hier herrschenden komplexeren Anforderungen nicht direkt einsetzbar. Diese besonderen Anforderungen sollen deshalb im folgenden Abschnitt zusammengestellt werden.

2.4 Besondere Anforderungen an SW-Engineering-Methoden beim Einsatz für Echtzeitanwendungen in der Produktion

Die Notwendigkeit des Einsatzes von SW-Engineering-Methoden ist gerade bei Echtzeitanwendungen in der Produktion evident. Die oftmals katastrophalen Auswirkungen von Fehlern erfordern ein hohes Maß an Zuverlässigkeit, das bei der Software nur durch SW-Engineering-Methoden erreichbar ist. Zudem sind wegen der besonderen Komplexität asynchroner Systeme erhöhte Fehlermöglichkeiten und eine schwierigere Fehlererkennung gegeben.

Aus den bisher diskutierten Besonderheiten bei Echtzeitanwendungen ergeben sich allerdings einige zusätzliche Anforderungen an SW-Engineering-Methoden:

- Echtzeitsysteme für die Produktion sind gekennzeichnet durch den asynchronen Ablauf mehrerer Aktivitäten. Eine Synchronisation dieser Aktivitäten, z.B. beim Zugriff auf gemeinsame Ressourcen oder Daten, ist deshalb notwendig. Dies macht eine Berücksichtigung von Synchronisationskonstrukten in Spezifikation und Implementierung erforderlich.

- Bei den parallel ablaufenden Aktivitäten kann zwischen zeitkritischen und zeitunkritischen differenziert werden. Der Vorzug zeitkritischer Aktivitäten wird im allgemeinen durch die Vergabe einer höheren Priorität an die Aktivität realisiert. Dies muß bei der anzugebenden Spezifikations- und Implementierungsmethode berücksichtigt werden.

- Ein weiterer wichtiger Gesichtspunkt, der zwar nicht nur bei Echtzeitanwendungen zum Tragen kommt, aber trotzdem bei bisherigen SW-Engineering-Methoden nicht berücksichtigt wurde, ist der Schutz vor unerlaubtem Zugriff auf Ressourcen oder Daten bzw. vor einer unerlaubten Modifikation, Zerstörung und Enthüllung von Daten. Die Integration von Schutzkonstrukten in das Spezifikations- und Implementierungskonzept ist deshalb notwendig.

Nach der Darstellung der Spezifika von Echtzeitsystemen und insbesondere der daraus resultierenden Anforderungen an SW-Systeme und SW-Engineering-Methoden für Echtzeitanwendungen in der Produktion sollen nun im folgenden Kapitel Prinzipien und Methoden des SW-Engineering als Grundlage für den in dieser Arbeit zu präsentierenden Ansatz diskutiert werden.

3. Die Konstruktion zuverlässiger SW-Systeme

Die Notwendigkeit des Einsatzes zuverlässiger SW-Systeme gerade für die Aufgaben der Fertigungsautomatisierung und Produktionssteuerung wurde im Kapitel 2 dieser Arbeit deutlich dargestellt. Zuverlässigkeit von Software ist jedoch nur zu erreichen durch ein systematisches Vorgehen bei ihrer Entwicklung. Unter den Begriffen "Software-Engineering" bzw. "Softwaretechnologie" entstanden - angestoßen durch die vielzitierte Software-Krise - ab Anfang der 70er Jahre zahlreiche Ansätze und Methoden zur systematischen Entwicklung von SW-Systemen. Der Grund dafür, daß für die Konstruktion von SW-Systemen nicht einfach bewährte Entwicklungsmethoden für andere technische Produkte eingesetzt werden konnten, liegt dabei in der Besonderheit des Produkts "Software".

Deshalb sollen in diesem Kapitel zunächst die Charakteristika von Software präsentiert werden und Qualitätsmerkmale für das Produkt Software entwickelt werden. Dies führt anschließend zu einer Darstellung der einzelnen Entwicklungsphasen im SW-Entstehungsprozeß und einem Überblick über die verschiedenen SW-Entwurfsprinzipien wie z.B. Abstraktion, Modularisierung, Strukturierung oder hierarchischer Systemaufbau und bereits existierende Entwurfsmethoden.

Daran schließt sich dann eine ausführliche Diskussion der Aufgaben der Objektverwaltung, wie der Synchronisation der Zugriffe auf ein Objekt und dem Schutz vor unerlaubten Zugriffen, an. Aufbauend auf diesen Aufgaben und den dargestellten SW-Entwurfsprinzipien werden schließlich die Anforderungen an ein formales Objektverwaltungsmodell als Basis für die Definition der Semantik der zu entwickelnden Spezifikations- und Implementierungsmethode gestellt.

3.1 Charakteristika und Qualitätsmerkmale von Software

Das Produkt "Software" unterscheidet sich von anderen herkömmlichen technischen Produkten prinzipiell in einigen wesentlichen Punkten (vgl. <Gew79> oder <Bal82>).

- Software ist ein abstraktes, immaterielles Produkt, das nicht altert, keinem Verschleiß unterliegt, und deshalb auch keine Wartung im üblichen Sinn und keine Ersatzteile benötigt.

- Software ist kein Serienprodukt, sondern vergleichbar mit einer Einzelanfertigung. Bei der Erstellung von Software handelt es sich deshalb nicht um einen Produktionsprozeß, sondern das eigentliche Problem ist die Produktentwicklung. Die Herstellung von Kopien - die eigentliche Produktion - ist dagegen eine triviale Angelegenheit. Die Entwicklung von Software ist somit ähnlich dem Schreiben eines Buches oder auch der Prototypentwicklung in der Automobilindustrie.

- Umfangreichere SW-Systeme besitzen einen sehr hohen Komplexitätsgrad. Die geistigen Fähigkeiten des Menschen zur Lösung komplexer, umfangreicher Probleme sind jedoch äußerst beschränkt; insbesondere, weil der Mensch nur über wenig umfangreiche Probleme intensiv nachdenken kann und sich nur eine begrenzte Anzahl von Begriffen gleichzeitig vergegenwärtigen kann. Gerade bei nichtsequentiellen Systemabläufen macht sich zusätzlich die rein sequentielle Denkweise des Menschen negativ bemerkbar.

  Um trotz dieser Voraussetzungen komplexe Systeme entwickeln zu können, müssen bei der Konstruktion Prinzipien und Methoden eingesetzt werden, die eine Reduktion der Komplexität ermöglichen. Als wichtigste Prinzipien - auf die in diesem Kapitel noch ausführlich eingegangen werden soll - haben sich hierfür das Abstraktionsprinzip und die Prinzipien der Strukturierung und der Lokalität herausgebildet.

- Software ist leichter und schneller änderbar als übliche technische Produkte. Diese Eigenschaft birgt besonders in der Testphase und bei Änderungen große Gefahren in sich. Es ist eine bekannte Tatsache, daß durch voreilige und unüberlegte Änderungen meistens mehr neue Fehler in Programme eingebaut werden, als alte dadurch eliminiert werden.

Diese spezifischen Eigenschaften von Software machen - bei den beschränkten geistigen Fähigkeiten des Menschen - eine systematische Vorgehensweise und die Entwicklung produktangepaßter Methoden für die SW-Erstellung erforderlich, um gute, überschaubare und vor allem zuverlässige SW-Produkte zu erhalten.

Für die Qualitätsmerkmale von Software gibt es in der Literatur keine einheitlichen Definitionen; zudem sind die existierenden Merkmale zumeist nur schwer quantifizierbar. Hinzu kommt noch, daß die Qualitätsanforderungen an Software, abhängig vom Einzelfall, durchaus unterschiedlich sein können, so daß sich die Frage, wann ein SW-Produkt qualitativ hochrangig ist, oft nicht trivial beantworten läßt. Jedoch haben sich einige allgemeine Eigenschaften herausgebildet, die im folgenden kurz vorgestellt werden sollen.

Die wohl wichtigste Qualitätsanforderung an SW-Systeme ist deren Zuverlässigkeit. Zuverlässigkeit von Software ist eine unabdingbare Voraussetzung für ihre Brauchbarkeit. Man versteht unter einem zuverlässigen SW-System ein System, das "genau das tut, was es tun soll". Hierzu ist es notwendig, die Anforderungen an das System zu spezifizieren. Der Nachweis der Zuverlässigkeit des Systems besteht dann darin, zu zeigen, daß das System die spezifizierten Anforderungen erfüllt. In den folgenden Abschnitten soll hierauf noch detailliert eingegangen werden.

Neben der wichtigen Eigenschaft der Zuverlässigkeit spielt die Forderung nach Verständlichkeit und Überschaubarkeit eine entscheidende Rolle bei der Beurteilung der Qualität eines SW-Produkts. Hierzu trägt vor allem die konsequente Anwendung von Strukturierungs-, Lokalitäts- und Abstraktionsprinzip bei. Durch die Reduktion der Komplexität werden SW-Systeme leichter überschaubar und damit letztendlich auch zuverlässiger bzw. leichter verifizierbar.

Eng damit im Zusammenhang steht die Testbarkeit und Modifizierbarkeit eines Systems. Diese für den praktischen Einsatz sehr wichtigen Eigenschaften werden ebenfalls stark durch einen strukturierten Systementwurf und die Anwendung von Abstraktions- und Lokalitätsprinzip beeinflußt.

Zusätzlich zu diesen grundlegenden Qualitätsmerkmalen von Software lassen sich noch Anforderungen aufführen, die weniger die innere Struktur und Zuverlässigkeit der Software, sondern mehr deren Anwendbarkeit betreffen. Hier wären z.B. die Benutzerfreundlichkeit oder die Portabilität der Software zu nennen. Auch die Effizienz von SW-Systemen, d.h. die optimale Ausnutzung der Betriebsmittel wird oft als Qualitätsmerkmal definiert. Dies birgt jedoch die Gefahr in sich, daß durch allzu große Optimierungen die Verständlichkeit und auch Zuverlässigkeit eines Systems stark abnimmt.

In den folgenden Abschnitten dieses Kapitels sollen nun überblicksartig die wichtigsten Prinzipien zur Erstellung qualitativ hochwertiger zuverlässiger Software vorgestellt werden, und daraus wesentliche Aufgabenstellungen der vorliegenden Arbeit abgeleitet werden.

3.2 Phasen der SW-Entwicklung

Die während einer Software-Entwicklung anfallenden Tätigkeiten können in verschiedene Klassen - die sogenannten Entwicklungs-Phasen - eingeteilt werden. Man unterscheidet im allgemeinen zwischen Planungs-, Definitions-, Entwurfs-, Implementierungs- sowie Abnahme- und Einführungsphase.

In der Planungsphase wird das entsprechende SW-Produkt geplant und kalkuliert. Die hier anfallenden Tätigkeiten beinhalten z.B. die Auswahl des Produkts, Voruntersuchungen, Durchführbarkeitsuntersuchungen und Wirtschaftlichkeitsbetrachtungen. Die zu lösenden Aufgaben sind vorwiegend im Marketingbereich angesiedelt.

Aufgabe der Definitionsphase ist das Zusammenstellen der Anforderungen an das Produkt. Das Ziel ist dabei, diese Anforderungen in einem konsistenten, vollständigen Anforderungskatalog - Pflichtenheft genannt - niederzulegen. Hierbei wird als Abgrenzung zur Entwurfsphase das zu entwickelnde Produkt aus der Sicht des Anwenders beschrieben, d.h. das Pflichtenheft ist noch keine softwaretechnische Lösung.

In der Entwurfsphase wird dann aus den gegebenen Anforderungen ein softwaretechnischer Entwurf im Sinne einer Systemstruktur entwickelt. Aufgaben des Entwurfs sind z.B. eine Aufteilung des Gesamtproblems auf Systemkomponenten, deren hierarchische Anordnung, die Spezifikation des Leistungsumfangs der Systemkomponenten und die Festlegung der Schnittstellen und Kommunikation zwischen den Systemkomponenten.

In der Implementierungsphase wird das System aufgrund der Spezifikation der einzelnen Systemkomponenten programmiert. Hier werden die durch die Spezifikation gegebenen abstrakten Datenstrukturen und Funktionen in konkrete Datenstrukturen und Algorithmen umgesetzt.

In der Abnahme- und Einführungsphase wird das System vom Anwender abgenommen und schließlich beim Anwender zum Einsatz gebracht. Im allgemeinen wird dabei ein Abnahmetest durchgeführt, dessen Ergebnisse in einem Abnahmeprotokoll festgehalten werden. Anschließend wird dann das System beim Anwender installiert und in Betrieb genommen. Mit dieser Phase ist die eigentliche SW-Entwicklung beendet. Das SW-Produkt wird dann von einem Wartungs- und Pflegesystem verwaltet, wobei z.B. unterschiedliche Versionen verwaltet werden müssen oder Fehlerkorrekturen durchgeführt werden.

Für die Qualität eines SW-Produkts sind - bei gegebener Anforderungsdefinition - somit die Entwurfs- und Implemementierungsphase von entscheidender Bedeutung. Häufig wurde und wird bei der Entwicklung von SW-Systemen nicht deutlich genug zwischen diesen beiden Phasen getrennt, oder die Entwurfsphase wird sogar völlig weggelassen. Eine solche Vorgehensweise widerspricht allerdings nicht nur modernen softwaretechnologischen Erkenntnissen, sondern ist der beste Weg zu einer unzuverlässigen und unüberschaubaren Programmierung. Vergleichbar wäre dies etwa mit dem Bau eines Hauses ohne Bauplan oder der Konstruktion einer Maschine ohne Konstruktionszeichnung.

So muß die Trennung zwischen Entwurf und Implementierung als wesentliche Voraussetzung für die Erstellung zuverlässiger SW-Systeme angesehen werden. Da gerade in der Entwurfsphase die Systemstruktur festgelegt wird, muß auf einen soliden Systementwurf größter Wert gelegt werden. Ziel des Systementwurfs ist dann die Zergliederung des Gesamtsystems in Einzelmoduln und die exakte und vollständige Spezifikation der einzelnen Moduln mittels einer Spezifikationssprache (auf die hierbei anzuwendenden Prinzipien wird in 3.3 noch eingegangen). Bereits in dieser Phase können gewisse invariante Systemeigenschaften nachgewiesen werden. Die Umsetzung der in der Entwurfsphase spezifizierten abstrakten Objekte und Operationen in konkrete Datenstrukturen und Algorithmen erfolgt dann in der Implementierungsphase.

Meistens ist zwischen der gegebenen Spezifikation und der durchzuführenden Implementierung eine so große Lücke, daß es sinnvoll ist, die Umsetzung in mehreren Schritten vorzunehmen. Jede Zwischenstufe definiert dann eine sogenannte Abstraktionsebene bzw. abstrakte Maschine. Der Zusammenhang zwischen den einzelnen Abstraktionsebenen wird dadurch hergestellt, daß die Implementierungen einer Ebene die Spezifikationen der nächstniedrigen Ebene benutzen, indem sie die entsprechenden Operationen der untergeordneten Ebene aufrufen. Bei der Erläuterung des Abstraktionsprinzips in 3.3.1 soll hierauf noch näher eingegangen werden.

Voraussetzung für die Zuverlässigkeit eines SW-Systems ist natürlich, daß die Spezifikationen durch die Implementierung korrekt umgesetzt werden. Der Nachweis einer korrekten Implementierung ist Sache der Verifikation und wird in 3.3.6 näher dargestellt.

Zusammenfassend läßt sich sagen, daß die Trennung zwischen Entwurf und Implementierung und der Entwurf eines Systems in mehreren Abstraktionsebenen eine wesentliche Voraussetzung für die Überschaubarkeit und Zuverlässigkeit eines SW-Systems ist. Besonderes Gewicht ist dabei auf einen präzisen und vollständigen SW-Entwurf zu legen. Das Hauptgewicht der Arbeit liegt deshalb auf der Entwicklung und Bereitstellung von Methoden für den Entwurf und die korrekte Implementierung von Software. Im Zuge des SW-Engineerings haben sich hierfür einige Prinzipien herausgebildet, die nun im folgenden Abschnitt dargestellt werden sollen.

3.3 SW-Entwurfsprinzipien

3.3.1 Abstraktion

Der Vorgang der Abstraktion ist als die wohl wesentlichste Voraussetzung für den Entwurf zuverlässiger und überschaubarer SW-Systeme anzusehen. Insbesondere ist das Abstraktionsprinzip grundlegend für alle anderen Entwurfsprinzipien, so daß sich viele der dort geforderten Systemeigenschaften schon durch eine konsequente Anwendung von Abstraktionsmethoden - wie z.B. das hier zu erläuternde Konzept der Abstrakten Datentypen - ergeben.

Unter Abstraktion versteht man die Konzentration auf das Wesentliche und die Vernachlässigung unwichtiger Details. Die so durchgeführte Reduktion der Komplexität ist die einzige Möglichkeit, umfangreiche Systeme zu überschauen, da man sich jeweils nur mit den momentan zur Lösung eines Teilproblems relevanten Systemaspekten beschäftigt. Die grundlegende Natur des Abstraktionsprozesses zeigt sich auch darin, daß Abstraktion in allen Lebensbereichen vorkommt, und als eines der wichtigsten Merkmale von Intelligenz und kreativem Denken angesehen werden muß.

In der SW-Technologie wird das Abstraktionsprinzip beim Entwurf eines Systems in mehreren Schritten angewandt. Es entsteht auf diese Weise eine Folge von Verfeinerungen des Systems, die jeweils das gesamte System beschreiben. Man nennt diese einzelnen Stufen Abstraktionsebenen oder virtuelle bzw. abstrakte Maschinen (vgl. Abbildung 3-1).

Jede Abstraktionsebene besteht aus der Spezifikation der Ebene und der Implementierung, wobei der Begriff "Implementierung" nicht automatisch präjudiziert, daß hierfür eine konkrete Maschine existiert, auf der die Implementierung abläuft; dies gilt nur für die unterste Ebene.

Die Verbindung zwischen den einzelnen Ebenen wird dadurch hergestellt, daß die Implementierung der i-ten Ebene die Spezifikationen der i+1-ten Ebene benutzt, indem sie die entsprechenden Operationen der Ebene i+1 aufruft.

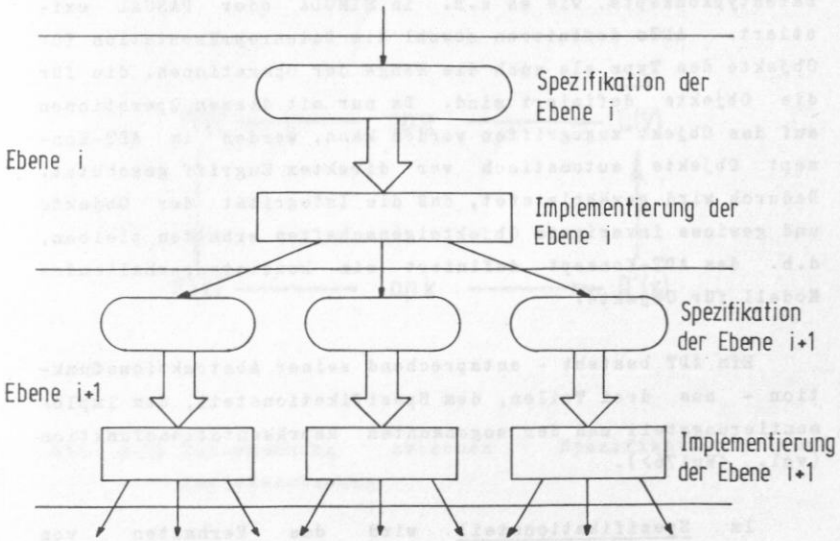


Abb. 3-1: Darstellung der Abstraktionshierarchie

Prinzipiell lassen sich beim SW-Entwurf zwei verschiedene Abstraktionsarten unterscheiden, die prozedurale Abstraktion und die Datenabstraktion. Bei der prozeduralen Abstraktion - die schon lange bekannt ist - wird durch die Definition von Prozeduren vom ausführenden Algorithmus abstrahiert. Die Datenabstraktion basiert auf dem Konzept der Abstrakten Datentypen (ADT) (<Lis74> bzw. <Lis75>). Hierdurch ist zusätzlich eine Abstraktion von der konkreten Realisierung von Objekten möglich.

Wodurch ist nun ein Abstrakter Datentyp definiert? Ein ADT ist eine Erweiterung und Modifikation des bekannten Datentypkonzepts, wie es z.B. in SIMULA oder PASCAL existiert. ADTs definieren sowohl die Datenrepräsentation für Objekte des Typs als auch die Menge der Operationen, die für die Objekte definiert sind. Da nur mit diesen Operationen auf das Objekt zugegriffen werden kann, werden im ADT-Konzept Objekte automatisch vor direktem Zugriff geschützt. Dadurch wird gewährleistet, daß die Integrität der Objekte und gewisse invariante Objekteigenschaften erhalten bleiben, d.h. das ADT-Konzept definiert ein konsistenzerhaltendes Modell für Objekte.

Ein ADT besteht - entsprechend seiner Abstraktionsfunktion - aus drei Teilen, dem Spezifikationsteil, dem Implementierungsteil und der sogenannten Repräsentationsfunktion (vgl. <Wul76>).

Im Spezifikationsteil wird das Verhalten von ADT-Objekten für den Benutzer beschrieben, wobei von für den Benutzer unwichtigen Implementierungsdetails abstrahiert wird. Der Spezifikationsteil enthält die Abstrakte Repräsentation für Objekte des Typs und eine Beschreibung der Operationen durch ihre Wirkung auf die Abstrakte Repräsentation. Diese Beschreibung ist durch Preconditions, Postconditions und invariante Eigenschaften in der von Hoare <Hoa69> eingeführten Weise gegeben.

Der Implementierungsteil enthält die konkrete programmtechnische Realisierung der Abstrakten Repräsentation und der Operationen. Hierbei kann auf Spezifikationen der nächstniedrigeren Abstraktionsebene aufgebaut werden. Die Beziehung zwischen abstrakten Objekten und konkreten Objekten wird durch die Repräsentationsfunktion festgelegt, die konkreten Objekten abstrakte Objekte zuordnet.

Die folgende Skizze soll den Zusammenhang weiter verdeutlichen:

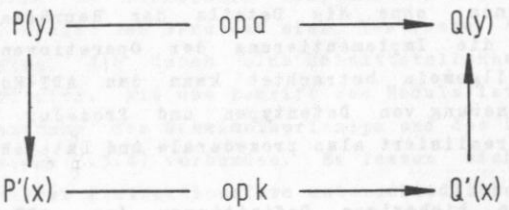


Abb. 3-2: Zusammenhang zwischen Spezifikation und Implementierung

op a ist hier eine nach außen sichtbare, abstrakte Operation eines ADTs und y eine Variable, deren Wertebereich die Abstrakte Repräsentation des Typs ist. Die Operation op a sei charakterisiert durch P(y) {op a} Q(y), d.h. P(y) ist Precondition und Q(y) Postcondition für op a. Die Operation op a wird hier durch das konkrete Programm op k realisiert, das die Implementierungsvariable x manipuliert. Für op k gelte P`(x) {op k} Q`(x). Die Repräsentationsfunktion ist dabei eine i.a. surjektive, aber nicht vollständige Funktion des Wertebereichs von x auf den Wertebereich von y.

Das nach außen sichtbare Verhalten der Operation op a wird also im Spezifikationsteil durch P(y) {op a} Q(y) charakterisiert, während das interne Verhalten durch P`(x) {op k} Q`(x) gegeben ist. Das o.a. Schema kann direkt für die Verifikation der Korrektheit der Implementierung benutzt werden, worauf in 3.3.6 und Kapitel 6 noch näher eingegangen werden soll.

Abstrakte Datentypen stellen eine Basis für die Datenabstraktion zur Verfügung, die genauso mächtig ist wie das Prozedurkonzept für die prozedurale Abstraktion. Wie eine Prozedur eine abstrakte Operation definiert, die ohne Kenntnis der Implementierungsdetails der Operation aufgerufen werden kann, definiert ein ADT abstrakte Objekte, die durch einen Aufruf der Operationen auf diese Objekte manipuliert werden können, ohne die Details der Repräsentation der Objekte und die Implementierung der Operationen zu verstehen. Allgemein betrachtet kann das ADT-Konzept als Verallgemeinerung von Datentypen und Prozedur betrachtet werden, es realisiert also prozedurale und Datenabstraktion.

In den bisherigen Definitionen des ADT-Konstrukts werden weder Aspekte des Datenschutzes noch asynchrone Zugriffe auf abstrakte Objekte berücksichtigt. Ein wichtiger Gesichtspunkt dieser Arbeit ist deshalb - im Sinne eines vollständigen Ansatzes - die Integration dieser Aspekte in das allgemeine ADT-Konzept, wobei auf Vorarbeiten von Keramidis <Ker82> aufgesetzt wird.

3.3.2 Geheimnisprinzip und Modularisierung

Beide Prinzipien wurden von Parnas (<Par72a> bzw. <Par72b>) entwickelt und stehen in engem Zusammenhang miteinander bzw. bauen aufeinander auf.

Das Geheimnisprinzip wurde ursprünglich für prozedurale Abstraktionen definiert, ist aber völlig analog für Datenabstraktionen anwendbar. Es besagt, daß dem Benutzer einer Abstraktionsform alle implementierungstechnischen Details verborgen sind. Bei ADTs bedeutet dies, daß der Benutzer nur mit den in der Spezifikation definierten Operationen zugreifen darf, und daß der Implementierungsteil für den Benutzer unsichtbar bleibt. Während die erste Forderung von ADTs direkt erfüllt wird, ist die zweite Eigenschaft nicht notwendigerweise durch die ADT-Definition gegeben. Jedoch verbindet man heute mit dem Begriff der Abstrakten Datentypen automatisch die Erfüllung des Geheimnisprinzips.

Das Prinzip der Modularisierung bedeutet die Zerlegung eines Systems in einfachere Komponenten, Module genannt. Jeder Modul stellt dem Benutzer eine bestimmte Abstraktion zur Verfügung, die durch eine Schnittstellenbeschreibung spezifiziert wird. Mit dem Begriff des Moduls ist außerdem die Realisierung des Geheimnisprinzips und des Lokalitätsprinzips (siehe 3.3.4) verbunden. Es lassen sich je nach bereitgestellter Abstraktionsform unterschiedliche Arten von Moduln unterscheiden, die sich jedoch alle aus einer allgemeinen Modulform ableiten lassen. Diese Form entspricht genau der Definition des Abstrakten Datentyps, so daß durch die Verwendung von ADTs automatisch das Prinzip der Modularisierung erfüllt wird. Hierdurch wird auch gleichzeitig eine sinnvolle Zerlegung des Gesamtsystems gewährleistet.

3.3.3 Hierarchiebildung und Strukturierung

Viele in der Natur vorkommende komplexe Systeme sind hierarchisch aufgebaut. Hierarchische Anordnungen sind dadurch charakterisiert, daß sich die Beziehungen der einzelnen Komponenten des Systems durch einen endlichen, gerichteten, zyklenfreien Graphen darstellen lassen. Je nach weiteren Restriktionen für die Form des Graphen unterscheidet man netzwerkorientierte, baumartige oder schicht-

orientierte Hierarchien (<Bal82>).

Entwirft man bei der SW-Erstellung sein System in mehreren Abstraktionsebenen, so ergibt sich automatisch eine schichtorientierte Hierarchie, wenn man die einzelnen ADTs betrachtet und als Relation zur Hierarchiebildung die "Implementiert"-Relation verwendet. Hierarchische Anordnungen tragen wesentlich zur Überschaubarkeit und Verifizierbarkeit eines Systems bei, da sie - vor allem in Verbindung mit einer Anwendung des Abstraktionsprinzips - eine erhebliche Reduzierung der Komplexität eines Problems bewirken.

Die Strukturierung eines Systems in der Entwurfsphase ist zu unterscheiden von der vielzitierten "strukturierten Programmierung", die sich auf die Implementierung bezieht. Unter Strukturierung versteht man nämlich eine Überlagerung des Prinzips der Modularisierung mit dem Prinzip der Hierarchiebildung. Als eine sehr sinnvolle Strukturierung eines Systems kann somit die Zerlegung in ADTs als Module zusammen mit der Herstellung einer "implementiert"-Hierarchie zwischen den einzelnen Abstraktionsebenen angesehen werden.

3.3.4 Lokalität und Modifizierbarkeit

Für die Überschaubarkeit eines Systems ist die Einhaltung des Lokalitätsprinzips beim Systementwurf von besonderer Bedeutung. Nach diesem Prinzip sollen sich alle Stellen, von denen aus ein Objekt manipuliert werden kann, und alle Kontrollinformationen, die diese Manipulation regeln, in örtlicher Nähe des Objekts selbst befinden.

In vielen existierenden Programmiersprachen gibt es Konstrukte, die eine eklatante Verletzung des Lokalitätsprinzips ermöglichen; dies zeigt, daß die Bedeutung des Lokalitätsprinzips als Strukturierungsmaßnahme noch wenig erkannt wurde. Andererseits wird durch die Definition von Moduln in einem System, und insbesondere durch die Einfüh-

rung Abstrakter Datentypen, das Lokalitätsprinzip stark unterstützt. Ein ADT definiert neben einer Datenrepräsentation für Objekte des Typs die Menge der Operationen, mit denen auf das Objekt zugegriffen werden kann. Zusätzlich muß im ADT noch angegeben werden, wann mit den Operationen auf Objekte des Typs zugegriffen werden darf. Das konventionelle ADT-Konzept erfüllt diesen letzten Punkt nur teilweise (vgl. 3.3.1), da in ihm Datenschutzaspekte und asynchrone Zugriffe auf ADT-Objekte noch nicht berücksichtigt sind. Wie bereits erwähnt, sollen in dieser Arbeit die entsprechenden Erweiterungen vorgenommen werden.

Die große Bedeutung des Lokalitätsprinzips zeigt sich auch darin, daß seine Einhaltung eine der wichtigsten Voraussetzungen für die Modifizierbarkeit eines Systems ist. Auch hier bietet die Verwendung von Moduln in Form von ADTs enorme Vorteile, da z.B. die Realisierung eines Moduls ohne Einfluß auf das übrige System modifiziert werden kann, wenn die Schnittstellenspezifikation beibehalten wird. Natürlich wird auch bei einer Änderung der Schnittstellenspezifikation eine Einhaltung des Lokalitätsprinzips von Vorteil sein, da die Modifikationen nur an einer Stelle erfolgen müssen, und so viele der bei SW-Änderungen üblichen Fehler von vorneherein vermieden werden.

3.3.5 Spezifikation

Bereits bei der Erläuterung des Abstraktionsprinzips wurde die Spezifikation als Mittel zur Definition des Leistungsumfangs und der externen Benutzerschnittstellen von ADTs bzw. Moduln eingeführt. In diesem Abschnitt soll nun näher auf die Aufgaben der Spezifikation und verschiedene Spezifikationsmethoden eingegangen werden. Eine vollständige und genaue Spezifikation dessen, "was ein Modul tut bzw. tun soll" ist aus drei Gründen wichtig:

- In der SW-Entwurfsphase dient die Spezifikation zur Festlegung des zu lösenden Problems unabhängig vom geplanten oder einzuschlagenden Lösungsweg. Bei größeren Projekten mit mehreren Mitarbeitern dient die Spezifikation in dieser Phase auch als Kommunikationsinstrument zwischen den einzelnen Beteiligten.

- War die Spezifikation in der Entwurfsphase als Vorgabe zu sehen, so definiert sie beim späteren Gebrauch der implementierten Moduln genau deren Leistungsumfang. Hierunter fällt auch eine exakte Festlegung der Benutzerschnittstellen, man spricht deshalb auch von Schnittstellenspezifikation.

- Für die Verifikation ist die Spezifikation ebenfalls Grundlage. Der Beweis, daß ein gegebener Lösungsalgorithmus die korrekte Lösung eines Problems darstellt, erfordert eine exakte Problemstellung in Form einer Spezifikation.

Aus diesen verschiedenen Verwendungen der Spezifikation ergibt sich die Forderung nach deren Eindeutigkeit, Vollständigkeit, Konsistenz und leichten Verifizierbarkeit. Diese Forderung kann nur durch den Einsatz formaler Spezifikationsmethoden erfüllt werden. Sieht man von der Methode der denotationalen Semantik (<Ten76>), die sich für die Spezifikation von ADTs weniger eignet, ab, so kann man im wesentlichen drei Methoden unterscheiden

- Die Methode der Algebraischen Spezifikation (<Gut76>, <Gut77>, <Gut78>, <Gog76>)

- Das Zustandsmaschinenmodell von Parnas (<Par72a>)

- Die Methode der Prädikatentransformation (<Hoa69>, <Wul76>)

Es soll hier nur sehr kurz auf diese Ansätze eingegangen werden; eine detaillierte Betrachtung findet sich z.B. in <Bal82> oder <Ker82>.

Bei der algebraischen Spezifikation werden ADTs durch die Angabe von Definitions- und Wertebereich der Operationen des ADTs und sogenannter Axiome, die die Operationen durch ihre Beziehungen zu den anderen Operationen beschreiben, definiert. Diese Methode ist zwar theoretisch elegant, jedoch für die Praxis weniger geeignet, da eine Spezifikation mit ihr nicht der Intuition naheliegt. Außerdem ist ihre Anwendung sehr komplex und schwer auf Konsistenz und Vollständigkeit zu prüfen.

Das Zustandsmaschinenmodell von Parnas - oder auch Methode der O- und V-Operationen genannt - teilt die Operationen eines ADTs in zwei Klassen ein. Die sogenannten V-Operationen (view) stellen die Werte des Objekts zur Verfügung, während die O-Operationen (operate) die eigentlichen Zustandsübergänge im Modul durchführen. Charakteristisch für diese Methode ist - im Gegensatz zur Methode der Prädikatentransformation - daß keine Angabe einer abstrakten Repräsentation für Objekte erforderlich ist. Dies führt allerdings zu sehr umfangreichen Spezifikationen, die schwer änderbar und erweiterbar sind.

Die Methode der Prädikatentransformation, die auch die Basis für diese Arbeit bildet, gibt die Operationen auf abstrakte Objekte in der Form $P(y) \{ op \} Q(y)$ an. Eine Operation kann nur ausgeführt werden, wenn $P(y)$ gilt, nach der Ausführung gilt dann $Q(y)$. $P(y)$ und $Q(y)$ sind dabei Prädikate über die Abstrakte Repräsentation y, so daß eine Angabe der Objektrepräsentation hier erforderlich ist.

Die mit dieser Methode spezifizierten ADTs enthalten darum in ihrem Spezifikationsteil den sogenannten DECLARATIONS-Teil, der die Abstrakte Repräsentation für Objekte, E/A-Parameter und Hilfsgrößen beinhaltet, und den OPERATIONS-Teil, der die Operationen in der o.a. Form

angibt.

In <Ker82> wird der Spezifikationsteil noch um den SYN-Teil erweitert, der die Synchronisation der Operationen auf das abstrakte Objekt spezifiziert. Eine Verträglichkeitsrelation VTGL gibt an, wann zwei Operationen gleichzeitig ausgeführt werden dürfen, und das PRIO-Konstrukt definiert die Prioritätsverhältnisse zwischen den Operationen. Insgesamt stellt sich damit der Spezifikationsteil folgendermaßen dar:

```
SPECIFICATION

  DECLARATIONS
     ABSTRACT REPRESENTATION
     PARAMETERS

  SYN
     VTGL
     PRIO

  OPERATIONS
```

In dieser Arbeit soll nun dieses Schema noch um die Spezifikation von Sicherheitspolicies und Datenschutzmechanismen erweitert werden.

Speziell für die Spezifikation von Moduln wurden bereits einige Spezifikationssprachen entwickelt. Es handelt sich hierbei um nichtprozedurale Sprachen wie z.B. SPEZI (<Flo78>) oder SPECIAL (<Rob77>). In ALPHARD (<Wul76>) wurde sogar der Versuch unternommen, Spezifikation und Implementierung in einem einheitlichen Ansatz zu präsentieren. Es ergibt sich so bei einer Moduldefinition eine Dreiteilung in Spezifikation, Implementierung und Repräsentationsfunktion. Dies entspricht aber genau dem vorgestellten Aufbau von ADTs. In dieser Arbeit soll deshalb eine

ähnliche Vorgehensweise, wie sie in ALPHARD definiert wurde, eingeschlagen werden - allerdings mit den bereits erwähnten notwendigen Erweiterungen.

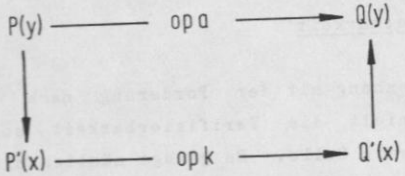
3.3.6 Verifizierbarkeit

Im Zusammenhang mit der Forderung nach zuverlässigen SW-Systemen spielt die Verifizierbarkeit solcher Systeme eine entscheidende Rolle. Es genügt nämlich nicht für ein System, zuverlässig zu sein, sondern man muß diese Zuverlässigkeit auch beweisen können. An dieser Stelle sei bereits erwähnt, daß ein absoluter Nachweis der Zuverlässigkeit eines Systems nicht existiert, da hierzu wiederum die Korrektheit des Zuverlässigkeitsbeweises bewiesen werden müßte, usw. usf. Trotzdem muß die Verifikation als ein wichtiges Hilfsmittel zum Erreichen zuverlässiger SW-Systeme angesehen werden, da die Bemühungen, ein verifizierbares System zu entwickeln, per se die Zuverlässigkeit des Systems erhöhen und die formale oder halbformale Verifikation zusätzliche Sicherheit bietet.

Setzt man beim Entwurf seines Systems die bisher aufgeführten Prinzipien ein, insbesondere das ADT-Konzept, so gibt es zwei Ansätze für eine Verifikation.

- Bereits für den Entwurf des Systems, d.h. für dessen Spezifikation, können gewisse invariante Systemeigenschaften bzw. Zusicherungen nachgewiesen werden, ohne daß hierzu die konkrete Implementierung bekannt sein muß. Diese mehr entwurfsorientierte Verifikation wird in der englischsprachigen Literatur "design verification" genannt. Beispiele hierfür sind z.B. der Nachweis, daß eine Spezifikation eine gewisse Sicherheitspolicy erfüllt oder daß die angegebenen Verträglichkeitsbedingungen "konsistent" sind (vgl. hierzu auch Kapitel 5/6)

- Der andere Verifikationsansatz besteht darin, nachzuweisen, daß die angegebene Implementierung die Spezifikation erfüllt. Hierbei spielt die Repräsentationsfunktion eine wichtige Rolle. Betrachtet man nochmals das Diagramm



so implementiert op k die abstrakte Operation op a korrekt, wenn gilt

$$\forall_x \forall_y (rep(x)=y)\ (P(y) \dashrightarrow P'(x) \wedge Q'(x) \dashrightarrow Q(y))$$

Näheres hierzu findet sich in Kapitel 6 dieser Arbeit.

An dieser Stelle muß noch bemerkt werden, daß sich gerade bei größeren SW-Projekten eine exakte Verifikation wegen der großen Komplexität und dem damit verbundenen Zeitaufwand selten durchführen läßt. Man setzt deshalb oftmals sogenannte "ingenieurmäßige" Verifikationsmethoden ein, bei denen nur einige kritische Stellen des Systems exakt verifiziert werden. Hier wurden bereits mit konkreten Industrieprojekten (<Ker79> und <Hof79>) gute Erfahrungen gesammelt. Insgesamt wird sich eine Verifikation größerer SW-Systeme jedoch wohl erst durch den Einsatz automatischer oder halbautomatischer, interaktiver Verifikationssysteme rationell bewerkstelligen lassen.

3.3.7 Zusammenfassung

Bei der Entwicklung umfangreicher und komplexer SW-Systeme kommt der Entwurfsphase eine besondere Bedeutung zu, da hier die Struktur und der Aufbau des Systems festgelegt werden. Zur Erreichung zuverlässiger und verständlicher Systeme ist es deshalb gerade in dieser Phase unabdingbar notwendig, durch geeignete Maßnahmen die Komplexität des zu lösenden Problems zu reduzieren. Hierfür wurden im Laufe dieses Abschnitts verschiedene Entwurfsprinzipien und als Inkarnation dieser Prinzipien das Konzept der Abstrakten Datentypen vorgestellt. An dieser Stelle soll nochmals zusammenfassend gezeigt werden, wie das ADT-Konzept diese verschiedenen Prinzipien erfüllt und somit als Basiskonstrukt für die Entwicklung zuverlässiger, verständlicher SW-Systeme verstanden werden muß.

- ADTs stellen dem Designer eines Systems eine Basis für prozedurale und Datenabstraktion zur Verfügung und entlasten somit von unnötigen implementierungstechnischen Details niedrigerer Abstraktionsebenen.

- Das ADT-Konzept definiert ein konsistenzerhaltendes Modell für Objekte, da durch die Verhinderung von direkten Zugriffen (mit nicht für das Objekt definierten Operationen) auf abstrakte Objekte deren Integrität und die Erhaltung invarianter Objekteigenschaften garantiert werden kann.

- ADTs unterstützen das Geheimnisprinzip, da der Benutzer eines abstrakten Objekts nur dessen Spezifikation, nicht aber Implementierungsdetails sieht.

- ADTs können als allgemeinste Form für Module angesehen werden, so daß das Prinzip der Modularisierung beim SW-Entwurf mit ADTs automatisch erfüllt wird.

- Das Lokalitätsprinzip wird durch das ADT-Konzept ebenfalls stark unterstützt. Alle Stellen, von denen aus ein

Objekt manipuliert werden kann, sowie alle Kontrollinformationen, sind bei ADTs in unmittelbarer örtlicher Nähe des Objekts selbst. Hierdurch erhöht sich auch die Modifizierbarkeit eines Systems wesentlich.

- Für die Verifizierbarkeit eines Systems bietet das ADT-Konzept besondere Unterstützung durch eine klare Trennung zwischen Spezifikation, Implementierung und Repräsentationsfunktion.

- Auch die Strukturierung eines Systems wird durch den Einsatz von ADTs erleichtert. Durch die Zerlegung des Systems in ADTs und Überlagerung einer "implementiert"-Hierarchie zwischen den einzelnen Abstraktionsebenen ergibt sich automatisch eine sinnvolle Systemstruktur.

Durch die Erfüllung aller dieser Prinzipien trägt das ADT-Konzept in hohem Maße zur Verbesserung der Zuverlässigkeit und Verständlichkeit von SW-Systemen bei. Zusätzlich führt der Einsatz von ADTs weg von der konventionellen funktionsorientierten zu einer modernen objektorientierten Programmierung von SW-Systemen. Gerade diese Eigenschaft macht das ADT-Konzept - neben seiner Verwendung als Entwurfshilfsmittel - zur idealen Basis für die Durchführung von Objektverwaltungsaufgaben. Die in diesem Zusammenhang zusätzlich in das ADT-Konzept zu integrierenden Maßnahmen zur Objektverwaltung, wie Synchronisation und Schutzmechanismen, werden im Abschnitt 3.5 detailliert dargestellt. Zuvor soll jedoch ein kurzer Überblick über die verschiedenen bereits existierenden Entwurfsmethoden gegeben werden, wobei der Schwerpunkt der Betrachtungen auf den Anforderungen, die an Entwurfsmethoden zu stellen sind, liegen wird.

3.4. SW-Entwurfsmethoden

Ziel dieses Abschnitts ist nicht, eine detaillierte Vorstellung aller existierenden SW-Entwurfsmethoden zu geben, - hier sei auf die zahlreiche Literatur (z.B. <Lud78>, <Flo81>, <Keu82> oder <Bal82>) verwiesen - sondern vielmehr die an SW-Entwurfsmethoden zu stellenden Anforderungen darzulegen und die existierenden Methoden bezüglich ihres Einsatzgebiets und auch bezüglich der Erfüllung dieser Anforderungen zu klassifizieren.

Die breite Palette der SW-Engineering-Methoden deckt neben der eigentlichen Entwurfsphase auch teilweise die vorausgehende Definitionsphase und die nachfolgende Implementierungsphase mit ab. Charakteristisch ist dabei, daß alle Methoden nur für jeweils eine Phase schwerpunktmäßig ausgelegt sind und keine Methode existiert, die für alle Phasen gleichermaßen gut anwendbar ist.

Zur besseren Einordnung der Methoden muß die Entwurfsphase noch in zwei voneinander verschiedene Teilphasen eingeteilt werden. In der Systementwurfsphase wird im wesentlichen die Systemstruktur entworfen, d.h. es erfolgt die Zerlegung des Gesamtsystems in einzelne Komponenten, die Abgrenzung von Teilaufgaben und die Beschreibung der systeminternen Schnittstellen. Ergebnis dieser Phase ist der sogenannte Grobentwurf, der die vollständige Systemarchitektur beschreibt.

In der Programmentwurfsphase wird der Lösungsansatz schrittweise verfeinert. Hierzu gehört insbesondere die exakte Spezifikation des Leistungsumfanges und der Schnittstellen der einzelnen Systemkomponenten als Basis für die nachfolgende Implementierungsphase. Zu bemerken ist, daß das Ergebnis dieser Phase - die Spezifikation - die Funktionen des Systems bereits vollständig festlegt.

Grob betrachtet lassen sich die meisten Entwurfsmethoden einordnen in solche, die entweder mehr die Definitions- und Systementwurfsphase oder die Programmentwurfs- und Implementierungsphase unterstützen. Die Methoden für die Definitions- und Systementwurfsphase sind im allgemeinen durch informale, d.h. graphische oder rein verbale Beschreibungen gekennzeichnet und können keinesfalls - auch wenn dies in der Praxis sicher oft geschieht - als direkte Basis für die Implementierung verwendet werden. Beispiele für entsprechende Systementwurfsmethoden sind SADT, HIPO, RSL, PSL/PSA, SD oder PLASMA/D (vgl. <Bal82> oder <Keu82>). Bei den Methoden für die Programmentwurfs- und Implementierungsphase existieren dagegen durchaus Spezifikationsansätze, die im Sinne einer Basis für die Implementierung eingesetzt werden können.

In dieser Arbeit sollen ausschließlich Methoden zur Unterstützung des Programmentwurfs betrachtet werden, mit dem Ziel, eine Spezifikations- und Implementierungsmethode für Echtzeitanwendungsfälle in der Produktion zu entwickeln. Zunächst sollen hier allerdings die Anforderungen, die an Programmentwurfsmethoden zu stellen sind, kurz präsentiert werden.

Die aus der Sicht der Echtzeitanwendungen an SW-Engineering-Methoden zu stellenden Anforderungen wurden bereits im Abschnitt 2.4 präsentiert. Es handelte sich dabei um Konstrukte für die Synchronisation, die Definition von Prioritäten und um Schutzkonstrukte. Aus der Sicht der erläuterten SW-Entwurfsprinzipien sind nun ebenfalls einige wichtige Anforderungen an SW-Engineering-Methoden zu stellen. Bereits hier sei erwähnt, daß viele der existierenden Programmentwurfsmethoden diese Anforderungen nicht oder nur zum Teil erfüllen.

Primäres Ziel von Methoden zur Unterstützung der Programmentwurfsphase ist die Erstellung einer Spezifikation der einzelnen Systemkomponenten als direkte Basis für deren Implementierung sowie als Mittel zur Festlegung des Lei-

stungsumfangs und der externen Benutzerschnittstellen der Module. Besonderer Wert ist dabei auf eine formale Spezifikation zu legen, da nur durch den Einsatz formaler Spezifikationsmethoden das notwendige Maß an Exaktheit gewährleistet ist. Weitere Vorteile einer formalen Spezifikation sind deren Prüfbarkeit auf Konsistenz und Vollständigkeit sowie der Nachweis invarianter Systemeigenschaften bereits in der Spezifikationsphase. Schließlich ist eine formale Spezifikation unabdingbare Voraussetzung für die Verifizierbarkeit der erstellten Software, d.h. für den Nachweis, daß die Implementierung die durch die Spezifikation festgelegten Eigenschaften besitzt.

Eine weitere Forderung ist eine möglichst direkte Ableitbarkeit der Implementierung aus der Spezifikation. Voraussetzung dafür ist ein einheitlicher Ansatz für das Spezifikations- und Implementierungskonzept. Hierfür kann die Zugrundelegung eines formalen Modells eine große Unterstützung bieten. Insbesondere zur exakten Festlegung der Semantik der Spezifikations- und Implementierungsmethode ist ein formales Modell einer rein verbalen Beschreibung vorzuziehen. Ein weiterer Vorteil eines formalen Modells ist beispielsweise die Möglichkeit, bereits auf dieser Ebene anwendungsbezogene Fragen, wie die Verträglichkeit asynchroner Aktivitäten zu formulieren bzw. zu klären.

Bei der Definition des Spezifikations- und Implementierungskonzepts sind naturgemäß die im vorhergehenden Abschnitt dargestellten SW-Entwurfsprinzipien besonders zu berücksichtigen. An erster Stelle ist hier die Möglichkeit der Abstraktion zu erwähnen, wobei in das Konzept neben der prozeduralen Abstraktion auch Datenabstraktionsmechanismen zu integrieren sind. Natürlich müssen auch die weiteren Forderungen wie Geheimnisprinzip, Modularisierung, Hierarchiebildung, Lokalität und Modifizierbarkeit von der Entwurfsmethode unterstützt werden.

Als Basiskonzept zur Erfüllung all dieser Forderungen wurde im Abschnitt 3.3 das ADT-Konzept eingeführt. Es liegt deshalb nahe, dieses Konzept als Grundlage für die zu entwerfende Spezifikations- und Implementierungsmethode zu verwenden, und die durch Echtzeitanwendungen zusätzlich zu fordernden Eigenschaften in das Konzept zu integrieren.

An dieser Stelle sei nun ein kurzer Überblick über existierende Programmentwurfsmethoden gegeben und die Frage nach Erfüllung der o.g. Eigenschaften durch die präsentierten Methoden beantwortet.

Eine weit verbreitete Entwurfsmethode ist die Jackson Design Methodology (JDM) (<Jac75>, <Jac76>). Ihr liegt eine datenstrukturorientierte Betrachtungsweise zugrunde. Die Programmstruktur wird aus der Struktur der Ein- und Ausgabedaten abgeleitet. Die Methode wird in der Literatur allgemein als wenig geeignet für den Programmentwurf angesehen, da sie eine Reihe gravierender Mängel aufweist. So wird gegen das Geheimnisprinzip und das Prinzip der Datenabstraktion verstoßen, und bei einer Änderung der Datenstruktur wird auch eine Änderung der Algorithmenstruktur notwendig. Die für Echtzeitanwendungen erforderlichen Konstrukte sind in der Methode nicht vorgesehen und auch praktisch nicht zu integrieren; durch Synchronisationsprobleme entstehen sogar sogenannte Strukturkonflikte (vgl <Bal82>), die nur schwer auflösbar sind. Schließlich ist die Methode völlig informal, so daß die wichtige Forderung nach einer exakten formalen Spezifikation unerfüllt bleibt.

Eine weitere informale Programmentwurfsmethode, die allerdings im Gegensatz zur JDM funktionsorientiert ist, ist die PDL (Program Design Language - <Cai75>). PDL ist gekennzeichnet durch Sprachmittel, die der natürlichen Sprache eine Struktur geben; man spricht deshalb oft von "strukturiertem Englisch" (bzw. Deutsch). Neben dem gravierenden Nachteil, daß eine formale Spezifikation mit PDL nicht erstellbar ist, fehlen bei der Methode Echtzeitkomponenten und die Möglichkeit, Datenstrukturen zu beschreiben.

Eine Reihe von Entwurfsmethoden wurde speziell für Echtzeitanwendungen konzipiert. Es handelt sich dabei um

- DARTS (Design Aid for Real Time Systems) von der GEI (Gesellschaft für Elektronische Informationsverarbeitung) <Keu82>.

- EPOS (Entwurfsunterstützendes Prozeßorientiertes Spezifikationssystem) vom Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart (<Bie79>, <Bie81>, <Göh81a>, <Göh81b>).

- ESPRESSO (Erstellung der Spezifikation von Prozeßrechner Software) vom KfK-Institut für Datenverarbeitung in der Technik (<Lud81a>, <Lud81b>).

Leider sind alle drei Methoden informal und bieten keine Möglichkeiten zur formalen Spezifikation. Zudem werden von diesen Entwurfsmethoden die aus den Entwurfsprinzipien abzuleitenden Forderungen wie z.B. Abstraktions- und Verifikationsmöglichkeiten nicht erfüllt. Ein direkter Übergang von der Spezifikation zur Implementierung ist ebenfalls nicht gegeben.

Neben diesen informalen und deshalb für den Programmentwurf wenig geeigneten Entwurfsmethoden existieren einige Sprachkonzepte, die speziell für diese Anwendungen entwickelt wurden. Man muß hier unterscheiden zwischen eigenständigen Spezifikationssprachen und der Integration entwurfsorientierter Sprachkonzepte in neuere Programmiersprachen. Als Beispiel für eigenständige Spezifikationssprachen sind zu nennen die Sprache SPEZI (<Koc79>, <Flo78>), die an der TU Berlin entwickelt wurde und SPECIAL vom Stanford Research Institute <Rou77>, die auf der Spezifikationsmethode von Parnas (vgl. 3.3.5) basiert. Moderne Programmiersprachen, die Programmentwurfskonzepte beinhalten, sind z.B. ADA, ALPHARD, CLU, EUCLID oder MODULA-2. Eine detaillierte

Beschreibung dieser Sprachkonzepte findet sich bei Balzert <Bal82>, so daß hier auf eine nochmalige Darstellung verzichtet werden kann. Jedoch soll hier eine Wertung der verschiedenen Konzepte bezüglich ihrer Erfüllung von Anforderungen an Entwurfsmethoden und ihrer Einsatzmöglichkeit bei Echtzeitanwendungen präsentiert werden.

Die reinen Spezifikationssprachen wie SPEZI und SPECIAL haben den Vorteil, daß sie unabhängig von den zur Implementierung verwendeten Programmiersprachen eingesetzt werden können, und so im Prinzip mit jeder Programmiersprache kombiniert werden können. Dem gegenüber steht der Nachteil, daß neben der eigentlichen Programmiersprache eine zusätzliche Sprache erlernt werden muß, und daß sich der Übergang von der Spezifikation zur Implementierung wegen des nicht durchgängigen Konzepts unter Umständen recht zeitaufwendig und schwierig gestaltet.

Beide Sprachen unterstützen in ausreichendem Maß die Forderung nach einer formalen Spezifikation und nach Abstraktionsmöglichkeiten, insbesondere ist bei SPECIAL - einer nicht prozeduralen Sprache - eine direkte Entsprechung zur Spezifikationsmethode von Parnas gegeben. In SPEZI ist die Spezifikation von einzuhaltenden Reihenfolgen zwischen den einzelnen Operationen durch Pfadausdrücke (vgl. 3.5.1) möglich. Schutzkonstrukte sind bei keiner der Sprachen vorgesehen.

Der große Vorteil moderner Programmiersprachen mit integrierten Entwurfskonzepten ist der einheitliche Ansatz für den Programmentwurf und die eigentliche Implementierung. Alle diese Sprachen unterstützen sowohl prozedurale als auch Datenabstraktion, eine formale und nichtprozedurale Spezifikation ist jedoch nur in der Sprache ALPHARD gegeben. Hier wird streng zwischen Spezifikationsteil, Implementierungsteil und Repräsentationsteil (Abbildung der Implementierung auf die Spezifikation - vgl. 3.3) unterschieden, wobei die nichtprozedurale Spezifikation an die Methode der Prädikatentransformation von Hoare angelehnt ist. Der be-

sondere Vorteil von ALPHARD gegenüber anderen Sprachkonzepten ist, daß die verwendeten Sprachkonstrukte direkt dem in Abschnitt 3.3 eingeführten ADT-Konzept mit Spezifikationsteil, Implementierungsteil und Repräsentationsfunktion entsprechen, und so sowohl ein einfacher Übergang von der Spezifikation zur Implementierung als auch gute Verifikationsmöglichkeiten gegeben sind. Die für Echtzeitanwendungen zu fordernden Synchronisations- und Schutzkonzepte sind standardmäßig nicht vorhanden, lassen sich jedoch leicht in das Konzept integrieren.

Zusammenfassend läßt sich somit sagen, daß der bei ALPHARD eingeschlagene Weg die beste Basis für eine zu entwickelnde Spezifikations- und Implementierungsmethodik bietet, wobei natürlich noch die Ergänzungen aufgrund der Echtzeitanwendungen integriert werden müssen. Das in dieser Arbeit zu entwickelnde Spezifikations- und Implementierungskonzept basiert deshalb auf den in ALPHARD angegebenen Konstrukten und besitzt als Kernkonstrukt das ADT-Konzept. Die in das ADT-Konzept zu integrierenden Synchronisations- und Schutzkonstrukte als Maßnahmen zur Verwaltung abstrakter Objekte sollen nun im folgenden Abschnitt näher erläutert werden.

3.5 Aufgaben der Objektverwaltung

Die Einführung von Mechanismen zur Objektverwaltung wurde bereits in einem sehr frühen Stadium der Datenverarbeitung notwendig. Durch die Umstellung vom Ein-Benutzer- auf Mehr-Benutzer-Betrieb entstand das Problem, daß mehrere aktive Einheiten wie Prozesse oder Programme auf gemeinsame passive Einheiten wie Betriebsmittel oder Daten gleichzeitig zugreifen wollten. Daraus resultierte zwangsweise die Notwendigkeit, diese passiven Einheiten - Objekte genannt - so zu verwalten, daß Zugriffe, die die Konsistenz der Objekte zerstören können, verhindert werden.

Die hierzu erforderlichen Maßnahmen bedeuten immer eine Reglementierung der Zugriffe auf Objekte. Zunächst muß sichergestellt werden, daß nur mit den "passenden" Operationen und bei definierten Zuständen auf das Objekt zugegriffen werden darf, um die allgemeine Konsistenz des Objekts zu erhalten. Eine weitere Einschränkung besteht darin, "nicht erlaubte" Zugriffe (beispielsweise abhängig vom Objektzustand oder Eingabeparametern) mit Operationen, die für das Objekt definiert sind, zu verhindern. Die hierunter fallenden Maßnahmen werden mit den Begriffen Datenschutz bzw. Datensicherheit bezeichnet. Schließlich müssen gleichzeitig stattfindende Zugriffe auf Objekte synchronisiert werden, d.h. so koordiniert werden, daß sie sich nicht gegenseitig beeinflussen.

In herkömmlichen Betriebssystemen wurden diese Maßnahmen allerdings nur in sehr unzureichendem Maße unterstützt. Die zur Verfügung gestellten Mechanismen reichten meist über einfachen Speicherschutz oder die Synchronisation der einzelnen Prozesse über Semaphore nicht hinaus. Die unmittelbare Konsequenz dieser sehr groben, globalen Mechanismen waren unzuverlässige SW-Systeme und nicht selten ganze Systemblockaden.

Erst die Einführung der objektorientierten Betrachtungsweise in die Programmierung ließ neue Denkansätze entstehen, um die bei der Objektverwaltung zu lösenden Aufgaben zuverlässiger zu erledigen. Entscheidend ist hier gegenüber herkömmlichen Konzepten der Übergang von globalen Objektverwaltungsmechanismen zu lokalen, objektorientierten Mechanismen. Als ideale Grundlage hierfür kann das ADT-Konzept gelten, das ja ein allgemeines konsistenzerhaltendes Modell für abstrakte Objekte und zugleich eine Basis für den zuverlässigen SW-Entwurf darstellt. Für die Erledigung von Objektverwaltungsaufgaben muß das ADT-Konzept dann noch um Synchronisations- und Schutzaspekte erweitert werden. Erste Ansätze existieren bereits; man betrachte hierzu z.B. <Ker82> (Erweiterung um Synchronisation) oder <Wul76> (Erweiterung um sehr elementare Schutzkonstrukte).

Allerdings gibt es bisher noch kein allgemeingültiges - alle Aufgabenbereiche der Objektverwaltung umfassendes - Modell. In dieser Arbeit soll deshalb aufbauend auf dem ADT-Konzept eine gemeinsame und einheitliche Behandlung sämtlicher Objektverwaltungsprobleme durchgeführt werden, wobei sich die Einbettung des ADT-Konzepts in ein Objektverwaltungssystem folgendermaßen darstellt:

- ADTs definieren das Verhalten der abstrakten Objekte des entsprechenden Typs. Die Regeln zur Verwaltung der Zugriffe auf die Objekte werden im Spezifikationsteil des ADT`s festgelegt.

- Im vorliegenden Objektverwaltungsmodell ist jedem abstrakten Objekt eine Abstrakte Maschine zugeordnet, die die Zugriffe auf das Objekt gemäß den im ADT spezifizierten Regeln verwaltet. In der Praxis müssen die Aufgaben dieser Abstrakten Maschine dann vom Implementierungsteil des ADT`s in Zusammenarbeit mit dem Betriebssystem erledigt werden. Moderne Betriebssysteme bieten hierfür zumindest teilweise schon eine Unterstützung an.

Man beachte, daß das ADT-Konzept in dieser Arbeit gleichzeitig als Basis für die Spezifikation und Implementierung und für das formale OV-Modell, das die Semantik der Spezifikations- und Implementierungsmethode festlegt, dient. Dieser einheitliche Ansatz für das formale Modell, die Spezifikation und Implementierung ist eine wichtige Voraussetzung für die Brauchbarkeit des Konzepts.

Bevor nun näher auf die bei der Konstruktion eines OV-Modells zu berücksichtigenden Bedingungen eingegangen wird, soll eine überblicksartige Darstellung einen Einblick in die Themenkreise Synchronisation und Schutz/Sicherheit geben.

3.5.1 Synchronisation

In umfangreichen SW-Systemen arbeiten die Subjekte (bzw. Prozesse) des Systems nicht isoliert für sich (Zugriff nur auf private Objekte), sondern kooperieren in dem Sinne, daß sie auf gemeinsame Objekte zugreifen. Eine wichtige Maßnahme zur Vermeidung inkonsistenter Objektzustände in solchen Fällen ist die Synchronisation. Sie ist verantwortlich dafür, daß eine Zugriffsoperation erst dann ausgeführt werden darf, wenn gewisse Bedingungen erfüllt sind. Diese Bedingungen können vom Typ des Objekts, von der Verwaltungsstrategie, vom momentanen Wert des Objekts und von den Werten der Aufrufparameter der Operation abhängen. Die Synchronisation muß bei Nicht-Erfüllung der entsprechenden Synchronisationsbedingungen dafür sorgen, daß der Zugriff so lange verzögert wird, bis die Bedingungen erfüllt sind.

Das erste Synchronisationskonzept außerhalb der Hardware stammt von Dijkstra <Dij68>. Er führte 1968 die P- und V- Operationen auf dem Datentyp "Semaphore" ein. Dieses Konzept ist wegen seiner leichten Implementierbarkeit zum Teil heute noch gebräuchlich. Sein Nachteil besteht darin,

daß die Synchronisation nicht im Objekt selbst, sondern in den auf das Objekt zugreifenden Prozessen stattfindet. Dies entspricht sehr stark einer "unstrukturierten Programmierung" und führte in der Praxis häufig zu Systemfehlern bzw. sogar Systemblockaden.

Die bedingten kritischen Abschnitte von Brinch-Hansen <Bri72> und Hoare <Hoa72b> stellten eine Verfeinerung des Semaphore-Konzepts dar. Ein Zugriff auf ein - als gemeinsam deklariertes - Objekt r (resource) darf hier nur in einem explizit deklarierten kritischen Abschnitt C stattfinden, falls gewisse Bedingungen B erfüllt sind. Hierfür wurde das Sprachkonstrukt "with r when B do C" eingeführt. Auch hier ist allerdings der wesentliche Nachteil, nämlich die Synchronisation innerhalb der Prozesse, noch nicht beseitigt.

Eine Verbesserung brachte hier erst das Monitorkonzept von Hoare <Hoa74>. Hier wurde zum ersten Mal die Synchronisation als Abstraktionsmittel eingesetzt, da sie nicht mehr bei den auf ein Objekt zugreifenden Prozessen lag. Ein Monitor enthält eine Reihe von Prozeduren auf ein Objekt und lokale Daten. Monitor-Prozeduren laufen unter gegenseitigem Ausschluß. Die Synchronisation findet hier zwar im Objekt selbst statt, ist aber innerhalb des Objekts immer noch auf die einzelnen Prozeduren verteilt. Die Prozeduren synchronisieren sich mit wait- und signal- Operationen, die den P- und V- Operationen mit allen ihren Nachteilen entsprechen.

Die bisher vorgestellten Konzepte sind reine Implementierungskonzepte. Zur Behandlung der Synchronisation im Hinblick auf die Unterstützung der Konstruktion zuverlässiger Systeme sind aber Konzepte notwendig, die sowohl die Spezifikation, als auch die Implementierung von Synchronisationsproblemen gestatten. Der erste Versuch in dieser Richtung, d.h. das Festlegen der Synchronisation auf Objekte unabhängig von der Implementierung (außerhalb der Prozeduren), stammt von Campbell und Habermann <Cam74>. In ihren Pfad-Ausdrücken wird die erlaubte Reihenfolge von Operationen in einem separaten Teil innerhalb der Spezifikation

des Datentyps festgelegt. Wegen der Regularität der Pfad-Ausdrücke ergeben sich allerdings nur eingeschränkte Anwendungen. Eine Erweiterung dieser Methode präsentierten 1978 Laventhal und Andler in <Lav78>, die sogenannten Prädikaten-Pfad-Ausdrücke. Allgemein betrachtet synchronisieren Pfad-Ausdrücke über die Historie von Operationsaufrufen; sie eignen sich gut zur Spezifikation von Reihenfolgeproblemen, sind jedoch schlechter bei einer wertabhängigen Synchronisation anwendbar.

Den neuesten Ansatz auf diesem Gebiet präsentiert Keramidis in <Ker82>. Er wird hier detaillierter vorgestellt, da die dort entwickelten Prinzipien für das Gebiet der Synchronisation Grundlagen dieser Arbeit sind. Dort findet sich auch eine wesentlich genauere Darstellung der hier nur informell vorgestellten anderen Synchronisationsmechanismen.

Basismodell für die Spezifikation und Implementierung von Objekten und deren Verwaltung ist bei Keramidis das ADT-Konzept. Dieses Konzept zur prozeduralen und Datenabstraktion wird um die Synchronisation als zusätzliches Abstraktionsmittel erweitert. Wie bei Laventhal und den PPAs kann die Synchronisation unabhängig von der Implementierung der abstrakten Operationen innerhalb des abstrakten Datentyps festgelegt werden. Dadurch wird z.B. auch das Lokalitätsprinzip berücksichtigt.

Die Festlegung der Synchronisation geschieht in einem eigenen Abschnitt des Spezifikationsteils, dem SYN-Teil. Dieser SYN-Teil betrifft die Ausführung mehrerer Operationen. In ihm wird festgelegt, ob zwei oder mehrere Operationen miteinander "verträglich" sind, d.h. ob - bzw. unter welchen Umständen - sie parallel ausgeführt werden können, ohne inkonsistente Objektzustände hervorzurufen. Auch Prioritäten zwischen den Operationen des ADT`s können hier festgelegt werden. Diese Verträglichkeits- bzw. Prioritätsspezifikationen gelten für jedes Objekt des entsprechenden ADT`s. Zugriffe auf verschiedene Objekte eines

ADT`s unterliegen keinen Einschränkungen.

Zur Formulierung der Verträglichkeits- und Prioritätsbedingungen wird der Begriff "Aktivität" eingeführt. Eine Aktivität ist die Ausführung einer aufgerufenen Operation des ADT`s. Sie beginnt mit dem Aufruf der Operation zu existieren und endet mit der Beendigung der Operation. Im allgemeinen besitzen die Operationen eines ADT`s mehrere Eingabeparameter. Eine Aktivität ist dann ein Aufruf einer Operation mit einer ganz bestimmten Belegung der Eingabeparameter. Eine Aktivitätenmenge ist die Zusammenfassung verschiedener Aktivitäten auf ein Objekt; die Menge aller Aktivitäten auf ein Objekt wird im folgenden mit A bezeichnet.

In <Ker82> werden nun verschiedene Konstrukte zur Spezifikation der Verträglichkeit und der Prioritäten zwischen Aktivitäten angegeben.

Die Relation VTGL $\subseteq$ A x A, die meist in Prädikatenform angegeben wird, legt fest, welche Aktivitäten miteinander verträglich sind. Alle nicht enthaltenen Paare sind unverträglich. Oft ist es einfacher, statt der Verträglichkeitsbedingung anzugeben, wann zwei Aktivitäten nicht miteinander verträglich sind. Dafür existiert das NVTGL-Konstrukt, wobei immer NVTGL = A x A - VTGL gilt. Die Angabe eines von beiden reicht dabei aus. Analog definiert PRIOR $\subseteq$ A x A die Priorität zwischen den einzelnen Aktivitäten. Beispiele findet man in <Ker82> oder im Kapitel 7 dieser Arbeit.

Worin liegen nun die Vorteile dieses Modells? Es bietet

- ein einheitliches Konzept und formales Modell für allgemeine Objektverwaltung und Synchronisation
- die Möglichkeit, Synchronisation auch von Zuständen des Objekts und Eingabeparametern der Operation abhängig zu formulieren
- eine leichte Formulierung bedingter Verträglichkeiten und dynamischer Prioritäten

Die Methode von Keramidis liegt außerdem der Intuition wesentlich näher als eine Synchronisation über die Historie von Ereignissen. Deshalb wurde sie in der vorliegenden Arbeit als Grundlage des zu erstellenden allgemeinen Objektverwaltungsmodells benutzt.

3.5.2 Schutz- und Sicherheitsaspekte

Seit den 60er Jahren befaßt sich die Informatik mit den Themenkreisen Schutz und Sicherheit. Ursprünglich waren dies Probleme des Speicherschutzes, wie z.B. Schutz des Betriebssystems vor Anwendern oder Schutz der Anwender untereinander; diese Schutzmechanismen waren teilweise sogar in der Hardware realisiert. Eine Erweiterung war dann der Schutz des Anwenders vor sich selbst oder der Schutz vor unerlaubten Zugriffen zu Dateien.

Heute versteht man darunter "jede unerlaubte oder unerwünschte Enthüllung, Modifikation oder Zerstörung von Information" (vgl <Lin76>). Das Thema gewinnt immer mehr an Bedeutung, seit Datenverarbeitungssysteme zur Verarbeitung sensitiver Daten eingesetzt werden, wie z.B. in militärischen Systemen oder zur Verwaltung von Datenbanken mit persönlichen Daten in der Medizin. In diesem Zusammenhang forcieren auch rechtliche Aspekte die Behandlung des Themas (man beachte z.B. das Bundesdatenschutzgesetz). Der wachsenden Bedeutung trägt auch die Tatsache Rechnung, daß gerade in den letzten 10 Jahren eine Vielzahl von Konzepten zur Lösung der anstehenden Probleme vorgeschlagen wurden.

Die bisherigen Ansätze hierzu sind stark von Forschungsergebnissen auf dem Gebiet der Betriebssysteme und auch von konkreten Betriebssystementwicklungen beeinflußt. Dies zeigt sich dadurch, daß diese Modelle alle von globalen Schutzkonstrukten ausgehen. Dies steht im Gegensatz zu der lokalen, objektorientierten Betrachtungsweise des ADT-Kon-

zepts, die in dieser Arbeit im Vordergrund steht. Trotzdem müssen natürlich die bisherigen Ansätze genauestens analysiert werden, um eine sinnvolle Integration von Schutzmechanismen in das Modell gewährleisten zu können. Im folgenden soll in diesem Abschnitt zuerst ein allgemeiner Überblick über den Themenkreis gegeben werden, und in Kapitel 4 dieser Arbeit detailliert auf existierende Schutzmodelle eingegangen werden.

Allen formalen Ansätzen für Schutzsysteme sind gewisse Grundkomponenten gemeinsam, die sich zum Teil schon durch die notwendige Integration des Schutzsystems in ein Betriebssystem ergeben:

In einem Schutzsystem wird die Menge der Individuen in eine Menge aktiver Einheiten (Subjekte) und eine Menge passiver Einheiten (Objekte) eingeteilt. Beispiele für Subjekte in einer Rechenanlage sind Prozesse, Jobs, Prozeduren, etc., für Objekte Files, Segmente, Speicherblöcke, HW-Ressourcen. Jedes Subjekt des Systems ist gleichzeitig auch Objekt (eine Prozedur kann z.B. eine andere aufrufen oder selbst aufgerufen werden). Subjekte handeln, indem sie auf Objekte in irgendeiner Weise "zugreifen". Für jedes Objekt existieren i.a. verschiedene Zugriffsarten (z.B. "Lesen", "Schreiben", "Aufrufen", "Belegen", ...), die insbesondere vom jeweiligen Typ des Objekts abhängen. Die Menge aller möglichen Zugriffe von Subjekten auf Objekte ist eine weitere Komponente eines Schutzsystems. Um Datenschutzaspekte berücksichtigen zu können, muß ein Schutzsystem erlauben, Einschränkungen für die Menge der Zugriffe, die ein Subjekt auf ein Objekt ausführen darf, zu formulieren. Diese Einschränkungen - Regeln des Schutzsystems genannt - bilden eine weitere Komponente eines Schutzsystems.

Für ein allgemeines Modell eines Schutzsystems ergeben sich also vier Grundkomponenten

- eine Menge von Subjekten
- eine Menge von Objekten
- eine Menge von möglichen Zugriffen der Subjekte auf die Objekte
- eine Menge von Regeln, die festlegen, welches Subjekt auf welches Objekt zugreifen darf

Bei allen bisher entwickelten konkreten Modellen von Schutzsystemen spielen die Begriffe "Schutz" und "Sicherheit" (englisch "protection" bzw. "security") eine entscheidende Rolle. Leider existiert gerade in der deutschsprachigen Literatur keine eindeutige Begriffsbildung: Die Begriffe "Schutz", "Sicherheit" und "Sicherung" werden oft überlappend benutzt. Insbesondere versteht man unter dem Begriff "Schutz" oft ausschließlich die mit dem Bundesdatenschutzgesetz zusammenhängenden Probleme. Aus diesem Grund erscheint eine Einordnung der in dieser Arbeit verwendeten Begriffe in das oben definierte allgemeine Konzept eines Schutzsystems an dieser Stelle notwendig.

In einem Schutzsystem soll unter dem Begriff "Sicherheit" die Garantie verstanden werden, daß die gegebenen Regeln eingehalten werden. Die Sicherheit eines Systems bezieht sich also immer auf vorgegebene Verhaltensregeln, d.h. Sicherheit ist ein relatives, nicht absolutes Konzept. Der Mechanismus, der für die Einhaltung der Regeln sorgt, heißt "Schutz". Der Schutzmechanismus muß, um arbeiten zu können, die Regeln des Schutzsystems mitgeteilt bekommen. Dies setzt die Formulierung der Regeln gemäß einer Konvention voraus; in Kapitel 4 dieser Arbeit finden sich Beispiele hierfür. Die Trennung zwischen dem Mechanismus, der die Einhaltung der Regeln überwacht (protection mechanism) und den Regeln selbst (security policy), heißt "policy/mechanism-separation". Dieses Prinzip ist ein wichtiges Abstraktionsmittel für Schutzsysteme - es entspricht im wesentlichen der Trennung zwischen Spezifikation und Implementie-

rung in der SW-Technologie.

Bevor nun die unterschiedlichen bereits existierenden Ansätze zum Erreichen von Sicherheit vorgestellt werden, soll an dieser Stelle auf verschiedene Schutzprobleme eingegangen werden; die hier beschriebenen Probleme sind vor allem Probleme, die bei der praktischen Anwendung von Schutzsystemen in Betriebssystemen auftauchen. Sie orientieren sich zum Teil an realen Zugriffssituationen in Betriebssystemen, wie z.B. dem Aufruf einer (System-) Prozedur durch einen Benutzer und dem damit verbundenen notwendigen Zugriffsschutz (vgl. <Coh75>). Trotzdem sind gerade diese praktischen Probleme von eminenter Bedeutung: zum einen muß ein Modell eines Schutzsystems natürlich Grundlagen für die Lösung der Probleme in der Praxis zur Verfügung stellen, zum anderen ermöglicht erst die Kenntnis dieser Probleme eine Beurteilung der bisher vorgeschlagenen Konzepte zu ihrer Lösung.

Im wesentlichen wird in einem Betriebssystem der Einsatz von Schutzmethoden an vier verschiedenen Stellen notwendig:

- Schutz des Betriebssystems vor Anwendern
- Schutz der Anwender untereinander
- Schutz der Anwender vor Betriebssystemkomponenten
- Schutz der Anwender vor eigenen Programmen

Die im folgenden dargestellten konkreten Schutzprobleme beziehen sich auf diese Situationen:

- Gegenseitiges Mißtrauen

  Beim Aufruf eines (eigenen oder System-) Programms durch einen Benutzer müssen zwei Forderungen erfüllt werden:
  * der Benutzer will sichergehen, daß das aufgerufene Programm nicht absichtlich oder versehentlich auf Daten oder Informationen des Benutzers zugreift

* das aufgerufene Programm braucht die Garantie, daß der Aufrufende nicht ohne Erlaubnis auf programmeigene Daten oder Informationen zugreifen kann

Diese beiden Aspekte bezeichnet man zusammen als das Problem des gegenseitigen Mißtrauens (mutual suspicion). Programme, die absichtlich versuchen, auf Daten eines Aufrufers ohne dessen Erlaubnis zuzugreifen, heißen "Trojanische Pferde". Ein Beispiel hierfür wäre ein Editor, der seinem Autor geheime Daten seiner Benutzer mitteilt. Je nachdem, ob es sich bei der Aktion eines Trojanischen Pferdes um eine unerlaubte Enthüllung oder Modifikation handelt, werden in der Literatur unterschiedliche Bezeichnungen verwendet. Im Rahmen der nächsten beiden Punkte soll darauf näher eingegangen werden.

- Das Confinement Problem

Das Confinement Problem ist eines der beiden Teilprobleme des Trojanischen Pferd-Problems. Es behandelt die unerlaubte Enthüllung von Information durch ein aufgerufenes Programm. Dieses Problem wurde zum ersten Mal von Lampson <Lam73> erkannt. Lampson identifiziert drei verschiedene Arten von "Kanälen", über die ein Programm Information "durchsickern" lassen kann. Legitime Kanäle sind z.B. der Output des Programms, wie Ausdrucke oder Ausgabeparameter. Speicherkanäle sind z.B. Files oder gemeinsame Variable, über die Information zu anderen Programmen übertragen werden kann. Verdeckte Kanäle sind Pfade, die normalerweise nicht für Informationsübertragung gedacht sind, wie z.B. die Laufzeit von Programmen, die Zahl oder Frequenz von Plattenzugriffen eines Programms etc.

- Das Integritätsproblem

  Das Integritätsproblem ist der zweite Teilaspekt der Trojanischen Pferd-Problems. Hier handelt es sich um das Verhindern von unerlaubter Modifikation oder Zerstörung von Information durch ein aufgerufenes Programm bzw. allgemein durch ein nicht autorisiertes Subjekt.

- Ausbreitungseinschränkungen für Rechte

  In vielen realen Schutzsystemen werden die Regeln mit Hilfe von Rechten realisiert, d.h. ein Subjekt darf einen bestimmten Zugriff auf ein Objekt nur dann ausüben, wenn es das Recht dazu hat (vgl. <Coh75>). Im allgemeinen kann in diesen Systemen der Besitzer eines Objekts Rechte an diesem Objekt an andere Benutzer weitergeben. Oft ist jedoch eine weitere Ausbreitung (weitere Übergabe an einen dritten Benutzer) unerwünscht. Zur Verhinderung dieser Situation müssen geeignete Mittel zur Verfügung gestellt werden. Auch der Rückruf von Rechten (Besitzer will ein Recht an einem seiner Objekte nur zeitweise vergeben) muß in einem solchen System möglich sein.

Zur Lösung dieser Schutzprobleme und zur Durchsetzung der Regeln eines Schutzsystems gibt es eine Vielzahl von Mechanismen. Für diese Mechanismen können drei grundlegende Ziele aufgezeigt werden:

- Schutz von Objekten vor unerlaubten Zugriffen von Subjekten
- Schutz vor unerlaubter Enthüllung von Information (Datengeheimhaltung, privacy)
- Schutz vor unerlaubter Modifikation von Information (Datenintegrität, integrity)

Diese Mechanismen können in interne Mechanismen (innerhalb des Rechners) und externe Mechanismem (außerhalb des Rech-

ners) eingeteilt werden (vgl. <Den79>).

- interne Mechanismen

  * Zugriffskontrolle (access control)
  * Informationsflußkontrolle (information flow control)
  * Schlußfolgerungskontrolle (inference control)
  * Verschlüsselung von Daten (cryptographic control)

- externe Mechanismen

  * Sicherung des Gebäudes und Raumes, in dem der Rechner steht (Zugangskontrolle)
  * Auswahl und Überwachung des Bedienpersonals und Benutzerkreises
  * Feuerschutz für Rechner und Speichermedien
  * Diebstahlschutz für Speichermedien
  * Duplizieren von Daten bis zu gewissen Stützpunkten
  * Abschirmung von Übertragungsleitungen
  * Verschlüsselung zu übertragender Daten
  * Benutzeridentifikation z.B. über Passwörter oder maschinenlesbare Ausweise (authentification)
  * Passwortverwaltung und Sicherung
  * security monitoring
  * security auditing

Diese internen und externen Mechanismen werden unter dem Begriff Datensicherung zusammengefaßt, d.h. unter Datensicherung versteht man alle organisatorischen und technischen Maßnahmen zur Bewahrung von gespeicherter Information vor Verlust, Zerstörung, unberechtigtem Zugriff und unberechtigter Änderung.

In dieser Arbeit sollen nur interne Mechanismen betrachtet werden, und hier auch nur die Themengebiete Zugriffskontrolle und Informationsflußkontrolle, da nur sie mit Objektverwaltung im eigentlichen Sinn zusammenhängen. Im folgenden findet sich nun ein kurzer Überblick über die vier internen Mechanismen (die Modelle für Zugriffskontrolle und Informationsflußkontrolle werden in Kapitel 4 dieser Arbeit noch ausführlich dargestellt und miteinander verglichen).

Die Aufgabe der Zugriffskontrolle (access control) ist der Schutz von Objekten vor unerlaubtem Zugriff von Subjekten. Das System wird in aktive Einheiten (Subjekte), passive Einheiten (Objekte) und Zugriffsarten (Operationen) auf Objekte, wie z.B. "read", "write", "append", "execute", "call" u.a. eingeteilt. Assoziiert zu jeder Zugriffsart sind Zugriffsrechte, die die Ausführung der entsprechenden Operationen erlauben. Üblicherweise ist der momentane Stand der Zugriffsberechtigung in der Zugriffsmatrix (access matrix) hinterlegt. Die Zeilen der Matrix bilden die Subjekte, die Spalten die Objekte (Subjekte sind gleichzeitig immer Objekte) des Systems. Das (i,j)-te Element der Matrix enthält die Zugriffsrechte des i-ten Subjekts auf das j-te Objekt. Meistens existiert noch ein spezielles Recht "besitzt", bzw. Rechte, die dazu berechtigen, Rechte an Objekten weiterzugeben oder zu nehmen. Hierfür und für das Erzeugen bzw. Zerstören von Subjekten oder Objekten gibt es Regeln, die dann eine entsprechende Veränderung der Zugriffsmatrix bewirken. Die Entscheidung, ob ein beabsichtigter Zugriff erlaubt ist, oder nicht, hängt allein vom Inhalt der Zugriffsmatrix ab, nicht vom Wert des Objekts. Man nennt deshalb diese Modelle "object-dependent" im Gegensatz zu "value-dependent" oder "content-dependent" Modellen. Da der Besitzer eines Objekts i.a. entscheiden kann, an wen er Rechte an diesem Objekt weitergibt, werden diese Modelle in der englischsprachigen Literatur "discretionary-models" genannt.

Eine Mittelstellung zwischen Zugriffskontrolle und Informationsflußkontrolle nehmen die Modelle für die militärische Sicherheit ein. Wichtig ist hier vor allem der Schutz vor unerlaubter Enthüllung von Information (In reinen Objektschutzmodellen ist dies nämlich nicht gewährleistet, vgl. Kapitel 4). Diese Modelle basieren ebenfalls auf einer Zugriffsmatrix. Zusätzlich sind jedoch Subjekte und Objekte in Sicherheitsklassen eingeteilt. Die Klassen sind eine endliche geordnete Menge, z.B. "unklassifiziert", "vertraulich", "geheim", "streng geheim". Ziel dieser Modelle ist, daß Information nur in einer Richtung (von unten nach oben) fließen darf. Gewährleistet wird dies durch zwei Regeln:

- Hat ein Subjekt read-Zugriff auf ein Objekt, so muß die Sicherheitsklasse des Subjekts größer oder gleich der des Objekts sein ("einfache Sicherheitsbedingung").
- Hat ein Subjekt write-Zugriff auf ein Objekt, so muß die Sicherheitsklasse des Subjekts kleiner oder gleich der des Objekts sein ("*-Eigenschaft").

Die erste Bedingung regelt die normale Weitergabe von Informationen, die zweite Bedingung verhindert das Kopieren von Information in Dokumente mit niedrigerer Klassifikation. Auch in diesen Modellen können Regeln zur Veränderung der Zugriffsmatrix angegeben werden. Zu beachten ist jedoch, daß diese Regeln die beiden Eigenschaften erhalten müssen.

Im Gegensatz zu den Modellen für militärische Sicherheit betrachten Modelle zur Informationsflußkontrolle (information flow control) nicht jeden potentiellen Informationsfluß, sondern nur tatsächliche Flüsse. Der Informationsfluß in Programmen wird von den Eingabevariablen über die einzelnen Statements bis zu den Ausgabevariablen verfolgt. Es gibt syntaktische Methoden, die Formen von Statements betrachten und den resultierenden Fluß spezifizieren und semantische Methoden, die zusätzlich noch Zustände (Werte aller Variablen) berücksichtigen. Semantische Me-

thoden sind feiner (vollständiger) als syntaktische, jedoch schwieriger anzuwenden. Beide Methoden basieren auf den Prinzipien der Programmverifikation; sie entdecken nur Flüsse über legitime und Speicherkanäle. Ein weithin noch offenes Gebiet ist der Fluß über verdeckte Kanäle. Diese Kanäle sind wahrscheinlich sehr schwierig zu schließen. Bis heute gibt es jedenfalls noch keine Lösung, nicht einmal vielversprechende Ansätze dazu.

Die Schlußfolgerungskontrolle ist ein Mechanismus, der bei statistischen Datenbanken notwendig ist: Hier müssen für statistische Abfragen sensitive Informationen über Individuen verknüpft werden und statistische Aussagen getroffen werden, ohne einzelne Informationen preiszugeben. Das Problem besteht darin, daß durch eine Reihe von geschickt gewählten statistischen Abfragen und die Korrelation der Antworten untereinander bzw. mit anderweitig erhaltenen Fakten, vertrauliche Information enthüllt werden kann. Die Verhinderung der Enthüllung sensitiver Daten über Individuen ist die Aufgabe der Schlußfolgerungskontrolle. Dies ist mit den bisherigen Methoden zur Zugriffs- oder Informationsflußkontrolle nicht lösbar. Da jede statistische Zusammenfassung irgendwelche Spuren der Originalinformation enthält, wird es nie ein System geben können, das nicht überlistbar ist. Ziel der Methoden zur Schlußfolgerungskontrolle kann deshalb nur sein, dies so schwierig wie möglich zu machen.

Bei einer Übertragung bzw. Speicherung von Daten in unsicheren Medien hilft keine der bisher erwähnten Methoden vor unerlaubter Enthüllung. Hier ist eine Verschlüsselung der Daten notwendig. Der Originaltext wird nach den Regeln eines geheimen Schlüssels verarbeitet, und ergibt so den - ohne Schlüssel unsinnigen - verschlüsselten Text. Es gibt auf diesem Gebiet die verschiedensten Methoden, von einfachen Schlüsseln (Information ist leicht zu verschlüsseln, der Schlüssel aber auch leicht zu knacken) bis zu unknackbaren Schlüsseln: Hier ist der Schlüssel eine Zufallsfolge von Bits - genau so lang wie die Information selbst; jedes Bit des Schlüssels gibt an, ob das entsprechende Informa-

tionsbit komplementiert wird oder nicht. Die Wahl der Schlüssel ist vor allem eine Frage der praktischen Anwendung. Im allgemeinen werden Schlüssel benutzt, die wesentlich kürzer sind als die Information selbst.

Wie bereits erwähnt, werden in dieser Arbeit ausschließlich die Themen Zugriffskontrolle und Informationsflußkontrolle behandelt. Die Gründe dafür sind:

- nur diese Gebiete sind Themen der Objektverwaltung im engeren Sinn
- für den Bereich der SW-Technologie sind nur diese Gebiete von Interesse
- in dieser Arbeit steht die Betrachtung von Schutzaspekten im Zusammenhang mit Konzepten zur Konstruktion zuverlässiger SW-Systeme im Vordergrund

Ziel der bisherigen Ausführungen über den Themenkreis Datenschutz/ -sicherheit war, einen groben Überblick über die Aufgaben, die möglichen Mechanismen zur Lösung dieser Aufgaben und die gängigsten Methoden dafür zu geben. Der Überblick sollte zum einen eine Einführung in die Problematik geben, zum anderen aber auch eine Abgrenzung des Themas Schutz innerhalb dieser Arbeit ermöglichen.

Abschließend soll nun an dieser Stelle eine Übersicht über die Historie der beiden hier behandelten Bereiche Zugriffs- und Informationsflußkontrolle gegeben werden, soweit dies nicht in Kapitel 4 bei der genaueren Vorstellung der einzelnen Modelle geschieht.

Im Jahre 1966 lieferten Dennis und van Horn mit <Den66> den ersten Beitrag zum Thema Schutzsysteme, der allerdings noch sehr stark am Thema Speicherschutz orientiert war. Der zentrale Begriff in dieser Arbeit war die sogenannte "sphere of protection" (SOP): Jede Berechnung läuft in einer bestimmten SOP ab; eine SOP wird dabei durch eine Liste von "capabilities" spezifiziert. Jede capability besteht aus

dem Namen eines Objekts und den erlaubten Zugriffen auf dieses Objekt, eine capability ist also die Zusammenfassung von Rechten bezogen auf Subjekte. Durch die capabilities der SOP wird somit eine Beschränkung des Zugriffs von Berechnungen in dieser SOP erreicht.

Der nächste Beitrag zu diesem Thema stammte dann im Jahre 1968 von Graham. In seiner Arbeit <Gra68>, die auch noch speicherschutzorientiert ist, definiert Graham sogenannte "Schutzringe", eine geordnete disjunkte Menge von Segmenten 0 ... max; je niedriger die Ringnummer ist, desto höher sind dabei die Zugriffsprivilegien. Der eigentliche Zugriff auf die Segmente findet dabei über "Segment-Deskriptoren" statt, die aus den Komponenten Segmentbeginn, Segmentlänge und Zugriffsanzeigen bestehen.

In den Jahren 1969 bis 1972 entstanden die ersten Ansätze für formale Schutzmodelle. Aufbauend auf Arbeiten von Lampson (<Lam69> und <Lam71>) präsentierten Graham und Denning in <Gra72> die erste formale Darstellung eines Schutzsystems, bestehend aus Subjekten, Objekten und einer Zugriffsmatrix; zur Manipulation der Zugriffsmatrix wurden von den beiden Autoren explizit Regeln formuliert. Das Modell von Graham und Denning gilt als der Vorläufer der heute auf dem Gebiet des Zugriffsschutzes grundlegenden Modelle von Harrison (vgl. 4.1.1).

Bis hierher beschäftigten sich die Arbeiten zum Thema Schutz bzw. Sicherheit ausschließlich mit dem Gebiet Zugriffsschutz, wobei die ersten Arbeiten mehr als eine Erweiterung des bekannten Speicherschutzes anzusehen sind. Wieder war es Lampson, der 1973 in seinem berühmten Artikel "A Note on the Confinement Problem" <Lam73> zum ersten Mal Probleme der Informationsflußkontrolle erkannte und formulierte. Unabhängig davon entstand ebenfalls 1973 das erste theoretische Modell zur militärischen Sicherheit, das Bell-LaPadula-Modell (vgl. <Bel73> und 4.2.1).

Das erste Betriebssystem mit dem Schwerpunkt Datenschutz/ -sicherheit, das System HYDRA, wurde dann im Jahre 1974 erstellt (vgl. <Coh75>). Der Schwerpunkt lag bei HYDRA auf dem Gebiet des Zugriffsschutzes; hierfür stellte das System verschiedene Mechanismen zur Verfügung, wie z.B. eine Unterscheidung der Objekte durch ihren Typ, eine Zugriffskontrolle zu Objekten durch capabilities und eine konsequente Einhaltung des "need-to-know"-Prinzips (Vergabe von nur soviel Zugriffsrechten wie unbedingt nötig) und des Prinzips des "information-hiding" (Jede Kenntnis über die Repräsentation und Implementierung von Objekten und Operationen ist in Modulen bzw. Subsystemen verborgen). Mit Methoden des Objektschutzes, durch das Verteilen von Information in verschiedene Objekte, konnte sogar elementarer Informationsschutz realisiert werden.

Im Jahre 1976 schließlich erstellten Harrison, Ruzzo und Ullman ein formales Modell für Zugriffsschutz und leiteten in diesem Modell wichtige Aussagen über die Entscheidbarkeit der Sicherheit eines Systems ab (vgl. <Har76> und 4.1.1). Mit dieser Arbeit war die erste Phase in der Entwicklung der Themengebiete Datenschutz und Datensicherheit beendet. Inzwischen gibt es viele neue Ansätze auf diesem Gebiet, vor allem im Bereich Informationsflußkontrolle. Die weitere Entwicklung soll daher bei der detaillierten Betrachtung der einzelnen Modelle in Kapitel 4 dieser Arbeit vorgestellt werden.

3.5.3 Anforderungen an ein Modell zur Objektverwaltung

In den vorhergehenden Abschnitten wurden Synchronisation und Schutz als Aufgaben der Objektverwaltung behandelt. Aufgabe der Synchronisation war dabei die Koordinierung des Zugriffs auf Objekte, d.h. die Verzögerung des Zugriffs, bis gewisse Bedingungen erfüllt sind; diese Bedingungen konnten z.B. abhängig sein vom Typ des Objekts, von dessen

Wert, vom Wert der Aufrufparameter und von der gewählten Verwaltungsstrategie. Die Aufgabe des Gebiets Schutz besteht - unabhängig, ob es sich um die Teilgebiete Zugriffskontrolle, militärische Sicherheit oder Informationsflußkontrolle handelt - im Schutz von Objekten vor irgendwelchen "unerlaubten" Zugriffen, wobei solche unerlaubten Zugriffe zurückgewiesen werden müssen. Auch hier ist ein unerlaubter Zugriff dadurch charakterisiert, daß gewisse Bedingungen, die von den Regeln zum Zugriff auf die Objekte abhängen - und damit letztendlich auch vom Objekttyp, Objektwert, den Aufrufparametern und der gewählten Strategie -, nicht erfüllt sind.

Hierbei fällt eine grosse Ähnlichkeit dieser beiden Objektverwaltungsaufgaben auf: In beiden Fällen versucht ein Subjekt, auf ein Objekt zuzugreifen, und ein Mechanismus stellt aufgrund ihm vorgegebener Regeln fest, ob gewisse Bedingungen erfüllt sind, und damit der gewünschte Zugriff erlaubt, verzögert oder zurückgewiesen wird. Es gibt eigentlich nur zwei kleine Unterschiede: Bei der Synchronisation betreffen die Bedingungen die Ausfuhrung mehrerer Operationen (beim Schutz nur die Ausführung einer Operation) und beim Schutz wird der Zugriffswunsch entweder erlaubt oder zurückgewiesen ("jetzt oder nie"), während bei der Synchronisation auch die Möglichkeit der Verzögerung um eine gewisse Zeit besteht ("jetzt oder später").

Diese Zusammenhänge und Gemeinsamkeiten zwischen Synchronisation und Schutz wurden bisher kaum erkannt. Die beiden Problemkreise wurden immer in - schon von der Konzeption her - völlig unterschiedlichen Modellen getrennt voneinander behandelt. Eine solche Betrachtungsweise erscheint jedoch sehr unrealistisch, da sie einer adäquaten Behandlung von Objektverwaltungsproblemen nicht gerecht wird:

- Beides sind Mechanismen zur Objektverwaltung, die sich nicht gegenseitig ersetzen können. Deshalb ist jedes Modell, das nur einen der beiden Aspekte behandelt, im Sinne eines Gesamtkonzepts für die Objektverwaltung unvollständig.

- Es gibt sehr wenige reale Probleme, die reine Synchronisations- oder Schutzprobleme sind. Dies zeigt sich schon am sehr einfachen Beispiel des 5-Philosophen-Problems (vgl. <Ker82>).

- Synchronisations- und Schutzmechanismen sind im engen Zusammenhang mit Methoden zur Konstruktion zuverlässiger SW zu sehen, sie bedingen sich gegenseitig: Ohne Synchronisation- und Schutzmechanismen ist keine zuverlässige Software möglich, da ja die Konsistenz von Objekten nicht gewährleistet werden kann, und ohne Methoden zur Konstruktion zuverlässiger Software ist keine Synchronisation und Schutz realisierbar, da keine zuverlässige Basis zur Verfügung stünde.

Natürlich müssen in dieser Arbeit, in der ja ein allgemeines Modell für Objektverwaltungsaufgaben entwickelt werden soll, Synchronisation und Schutz gemeinsam und einheitlich behandelt werden. Die weiteren an das Modell zu stellenden Anforderungen entsprechen den in 3.3 erläuterten SW-Entwurfsprinzipien und sind durch den Einsatz des ADT-Konzepts im vorliegenden Modell automatisch erfüllt. Hier sollen diese Anforderungen noch einmal stichpunktartig aus der Sicht der Objektverwaltung dargestellt werden.

- Einheitliches Konzept zur Realisierung von OV

  Da die drei Aufgaben allgemeine Objektverwaltung, Synchronisation und Schutz untrennbar miteinander verbunden sind, stellt sich die Forderung nach einem einheitlichen Konzept. Insbesondere für das Gebiet Schutz ergibt sich die Notwendigkeit, in diesem Modell Zugriffskontrolle und Informationsflußkontrolle zu berücksichtigen. Informationsfluß-Kontrollmechanismen sind vollständiger als Zugriffskontrollmechanismen, d.h. es gibt Schutzprobleme (wie z.B. das Confinement-Problem), die alleine mit Zugriffskontrollmechanismen nicht lösbar sind. Andererseits sind viele einfache Zugriffskontrollprobleme mit Informationsfluß-Kontrollmechanismen nur ineffektiv lösbar. Zudem gibt es auch Schutzprobleme, die sich allein mit einer Informationsflußkontrolle nicht lösen lassen; hierunter fallen alle mit der Datenintegrität zusammenhängenden Aufgaben, wie z.B. der Schutz vor unerlaubter Modifikation oder Zerstörung.

- Abstraktions- und Spezifikationsmöglichkeiten

  Gerade für ein Modell zur Objektverwaltung spielt die Möglichkeit der Abstraktion eine wichtige Rolle. Erst die Trennung zwischen Spezifikation und Implementierung sowie die Einteilung des Systems im mehrere Abstraktionsebenen garantiert das Maß an Flexibilität, das bei der Lösung von OV-Problemen gegeben sein muß. Das Modell muß gestatten, daß durch den OV-Mechanismus die unterschiedlichsten OV-Strategien durchgesetzt werden können. Die Spezifikation dieser OV-Strategien geschieht in nichtprozeduraler Form, z.B. durch Relationen oder Prädikate, und muß eine adäquate Formulierung von OV-Strategien ermöglichen. Hierbei müssen für jeden Objekttyp wertabhängige Formulierungen möglich sein, d.h. die Abhängigkeit vom Wert des Objekts und dem Wert der Eingabe- und Ausgabeparameter. Auch die Möglichkeit der Einschränkung des Zugriffs auf Teilkomponenten des Objekts (z.B. nur

bestimmte Sätze einer Datei, ...) muß gegeben sein.

- Verifizierbarkeit

Beim Schutz von Objekten spielt der Begriff Sicherheit eine entscheidende Rolle. Sicherheit bedeutet die Einhaltung der spezifizierten Regeln (policy) durch den OV-Mechanismus. Der Begriff Sicherheit läßt sich auf das Gebiet der Synchronisation in der gleichen Weise anwenden. Es genügt jedoch für ein System nicht allein, sicher zu sein; von entscheidender Wichtigkeit ist vielmehr die Beweisbarkeit seiner Sicherheit. Man beachte, daß es sich bei der hier angesprochenen Verifikation von Sicherheit um den Nachweis invarianter Systemeigenschaften handelt, die implementierungsunabhängig sind. Im Abschnitt 3.3.6 wurde dies unter dem Begriff design verification eingeführt.

- Lokalität und Modifizierbarkeit

Für ein Modell zur Objektverwaltung ist die Einhaltung des Lokalitätsprinzips eine wichtige Voraussetzung, da es sehr stark zur leichteren Modifizierbarkeit und Verifizierbarkeit der spezifizierten OV-Strategien beiträgt. Für die Objektverwaltung bedeutet dieses Prinzip, daß die Strategien zur Verwaltung des Objekts im Objekt selbst definiert sind, und die eigentliche Objektverwaltung in der dem Objekt zugeordneten abstrakten Maschine stattfindet; sämtliche Verwaltungsdaten müssen also im Objekt enthalten sein. Die Spezifikation der OV-Strategie sollte außerhalb der eigentlichen Operationen möglich sein, um die Modifizierbarkeit von OV-Strategien noch weiter zu erhöhen.

Durch den Einsatz des um Synchronisations- und Schutzaspekte erweiterten ADT-Konzepts als Grundlage für das zu erstellende OV-Modell werden diese Anforderungen alle er-

füllt. Bevor hier allerdings die Konstruktion des formalen OV-Modells präsentiert wird, sollen im nun folgenden Kapitel bisher existierende Schutzmodelle vorgestellt werden und anhand der in diesem Abschnitt entwickelten Anforderungen an ein OV-Modell einer kritischen Betrachtung unterworfen werden.

4 Formale Modelle zur Lösung von Schutzproblemen

Im Kapitel 3 dieser Arbeit wurde das Gebiet Schutz und Sicherheit als eines der notwendigen Mittel zur Objektverwaltung vorgestellt. Die Schwerpunkte lagen hier bei Schutz vor unerlaubter Enthüllung, Modifikation oder Zerstörung von Information. In diesem Zusammenhang wurden auch die heute üblichen Methoden zum Erreichen dieser Ziele kurz präsentiert:

- Zugriffskontrolle durch Objektschutzmethoden
- Informationsschutz durch Objektschutzmethoden (militärische Sicherheit)
- Informationsflußkontrolle

Ziel dieses Kapitels ist eine Vertiefung der Betrachtungsweise des Themenkreises Schutz und Sicherheit. Hierzu werden die wichtigsten existierenden Modelle vorgestellt und an den im Abschnitt 3.5.4 entwickelten Anforderungen an ein OV-Modell gemessen.

4.1 Objektschutzmodelle

In Objektschutzmodellen steht der Schutz von Objekten vor unerlaubten Zugriffen von Subjekten im Vordergrund. Die Entscheidung, ob ein Zugriff erlaubt ist, wird hier unabhängig von der Art der Information, die das Objekt enthält, gefällt. Als Grundlage für diese Entscheidungen sind in einem Objektschutzmodell gewisse Regeln gegeben, die die Möglichkeiten der Zugriffe von Subjekten auf Objekte im allgemeinen einschränken. Um unerlaubte Zugriffe verhindern zu können, müssen in Systemen, die sich an Objektschutzmodellen orientieren, Zugriffsmonitore existieren, die die Zugriffsberechtigung prüfen und den Zugriffswunsch erlauben oder zurückweisen. Eine Umgehung dieses Monitors, d.h. ein direkter Zugriff eines Subjekts auf ein Objekt, muß natürlich ausgeschlossen sein.

An dieser Stelle soll nicht auf die verschiedenen möglichen Implementierungsarten solcher Monitore eingegangen werden, stehen doch in dieser Arbeit mehr die grundlegenden Aspekte von Modellen zur Objektverwaltung im Vordergrund.

Objektschutzmodelle sind die grundlegendsten und ältesten Modelle zum Erreichen von Sicherheit in Betriebssystemen. Bis heute wurden schon sehr viele - teilweise auch sehr verschiedene - Objektschutzmodelle vorgeschlagen, von denen einige auch in Betriebssystemen implementiert wurden. Alle diese Ansätze basieren jedoch auf einem gemeinsamen Grundmodell - dem Zugriffsmatrix-Modell. Aus diesem Grund wird die Vorstellung und Diskussion dieses Modells einen breiten Raum in diesem Abschnitt einnehmen. Als ein Beispiel für ein auf dem Zugriffsmatrix-Modell aufbauendes Modell wird der von Popek an der University of California at Los Angeles (UCLA) spezifizierte, realisierte und verifizierte Sicherheitskern für das Betriebssystem UNIX präsentiert. Weitere Ansätze, wie die Take-Grant-Modelle oder der Send-Receive-Mechanismus von Minsky werden anschließend kurz vorgestellt.

4.1.1 Das Zugriffsmatrix-Modell

Die ersten Ansätze zu einem formalen Modell für den Zugriffsschutz stammen von Lampson <Lam71>. Er definiert als Grundkomponenten für ein Schutzmodell eine Menge von Objekten, eine Menge von Domänen (die Größen, die auf Objekte zugreifen) und eine Zugriffsmatrix. Die Zugriffsmatrix enthält den (momentanen) Stand der Zugriffsberechtigungen der Domänen auf Objekte:

Die Zeilen der Matrix bilden die Domänen, die Spalten die Objekte. Das (i,j)-te Element der Matrix spezifiziert nun die erlaubten Zugriffe (Rechte) der Domäne i auf das Objekt j. Jedes Matrixelement enthält eine Menge von Strings, die sogenannten Zugriffsattribute (z.B. "read", "write", "execute", ...). Lampson definiert für jedes Zugriffsattribut eine "copyflag", die angibt, ob das Attribut kopiert werden darf, d.h. an andere Domänen weitergegeben werden darf. Im System existiert noch ein spezielles Attribut "owner", das den Besitz eines Objekts charakterisiert.

Die Veränderung der Zugriffsmatrix (z.B. durch die Weitergabe von Rechten) geschieht durch bestimmte Regeln. Lampson gibt in seiner Arbeit nur Beispiele, wie die Regeln aussehen können; da die möglichen Regeln nicht näher spezifiziert werden, kann Lampson auch auf den Begriff Sicherheit nicht eingehen. Stattdessen versucht der Autor, verschiedene Implementierungsmöglichkeiten für seinen Ansatz anzugeben.

Aufbauend auf den Ausarbeitungen von Lampson erweitern Graham und Denning in <Gra72> das Zugriffsmatrix-Modell. Sie ordnen jedem Objekttyp einen Monitor zu. Jeder Zugriffsversuch eines Subjekts (Domäne) auf ein Objekt wird vom, dem Objekttyp zugeordneten, Monitor durch Lesen des entsprechenden Elements der Zugriffsmatrix überprüft. Veränderungen der Zugriffsmatrix selbst laufen über den Zugriffsmatrix-Monitor. Für die Änderung der Zugriffsmatrix

definieren Graham und Denning 8 Regeln. Sie erlauben das Kreieren und Zerstören von Subjekten und Objekten und die Weitergabe bzw. den Rückruf von Rechten. Die Autoren unternehmen in ihrer Arbeit auch den Versuch, den Begriff Sicherheit zu definieren:

- Aktionen eines Subjekts, die den Schutzzustand des Systems nicht ändern, dürfen nie nichtautorisierte Zugriffe sein
- Aktionen eines Subjekts, die den Schutzzustand des Systems ändern, dürfen nicht zu einem neuen Schutzzustand führen, in dem irgendein Subjekt nichtautorisierten Zugriff zu irgendeinem Objekt hat

Die Definition wird von Graham und Denning leider nur in dieser informellen Darstellung gegeben. Dies hat zur Konsequenz, daß in diesem Modell keine Aussagen über die Beweisbarkeit von Sicherheit getroffen werden können. Aus der weiteren Diskussion wird zudem klar, daß der hier verwendete Sicherheitsbegriff sehr stark auf dem Vorhandensein "vertrauenswürdiger Subjekte" aufbaut. Es soll an dieser Stelle nicht weiter auf den Sicherheitsbegriff von Graham und Denning eingegangen werden. Aus dem Gesagten sollte nur deutlich werden, daß hier keine klare Definition von Sicherheit im Sinne unserer Anforderungen an ein OV-Modell vorliegt.

Die Ansätze von Lampson und Graham und Denning waren die Basis für eine formale Darstellung des Zugriffsmatrix-Modells von Harrison, Ruzzo und Ullman <Har76>. In dieser Darstellung wurde der Begriff "Sicherheit" zum ersten Mal formal definiert und Aussagen über die Beweisbarkeit von Sicherheit in Zugriffsmatrix-Modellen getroffen. Aus diesem Grund soll das Modell von Harrison hier detaillierter vorgestellt und einer eingehenden Betrachtung unterworfen werden.

Modellkomponenten

Das Zugriffsschutz-Modell von Harrison besteht aus einer Menge von Subjekten, einer Menge von Objekten und einer Menge von Zugriffsattributen von Subjekten auf Objekte als Basiskomponenten. Eine Konfiguration des Modells (momentaner Schutzzustand) ist ein Tripel (S,O,M), wobei S die Menge der aktuellen Subjekte, O die Menge der aktuellen Objekte ($S \subseteq O$) und M die aktuelle Zugriffsmatrix mit einer Zeile pro Subjekt und einer Spalte pro Objekt ist. Das (i,j)-te Element M[i,j] gibt die aktuellen Rechte des i-ten Subjekts am j-ten Objekt an, ist also eine Teilmenge der Menge von Zugriffsattributen. Zur Veränderung der momentanen Konfiguration stehen eine endliche Menge von Kommandos (Regeln des Schutzsystems) zur Verfügung. Anders als bei Graham und Denning gibt Harrison keine feste Menge von Regeln vor, sondern nur die Form der Kommandos, d.h. die erlaubte Syntax der Regeln:

```
command α (X1,X2,...,Xk)
  if r1 in (Xs1,Xo1) and
     r2 in (Xs2,Xo2) and
     ...................
     rm in (Xsm,Xom)
  then op1; op2; ...; opn
end
```

Hier ist α der Name des Komandos und X1,...,Xk sind formale Parameter für Objekte bzw. Subjekte; r1,...,rm sind Zugriffsattribute, s1,...,sm und o1,...,om sind ganze Zahlen zwischen 1 und k. Jedes opi ist eine der Grundoperationen

```
enter r into (Xs,Xo)          delete r from (Xs,Xo)
create subject Xs             destroy subject Xs
create object  Xo             destroy object  Xo
```

auf die Zugriffsmatrix. Das Prädikat nach "if" heißt Bedin-

gung für α , die Folge von Grundoperationen op1,...,opn der Rumpf des Kommandos α . Die Kommandos sind folgendermassen zu interpretieren: Falls sämtliche Einzelbedingungen "rj in (Xsj,Xoj)" (d.h. das Subjekt Xsj hat das Recht rj auf das Objekt Xoj) erfüllt sind, wird die Folge der Grundoperationen op1,...,opn ausgeführt.

Ein konkretes Beispiel soll dies verdeutlichen: Jedes Subjekt sei ein Prozeß, und die übrigen Objekte (außer den Subjekten) seien Dateien. Jede Datei habe als Besitzer genau einen Prozeß. Außer dem Recht "own" gebe es noch die Zugriffsattribute "read", "write" und "execute". Die Subjekte können folgende Aktionen, die eine Veränderung der Zugriffsmatrix bewirken, ausführen:

(i) Erzeugen einer neuen Datei

```
command CREATE (process,file)
  create object file
  enter own into (process,file)
end
```

(ii) Übertragung von Zugriffsattributen (außer own) an ein beliebiges anderes Subjekt

```
command CONFER r (owner,friend,file)
  if own in (owner,file)
  then enter r into (friend,file)
end
```

wobei r ε {read,write,execute}

(iii) Rückruf von Zugriffsattributen durch der Besitzer einer Datei

```
command REMOVE r (owner,exfriend,file)
  if own in (owner,file) and
       r  in (exfriend,file)
  then delete r from (exfriend,file)
end

wobei r ε {read,write,execute}
```

Dieses Beispiel wird im folgenden zur Erläuterung der Wirkungsweise des Modells benutzt.

Zusammenspiel der Grundkomponenten

Die Wirkung der Kommandos ergibt sich aus ihrem Effekt auf die Zugriffsmatrix. Harrison definiert hierzu erst die Zustandsübergänge aufgrund der sechs Grundoperationen enter, delete, create subject/object, destroy subject/object. Dies wird auf Kommandos erweitert:
Q $\vdash$-- α (x1,...,xk) Q' bedeutet, die Konfiguration Q geht durch das Kommando α mit den aktuellen Parametern x1,...,xk in Q' über. In der üblichen Weise werden darauf aufbauend Q $\vdash$--α Q', Q $\vdash$-- Q' und Q $\vdash$-- * Q', wobei $\vdash$-- * die reflexive und transitive Hülle von $\vdash$-- ist, definiert. Es soll hier auf eine formale Darstellung verzichtet werden (siehe <Har76> pp.463-464), sondern stattdessen die Wirkungsweise der Kommandos am angegebenen Beispiel verdeutlicht werden:

Die Anfangskonfiguration bestehe aus den zwei Prozessen P1 und P2, und keinen Dateien. Kein Prozeß soll auf sich selbst oder den anderen Prozeß ein Zugriffsrecht haben. Dies ergibt den folgenden Anfangszustand:
({P1,P2},{P1,P2},Mo) mit

Mo:	P1	P2
P1	0	0
P2	0	0

P1 erzeuge nun zwei Dateien "code" und "daten" und übergebe P2 die Rechte "execute" für "code" und "read" für "daten". Die erforderliche Sequenz von Kommandos ist

```
CREATE (P1,code)
CREATE (P1,daten)
CONFER execute (P1,P2,code)
CONFER read    (P1,P2,daten)
```

Die Wirkung auf die Zugriffsmatrix kann folgendermaßen dargestellt werden

	P1	P2
P1	0	0
P2	0	0

⊢--

	P1	P2	code
P1	0	0	{own}
P2	0	0	0

⊢--

	P1	P2	code	daten
P1	0	0	{own}	{own}
P2	0	0	0	0

⊢--

	P1	P2	code	daten
P1	0	0	{own}	{own}
P2	0	0	{exe}	0

	P1	P2	code	daten
P1	0	0	{own}	{own}
P2	0	0	{exe}	{read}

Der entscheidende Vorteil des Modells von Harrison gegenüber den Vorläufermodellen von Lampson bzw. Graham und Denning ist die Möglichkeit, den Begriff Sicherheit klar definieren zu können.

Harrison unterscheidet zwei Arten von Sicherheit: Er bezeichnet zunächst ein Schutzsystem als sicher, wenn der Zugriff zu Objekten ohne das Zusammenspiel mit dem Besitzer unmöglich ist. Wegen der Möglichkeit der Weitergabe von Rechten an einem Objekt existiert aber im allgemeinen kein sicheres Schutzsystem in diesem Sinn. Die Bedingung für Sicherheit muß also eingeschränkt werden. Harrison schlägt vor, daß als minimale Bedingung für Sicherheit ein Subjekt zumindest in der Lage sein müßte, zu bestimmen, ob von ihm weitergegebene Rechte an seinen eigenen Objekten weiter zu "nicht vertrauenswürdigen" Subjekten "durchsickern" können; er definiert:

- Ein Kommando α (X1,...,Xk) läßt ein Recht r im Zustand Q:(S,O,M) durchsickern, wenn α angewandt auf Q das Recht r in eine Position der Matrix einträgt, in der es im Zustand Q nicht war.

- Ein Anfangszustand Qo:(So,Oo,Mo) eines Schutzsystems heißt sicher für ein Recht r, wenn von Qo kein Zustand erreichbar ist, der r durchsickern läßt.

Gemäß dieser Definition ist natürlich der Anfangszustand eines Systems, in dem der Besitzer von Objekten Rechte an diesen weitergeben kann, nicht sicher für diese Rechte. Dieser triviale Fall von "Unsicherheit" ist jedoch nicht von Interesse. Man wird also vor der Untersuchung, ob ein Zustand für ein bestimmtes Recht sicher ist, den Besitzer des betreffenden Objekts und eventuell auch "zuverlässige" Subjekte, denen der Besitzer vertraut, aus der Matrix entfernen. Damit können z.B. solche Subjekte überprüft werden, denen der Besitzer eines Objekts mißtraut.

Harrison geht nun der Frage nach, ob in einem beliebigen System die Sicherheit eines Zustands gemäß seiner Definition entscheidbar ist. Leider ist dies nicht der Fall, d.h. es existiert kein Algorithmus, der die Sicherheit eines beliebigen Systems entscheiden könnte. Selbst die Hoffnung, für jedes System einen speziellen Algorithmus finden zu können, der dessen Sicherheit entscheidet, mußte begraben werden. Harrison beweist diese beiden Sätze durch eine Simulation der Turing-Maschine (bzw. universellen Turing-Maschine) in einem Schutzsystem. Das Sicherheitsproblem eines Schutzsystems wird dann durch die Äquivalenz mit dem Halteproblem der Turing-Maschine als unentscheidbar bewiesen.

Da die Sicherheitsfrage für allgemeine Schutzsysteme nicht entscheidbar ist, betrachtet Harrison in <Har76> und <Har78> eingeschränkte Klassen von Schutzsystemen:

- Ein Schutzsystem heißt mono-operational, wenn der Rumpf jedes Kommandos aus einer einzigen Grundoperation besteht. Für mono-operationale Schutzsysteme ist die Sicherheitsfrage entscheidbar <Har76>.

- Die Sicherheitsfrage ist für Schutzsysteme ohne "create"-Grundoperationen entscheidbar <Har76>.

- Die Sicherheitsfrage ist für Schutzsysteme mit einer endlichen Anzahl von Subjekten entscheidbar <Har78>. Lipton und Snyder zeigen sogar, daß solche Schutzsysteme rekursiv äquivalent zu Vektoradditionssystemen sind <Lip78>.

- Ein Schutzsystem heißt monoton, wenn keine Grundoperationen der Form "delete" und "destroy" vorkommen. Auch für monotone Schutzsysteme ist die Sicherheitsfrage nicht entscheidbar <Har78>. Selbst wenn man die Zahl der Bedingungen pro Kommando in einem monotonen System auf maximal zwei beschränkt, bleibt die Sicherheitsfrage unentscheidbar <Har78>.

- Ein Schutzsystem heißt monoconditional, wenn jedes Kommando höchstens eine Bedingung enthält. Es ist noch unbekannt, ob die Sicherheitsfrage für monoconditionale Schutzsysteme entscheidbar ist <Har78>. Die Sicherheitsfrage ist jedoch für monotone, monoconditionale Schutzsysteme entscheidbar <Har78>. Auch für monoconditionale Schutzsysteme mit "create", "enter" und "delete" , aber ohne "destroy" Grundoperationen ist die Sicherheitsfrage entscheidbar <Har78>.

Nach dieser Vorstellung des Zugriffsmatrix-Modells und insbesondere des Modells von Harrison sind natürlich die Vor- und Nachteile und die Frage nach der Anwendbarkeit von Interesse. In dieser Arbeit interessiert daneben besonders, inwieweit die Anforderungen an ein OV-Modell vom Zugriffsmatrix-Modell erfüllt werden.

Bemerkungen zum Modell

Das Modell von Harrison ist so allgemein gehalten, daß sich praktisch jedes Objektschutzsystem, das mit einer Zugriffsmatrix oder einer äquivalenten Darstellung arbeitet, in dieses Modell einordnen läßt. Dies und die formale Darstellung im Modell machen die Bedeutung des Modells - vor allem für die Erörterung theoretischer Fragen - aus. Das allgemeine Zugriffsmatrix-Modell ist auch die Basis für viele erweiterte Modelle, z.B. sämtliche in diesem Abschnitt vorgestellten Objektschutzmodelle. Wegen der relativ leichten Implementierbarkeit einer Zugriffsmatrix und des dazugehörigen Schutzmechanismus wurde das Modell schon in realisierten Schutzsystemen berücksichtigt (besonders bei Systemen, die auf dem capability-mechanismus aufsetzen; vgl. <Coh75>). In Systemen, die auf dem Zugriffsmatrix-Modell basieren, liegt der Schutz eines informationsenthaltenden Objekts im allgemeinen in der Verantwortung des Besitzers (discretionary model). Deshalb ist es - ohne Einschränkungen im Modell zu machen - nicht möglich, Aussagen über den Fluß von Information zu treffen.

Harrison betrachtet in seinem Modell ausschließlich Veränderungen der Zugriffsmatrix. Fragen, die den eigentlichen Zugriff von Subjekten auf Objekte betreffen, werden nicht angesprochen. Der Sicherheitsbegriff von Harrison bezieht sich deshalb nur auf die Weitergabe von Zugriffsrechten und nicht auf den eigentlichen Schutz der Objekte. Der in dieser Arbeit definierte Sicherheitsbegriff, nämlich die Garantie, daß durch einen Schutzmechanismus eine gegebene Menge von Regeln (policy) eingehalten wird, ist wesentlich umfassender. Von Harrison wird weder der Begriff policy, noch ein Schutzmechanismus, der die Zugriffe der Subjekte auf Objekte regelt, angesprochen. In den Arbeiten von Lampson und Graham und Denning sind durch die Monitore für die einzelnen Objekttypen in gewisser Weise Schutzmechanismen gegeben; sie überwachen den Zugriff der Subjekte auf die jeweiligen Objekte; die durchzusetzende policy ist dabei der momentane Stand der Zugriffsmatrix. Bei dieser Interpretation von "policy" kann diese allerdings durch jedes Subjekt, das durch Kommandos die momentane Zugriffsmatrix ändern kann, ebenfalls geändert werden. Diese Definition von policy widerspricht derjenigen dieser Arbeit: Hier wird unter policy eine Menge von - für jeden Objekttyp frei vorgebbaren - Regeln für Zugriffseinschränkungen auf Objekte dieses Typs verstanden, die nach ihrer Spezifikation für diesen Typ fest definiert sind. Nur bei dieser Interpretation läßt sich in einem Modell die Sicherheit - nämlich die Einhaltung der gegebenen Regeln durch den Schutzmechanismus - nachweisen.

Die im Zugriffsmatrix-Modell vorgebbaren Zugriffseinschränkungen hängen vom Subjektnamen, Objektnamen und den Rechten, die das Subjekt auf das Objekt hat, ab. Solche - nicht wertabhängigen - Strategien ermöglichen nur eine grobe Definition des Zugriffsschutzes. Auf diese Weise sind "content-dependent-policies", d.h. policies, die vom Inhalt des Objekts (dessen Wert) abhängen, nicht definierbar. Nicht zuletzt steht das Zugriffsmatrix-Modell im Widerspruch zum Lokalitätsprinzip, da die Entscheidungen über den Zugriff auf ein Objekt nicht in der örtlichen Nähe des Ob-

jekts, sondern global im System, gefällt werden.

Die vorstehenden Ausführungen sollten zeigen, daß das Zugriffsmatrix-Modell, um als formales Modell für Zugriffsschutz zu dienen, aber auch um die Basis für konkrete Implementierungen zu sein, noch in einigen Punkten erweitert und mit Leben erfüllt werden muß. Nichtsdestoweniger sind gerade die Resultate von Harrison bezüglich der Entscheidbarkeit von Sicherheit in seinem Sinn sehr interessant für weitere Forschungen auf diesem Gebiet.

4.1.2 Das UCLA-Modell von Popek

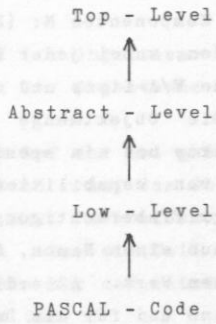
Ziel der Arbeiten, die unter der Leitung von Popek an der University of California at Los Angeles (UCLA) durchgeführt wurden, war die Entwicklung von Mitteln, die von einem Betriebssystem "Datensicherheit" beweisen können (<Pop78>, <Pop79>, <Wal80>). Dies erforderte zum einen die Entwicklung eines Modells und einer Systemarchitektur, die helfen sollte, den SW-Aufwand für Schutzentscheidungen und die Durchsetzung von Schutz zu reduzieren; durch die Verlagerung der Aufgaben in den Betriebssystemkern und die Trennung zwischen policy und Mechanismus wurde dies unterstützt. Zum anderen mußten Verifikationsmethoden entwickelt werden, um die Datensicherheit beweisbar zu machen.

Konkreter Anwendungsbereich der Arbeiten war das Design, die Spezifikation, Implementierung und Verifikation eines Sicherheitskerns für das Betriebssystem UNIX. Die wichtigste Aufgabe in diesem Zusammenhang war eine klare Definition von Sicherheit, um die Grundlage für die Verifizierbarkeit von Sicherheit zu schaffen.

Das Modell von Popek basiert im Grunde auf dem Zugriffsmatrix-Modell: Entscheidungen über die Erlaubnis eines Zugriffs werden aufgrund einer Capability-Liste, die gewissermaßen die Implementierung der Zugriffsmatrix dar-

stellt, getroffen. Allerdings ist der Blickwinkel bei Popek völlig anders als bei Harrison: Während Harrison die Veränderungen der Zugriffsmatrix z.B. durch die Weitergabe von Rechten betrachtet, interessiert beim Konzept der Datensicherheit (data security) von Popek ausschließlich der eigentliche Zugriff der Subjekte auf die Objekte. Dies hat natürlich auch eine unterschiedliche Interpretation des Sicherheitsbegriffs zur Folge, die aber dem Verständnis in dieser Arbeit näher ist. Das Modell enthält keine spezielle policy, und es werden auch keine Regeln für den Aufbau und die Veränderung der Zugriffsmatrix gegeben. Popek räumt jedoch ein, daß jede beliebige policy auf einfache Art und Weise in das Modell eingebettet werden kann. Ansatzpunkt hierfür ist der sogenannte "policy manager", der die Zugriffsberechtigung abcheckt.

Der Sicherheitskern des UCLA-Betriebssystems wurde in vier Abstraktionsebenen spezifiziert:



Die Top-Level Spezifikation entspricht dem formalen Modell und wird als abstrakte Maschine dargestellt. Die Abstract-Level Spezifikation ist eine bestimmte Interpretation der abstrakten Maschine: die Kern-Operationen werden festgelegt, die Objekte werden feiner strukturiert. Die hier näher spezifizierten Objekte und Operationen werden dann in der Low-Level Spezifikation noch detaillierter ausgearbeitet. Die eigentliche Implementierung findet in PASCAL-Code

statt.

Die Verifikation dieses Sicherheitskerns ist das Hauptanliegen der Arbeiten von Popek. Hierzu muß die korrekte Implementierung der jeweils höheren Ebene durch die nächstniedrige Ebene gezeigt werden. Dabei werden die bekannten Verifikationstechniken von Hoare <Hoa69> und Wulf <Wul76> eingesetzt (vgl. 3.3).

Das Sicherheitskriterium wird von Popek in den Begriffen der abstrakten Maschine des Top-Levels angegeben. In <Pop78> findet sich eine formale Darstellung der abstrakten Maschine, auf die im folgenden näher eingegangen werden soll.

Modellkomponenten

Grundlage des Modells von Popek ist eine abstrakte Maschine. Sie besteht aus vier Komponenten M: (S,so,I,T). S ist hier eine Menge von Zuständen, wobei jeder Zustand durch die momentane Objektmenge, eine E/A-Liste und den aktuellen Prozeß charakterisiert ist. Die Objektmenge enthält die Subjekte, die sonstigen Objekte und ein spezielles Objekt "protection data" (eine Menge von capabilities), das den Schutzzustand, d.h. die Zugriffsberechtigungen enthält. Jedes Objekt besteht wiederum aus einem Namen, dem Typ, dem Speicherort und dem eigentlichen Wert. Allerdings sind auf der Ebene der abstrakten Maschine und für die Definition von Sicherheit im Modell ausschließlich die Existenz von Objekten und die Forderung, daß die Namen der Objekte eindeutig sind, von Interesse.

so ist der Anfangszustand der Maschine, I ist eine Menge von Instruktionen, die bei ihrer Ausführung die Maschine von einem Zustand in einen anderen überführen. Der Effekt der Instruktionen ist durch die Zustandsübergangsfunktion T: I x S -> S gegeben; es handelt sich hier ausschließlich um die Veränderung von Objektwerten, nicht der Zu-

griffsmatrix (d.h. von "protection data"). Die detaillierte Struktur der Maschine wird erst bei ihrer Interpretation durch die Abstract-Level Spezifikation gegeben; sie ist für die Herleitung des Sicherheitsbegriffs nicht nötig.

Zusammenspiel der Grundkomponenten

Um die Sicherheit der Instruktionen beim Zugriff auf Objekte beweisen zu können, braucht man zuerst einen Maßstab, d.h. eine Möglichkeit, anzugeben, wie sich die Instruktionen gemäß Spezifikation - wenn der Zugriff vom System erlaubt wird - verhalten sollen; konkret heißt dies, auf welche Objekte eine bestimmte Instruktion bei einem gewissen Zustand beabsichtigt, zuzugreifen. Zur Spezifikation des beabsichtigten Verhaltens der Instruktionen werden im Modell zwei Relationen Ref (für reference, d.h. lesender Zugriff) und Mod (für modify, d.h. verändernder Zugriff) eingeführt:

Ref: S x I x Objektname -> {true,false}
Mod: S x I x Objektname -> {true,false}

Die Relationen Ref und Mod geben an, ob eine Instruktion i in einem Zustand s auf das Objekt mit Namen n laut Spezifikation lesend bzw. schreibend zugreift, oder nicht. Ref und Mod sind also ein Teil der Spezifikation der Instruktionen und haben mit dem tatsächlichen Effekt einer Instruktion nichts zu tun.

Ein Aspekt des Sicherheitsbegriffs von Popek ist nun die Frage, ob sich die Instruktionen bei ihren Zugriffen auf Objekte wirklich gemäß ihrer Spezifikation verhalten. Hierzu ist es notwendig, zu definieren, wann ein Objekt durch die Ausführung einer Instruktion tatsächlich gelesen oder verändert wird. Popek führt hierfür die "actual access relations" ein. An dieser Stelle soll auf eine formale Darstellung dieser Relationen verzichtet werden, da in Kapitel 5 dieser Arbeit bei der Definition von Abfrage und

Veränderung im formalen Modell näher auf diese Zugriffsdefinitionen eingegangen wird.

Mit den Definitionen, wann ein Objekt in einem Zustand durch eine Instruktion gelesen oder modifiziert wird, ist Popek nun in der Lage, seinen Sicherheitsbegriff zu definieren. Er gibt hierfür drei Sicherheitszusicherungen an, die für jede Interpretation der abstrakten Maschine bewiesen werden müssen:

- Geschützte Objekte dürfen nur durch explizite Nachfrage verändert werden, d.h. ein Objekt darf durch eine Instruktion nur dann geändert werden, wenn diese die Veränderung spezifiziert.
- Geschützte Objekte dürfen nur durch explizite Nachfrage gelesen werden, d.h. ein Objekt darf durch eine Instruktion nur dann gelesen werden, wenn diese das Lesen spezifiziert.
- Zugriffe zu geschützten Objekten sind nur erlaubt, wenn es das Objekt "protection data" gestattet, d.h. auf ein Objekt darf nur zugegriffen werden, wenn die jeweilige policy es erlaubt.

Mit diesen drei Zusicherungen wurde von Popek die Grundlage geschaffen, für den entwickelten Betriebssystemkern "Datensicherheit" zu beweisen.

Bemerkungen zum Modell

Im Modell von Popek spielt der Begriff "Datensicherheit" eine entscheidende Rolle. Wichtigstes Ziel des Modells ist eine klare Definition von Sicherheit, um deren Verifizierbarkeit zu ermöglichen. Durch die Trennung zwischen Spezifikation und Implementierung der Instruktionen wurde eine Basis geschaffen, auf der aufbauend - durch die Verifikation der drei Sicherheitszusicherungen - die Datensicherheit eines Systems nachgewiesen werden kann. Hierbei bringt die Spezifikation eines Systems in mehreren Abstrak-

tionsebenen weitere Erleichterung. Diese Spezifikations-, Abstraktions- und Verifikationsmöglichkeiten tragen wesentlich zur Bedeutung des Modells bei; sie entsprechen auch im wesentlichen den Anforderungen, die in Abschnitt 3.4.4 an ein OV-Modell gestellt wurden. Ein weiteres wichtiges Merkmal ist die Allgemeinheit des Modells. Sie läßt eine Einbettung der unterschiedlichsten Systeme zur Lösung von OV-Aufgaben zu. Dies wird natürlich durch die Möglichkeit, beliebige policies in das Modell einzubringen, zusätzlich unterstützt.

Im Gegensatz zu Harrison betrachtet Popek keine Veränderungen des Objekts "protection data", d.h. die Weitergabe von Rechten oder das Kreieren bzw. Zerstören von Objekten. Popek erwähnt zwar, daß entsprechende Regeln bzw. Operationen leicht in sein Modell eingebettet werden können, er läßt jedoch völlig offen, wie diese Einbettung in den "policy-manager" zu geschehen hat. Insbesondere fehlen an dieser Stelle Ansätze zur Formalisierung von policies bzw. der Regeln zur Veränderung des Objekts protection data. Dies ist deshalb von so grosser Bedeutung, weil durch die Erweiterung der bisher statischen policy (nämlich des Objekts protection data) auf dynamische policies der Sicherheitsbegriff wesentlich beeinflußt wird.

Die Entscheidung, ob ein Zugriff erlaubt oder zurückgewiesen wird, fällt abhängig vom Vorhandensein einer entsprechenden capability im Objekt protection data. Content-dependent-policies sind wie beim Modell von Harrison deshalb nicht darstellbar. Eine weitere Einschränkung des Modells von Popek besteht darin, daß für die Definition von Sicherheit nur die Zugriffstypen "read" und "write" betrachtet werden. Abschließend wäre hier noch zu erwähnen, daß - wie bei Harrison - durch die Entscheidung über die Erlaubnis eines Zugriffs an zentraler Stelle das Lokalitätsprinzip verletzt wird, und Informationsflußaspekte nicht betrachtet werden.

Durch seine Spezifikations-, Abstraktions- und Verifikationsmöglichkeiten eignet sich das Modell von Popek sehr gut als Basismodell für konkrete Betriebssystem-Implementierungen. Hierfür spricht auch, daß das Modell bereits seine erste Bewährungsprobe in der Praxis hinter sich hat. Eine notwendige Ergänzung des Modells ist allerdings die Möglichkeit der Formalisierung von policies.

4.1.3 Weitere Objektschutzmodelle

Neben den bisher vorgestellten Modellen von Harrison und Popek gibt es noch einige Ansätze für Modelle zur Lösung von Zugriffsschutzproblemen. An erster Stelle sind hier die sogenannten "Take-Grant-Modelle" zu nennen, die von mehreren Autoren betrachtet wurden (<Jon76>, <Lip77>, <Jon78a>, <Bis79>, <Sny81>). Ein weiteres Modell ist der Send-Receive-Mechanismus von Minsky (<Min81a> und <Min81b>). Um den Rahmen dieser Arbeit nicht zu sprengen, sollen diese Ansätze jedoch nur kurz vorgestellt werden, sind sie doch nicht von so grosser Bedeutung für den Objektschutz wie die beiden Modelle von Harrison und Popek.

Take-Grant-Modelle

Die Take-Grant-Modelle benutzen zur Modellierung von Zugriffskontrolle Graphen, sind aber im Prinzip spezielle Zugriffsmatrix-Modelle. Der Schutzzustand in einem Take-Grant-Modell ist ein gerichteter Graph. Die Knoten des Graphen sind Subjekte oder Objekte, die gerichteten Kanten stellen die Zugriffsrechte im momentanen Schutzzustand dar, d.h. eine Kante von einem Knoten x zu einem Knoten y zeigt an, daß x gewisse Zugriffsrechte auf y hat. Die Beschriftung der gerichteten Kante ist dabei die Menge der Zugriffsrechte von x auf y, wobei als mögliche Zugriffsrechte "read", "write", "take" und "grant" in Frage kommen. Ein take-Recht von Knoten x auf Knoten y bedeutet, daß sich x

alle Rechte von y auf andere Knoten nehmen kann, ein grant-Recht von x auf y, daß x alle eigenen Rechte auf andere Knoten an y weitergeben kann.

Es gibt eine feste Menge Regeln, um Knoten oder Kanten zu einem Graph hinzuzufügen oder von einem Graph wegzunehmen: Die Take-Regel und Grant-Regel üben die entsprechenden Rechte aus, mit der Create-Regel kann ein neuer Knoten kreiert werden, und die Remove-Regel nimmt Rechte von einer Kante weg (werden sämtliche Rechte einer Kante entfernt, wird auch die Kante entfernt).

Die Sicherheitsfrage stellt sich in Take-Grant-Modellen genau wie bei Harrison: "Kann ein Subjekt x ausgehend von einem gegebenen Anfangszustand ein bestimmtes Zugriffsrecht zu einem Objekt y bekommen ?" Dies ist hier - im Gegensatz zum Modell von Harrison - entscheidbar. Der Grund dafür ist, daß es sich bei Take-Grant-Modellen um eine feste Menge von Regeln handelt, nicht um beliebige Regeln aus einer festen Menge von Grundkomponenten wie bei Harrison. Aus dieser Einschränkung gegenüber dem Modell von Harrison resultiert aber auch die beschränkte Einsatzmöglichkeömit der Take-Grant-Modelle. Ein weiterer grosser Nachteil dieser Modelle ist ihre "Symmetrie": Kann ein Subjekt x ein Zugriffsrecht auf ein Objekt y, auf das bereits ein Subjekt z ein Zugriffsrecht besitzt, bekommen, so kann z auf sämtliche Objekte, zu denen x Zugriffsrechte besitzt, ebenfalls diese Zugriffsrechte bekommen.

Aufbauend auf dem Take-Grant Grundmodell präsentiert Jones in <Jon78a> eine Erweiterung, die die Anwendbarkeit des Modells vergrössern soll: Sie führt "property sets" ein, die das beabsichtigte Verhalten der Subjekte spezifizieren. Hierdurch läßt sich eine selektive Weitergabe von Rechten modellieren. Neuere Ansätze, die auf dem Take-Grant-Modell aufbauen, stammen von Bishop und Snyder <Bis79>. Sie betrachten den möglichen Fluß von Information in Take-Grant-Modellen und unterscheiden zwischen zwei Transferarten, de-jure Transfer, d.h. das Recht, die Infor-

mation zu lesen, wird übertragen, und de-facto Transfer, d.h. die Information selbst wird übertragen, ohne daß das Recht übertragen wurde. Dieser Ansatz ist durchaus interessant, es fehlt jedoch die Möglichkeit der spezifischen Weitergabe von Rechten wie bei Jones.

Das Modell von Minsky

Minsky betrachtet in seinem Modell den Transport von Rechten zwischen Subjekten. Er setzt ebenfalls auf einem graphentheoretischen Ansatz auf, versucht jedoch die Nachteile der Take-Grant-Modelle zu vermeiden. Sein Send-Receive-Mechanismus erlaubt Selektivität beim Transport von Rechten, er bietet sogar Möglichkeiten, wertabhängige Entscheidungen bei der Weitergabe von Rechten zu treffen. Die wohl wichtigste Verbesserung gegenüber den bisherigen Modellen ist die Berücksichtigung des Lokalitätsprinzips: Der Send-Receive-Mechanismus ist so konzipiert, daß der Transport von Rechten in eine gegebene Domäne bzw. aus einer bestimmten Domäne von Entscheidungen und Rechten innerhalb dieser Domäne abhängt.

Minsky gibt aufbauend auf dem Send-Receive-Mechanismus Approximationen für "flow(x,y,right)" (d.h. das Recht "right" kann von x nach y fließen) an, und legt die Fälle dar, in denen flow(x,y,right) entscheidbar ist. Allerdings beschränkt sich Minsky wie Harrison oder die einfachen Take-Grant-Modelle auf den Fluß der Rechte in seinem System.

Bemerkungen

Die Take-Grant-Modelle sind Spezialfälle des Modells von Harrison, besitzen allerdings gegenüber diesem aufgrund ihrer Einschränkungen entscheidende Nachteile. Die Erweiterung von Jones ermöglicht zwar eine selektive Weitergabe von Rechten, jedoch räumt Jones selbst ein, daß dies nur eine von vielen notwendigen Erweiterungen der Take-Grant-Modelle ist. Interessant ist der Ansatz von Bishop und Snyder, in Take-Grant-Modellen Fragen des Informationsflusses zu betrachten; allerdings bleiben die sonstigen Mängel der Take-Grant-Modelle hier weiterhin bestehen. Das Modell von Minsky vermeidet zwar die Nachteile der Take-Grant-Modelle, beschränkt sich aber auf Betrachtungen über die Weitergabe von Rechten.

Insgesamt sind alle vorgestellten Ansätze mehr oder weniger unvollständig verglichen mit den in 3.5.3 gestellten Anforderungen an ein OV-Modell, insbesondere, was Spezifikations-, Abstraktions- und Verifikationsmöglichkeiten anbelangt.

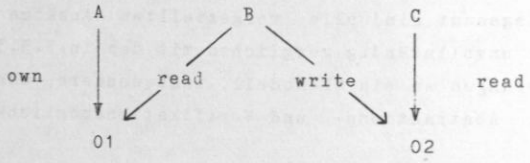
4.1.4 Zusammenfassung

In den vorhergehenden Abschnitten wurden die existierenden Objektschutzmodelle vorgestellt. Diese Modelle lassen sich in zwei Klassen einteilen: Modelle, die die Weitergabe von Rechten betrachten wie das Zugriffsmatrix-Modell, die Take-Grant-Modelle und der Send-Receive-Mechanismus, und solche, bei denen der eigentliche Zugriff auf Objekte im Vordergrund steht, wie das UCLA-Modell. Die unterschiedliche Betrachtungsweise von Zugriffsschutz führt in beiden Klassen zu verschiedenen Sicherheitsbegriffen.

Bisher gibt es noch kein Objektschutzmodell, das beide Aspekte - den Transport von Rechten und die eigentlichen Zugriffe auf Objekte - behandelt. Die Modelle für die Weiter-

gabe von Rechten erlauben es nicht, durch Angabe einer policy das beabsichtigte Verhalten beim Transport von Rechten zu spezifizieren, während Popek in seinem Modell für die Zugriffe auf Objekte - durch die Angabe der Relationen für das beabsichtigte Verhalten der Instruktionen - eine Zugriffspolicy definiert. Auch die in dieser Arbeit geforderten Abstraktions- und Verifikationsmöglichkeiten sind nur im Modell von Popek enthalten.

Das einzige Modell, das auch Informationsflußaspekte behandelt, ist das Take-Grant-Modell von Bishop und Snyder. Die Notwendigkeit einer solchen Betrachtung zeigt schon ein einfaches Beispiel: Seien A, B und C Subjekte, O1 und O2 Objekte. Die Zugriffsrechte seien folgendermassen verteilt:



Hier kann Information durch B von O1 nach O2 fliessen, d.h. C kann unter Umständen de-facto einen read-Zugriff auf O1 bekommen. Solche Informationsflüsse sind in reinen Objektschutzmodellen nicht zu entdecken.

Die bisherigen Ausführungen haben gezeigt, daß die existierenden Objektschutzmodelle - gemessen an den aufgezeigten Anforderungen - alle unvollständig sind. Darüberhinaus muß ein OV-Modell neben reinem Zugriffsschutz auch Informationsschutzaspekte berücksichtigen. Im nächsten Abschnitt sollen nun Modelle vorgestellt werden, die mit Objektschutzmethoden Informationsschutz realisieren.

4.2 Modelle für militärische Sicherheit

Alle Modelle, die in diese Klasse von Schutzsystemen fallen, versuchen, Informationsschutz innerhalb eines Objektschutzmodells zu erreichen, indem der gleichzeitige Lese- und Schreibzugriff, den ein Subjekt auf Objekte ausüben kann, beschränkt wird. Hauptziel ist hier der Schutz vor unerlaubter Enthüllung von Information. Grundlage vieler Modelle ist das Zugriffsmatrix-Modell. Die Erweiterung besteht darin, daß Sicherheitsebenen eingeführt werden, die den Subjekten und Objekten zugeordnet werden. Informationsfluß ist nur erlaubt von niedrigeren zu höheren Sicherheitsebenen bzw. innerhalb einer Ebene. Der so definierte militärische Sicherheitsbegriff wird in der englischsprachigen Literatur als "multilevel-security" bezeichnet.

Die ersten formalen Modelle hierfür stammen von Walter et al. (<Wal74>, <Wal75a>, <Wal75b>) und Bell und LaPadula (<Bel73>, <Bel74>, <Bel75>). Da beide Modelle einander sehr ähnlich sind, soll hier nur das Bell-LaPadula-Modell dargestellt werden. Alle weiteren existierenden Modelle für Informationsschutz mit Objektschutzmethoden basieren auf dem Bell-LaPadula-Modell. An erster Stelle wäre hier das Modell von Feiertag zu nennen: es diente als formales Modell für die Spezifikation von PSOS (Provable Secure Operating System) und die Verifikation von KSOS (Kernelized Secure Operating System). Weitere Ansätze, wie die Modelle von Biba und Grohn sowie die hierarchischen Take-Grant-Modelle von Bishop werden im Anschluß daran kurz präsentiert.

4.2.1 Das Bell-LaPadula-Modell

Die Notwendigkeit des Schutzes von Information vor unerlaubter Enthüllung bei militärischen Anwendungen veranlaßte die amerikanische Luftwaffe, die Entwicklung von Sicherheitskernen und von formalen Modellen zur Realisierung der Anforderungen eines militärischen Systems zu unterstützen. Auf diese Weise entstanden die beiden - einander sehr ähnlichen - Modelle von Walter und Bell und LaPadula. Auf der Basis des Zugriffsmatrix-Modells werden Subjekten und Objekten sogenannte Klassifikationen und Kategorienmengen zugeordnet, und so die Sicherheitsstufe der Subjekte und Objekte definiert. Der Sicherheitsbegriff der Modelle unterscheidet sich wesentlich von dem der Objektschutzmodelle von Harrison oder Popek: Sicherheit bezieht sich hier auf den Informationsfluß im Modell, der nur von niedrigeren zu höheren Sicherheitsebenen gestattet wird.

Wegen der Ähnlichkeit der beiden Modelle soll hier nur das Bell-LaPadula-Modell vorgestellt werden. Eine formale Darstellung des Modells wird dabei nur soweit gegeben, wie es zu seinem Verständnis notwendig ist; wichtiger ist an dieser Stelle eine Interpretation und Beurteilung des Modells. In <Rüd77> findet sich eine Zusammenfassung der verschiedenen Artikel von Bell (<Bel73>, <Bel74>, <Bel75>) in formaler Darstellung.

Modellkomponenten

Die Grundkomponenten des Modells sind eine endliche Menge von Subjekten S, eine endliche Menge von Objekten O und eine Menge von Zugriffsattributen Z := {lesen, schreiben, anhängen, ausführen, besitzen}, wobei hier "lesen" als read-only, "anhängen" als write-only und "schreiben" als read/write-Zugriff zu verstehen sind. Die Zugriffsmatrixmenge ZM wird definiert als Menge aller m x n Matrizen mit Elementen aus $\mathcal{P}(Z)$ (wobei m und n die Kardinalität von S

bzw. O sind). Für den Informationsschutz in diesem Zugriffsmatrixmodell definieren Bell und LaPadula die folgenden Komponenten:

- eine endliche, geordnete Menge von Klassifikationen C
- eine endliche Menge von Kategorien Kat
- vier Zuordnungsfunktionen F: f1: S -> C f2: O -> C
  f3: S -> $\mathcal{P}$(Kat) f4: O -> $\mathcal{P}$(Kat)

Die Klassifikationen geben an, wie stark eine Information zu schützen ist; ein Beispiel wäre {"unklassifiziert", "vertraulich", "geheim", "streng-geheim"}. Die Kategorien definieren spezielle Zugriffsprivilegien, z.B. {"Personalbüro", "Forschungslabor", "Fertigung", "Verwaltung"}. Die Zuordnungsfunktionen ordnen jedem Subjekt bzw. Objekt eine Klassifikation und eine Menge von Kategorien zu; sie definieren dadurch die Sicherheitsstufe des Subjekts bzw. Objekts.

Ein Zustand v im Modell ist charakterisiert durch die aktuell stattfindenden Zugriffe, den momentanen Stand der Zugriffsmatrix und die aktuellen Zuordnungen der Sicherheitsstufen. Die Menge aller Zustände V ist somit definiert als V:= $\mathcal{P}$(S x O x Z) x ZM x F, ein Zustand v als v:= (b,M,f). Die Zustandsübergänge werden definiert durch eine Menge von Regeln der Form ϱ : R x V -> E x V , wobei R Anforderungen von Subjekten sind und E die Entscheidungen über die gestellten Anforderungen. Anforderungen bestehen aus Anforderungselementen ("bekommen", "geben", "freigeben", "nehmen", "ändern", "erzeugen", "zerstören") und den Angaben, welches Subjekt die Anforderung stellt, an welches Subjekt/Objekt die Anforderung gestellt wird und um welche Zugriffswünsche es sich dabei handelt. Ein Beispiel hierfür wäre r = (s1,nehmen,s2,o3,anhängen), d.h. s1 möchte s2 das Zugriffsattribut "anhängen" für o3 entziehen. Als mögliche Entscheidungen kommen "ja", "nein", "Fehler" und "?" (d.h. unzuständig) in Frage.

Die so definierten Regeln entsprechen genau den Kommandos bei Harrison, beschränken sich aber nicht - wie bei Harrison - auf Veränderungen der Zugriffsmatrix. Bell und LaPadula definieren für ihr Modell 10 feste Regeln:

- Leseanforderung
- Anhängeanforderung
- Ausführungsanforderung
- Schreibanforderung
- Freigabe von Zugriffen
- Übergabe von Zugriffsattributen zwischen Subjekten
- Zurückforderung von Zugriffsattributen zwischen Subjekten
- Änderung der Klassifikations- und Kategorienfunktion eines Objekts
- Erzeugung von Objekten
- Zerstörung von Objekten

} Aktivieren bzw. Deaktivieren von Objekten, da feste Matrix

Auf eine formale Darstellung aller Regeln soll hier verzichtet werden, da dies den Rahmen dieser Arbeit erheblich sprengen würde. Stattdessen wird im folgenden am Beispiel der Regel für Leseanforderungen das Wesentliche für das Verständnis der Wirkungsweise der Regeln herausgearbeitet.

Zunächst werden hierfür einige Abkürzungen und Notationen benötigt:

$r = (\sigma 1, a, \sigma 2, o, z)$ definiere eine Anforderung, $v = (b, M, f)$ einen Zustand.
plus $b(r,v) := (b \cup \{(\sigma 2, o, z)\}, M, f)$

Seien A,B Subjekte oder Objekte.
$A >=^* B := fc(B) >= fc(b) \wedge fkat(A) \supseteq fkat(B)$, d.h. die Klassifikation von A ist grösser als die von B und die Kategorienmenge von B ist in der von A enthalten. Die Regel für Leseanforderungen ist nun definiert als:

$$
\varrho 1(r,v) := \begin{cases} (?,v) & \text{falls } \neg\, B1(r),\ \text{d.h. } \neg\,(\sigma 1 = \lambda \wedge a = \text{"bekommen"} \wedge \sigma 2 \neq \lambda \wedge z = \text{"lesen"}) \\ (\text{ja}, \text{plus } b(r,v)) & \text{falls } B1(r) \wedge \text{"lesen"} \in M(\sigma 2, o) \wedge \sigma 2 >= {}^{*} o \wedge \forall o^{\frown}[(s,o^{\frown},\text{schreiben}) \in b \vee (s,o^{\frown},\text{anhängen}) \in b]\ (o^{\frown} >= {}^{*} s) \\ (\text{nein}, v) & \text{sonst} \end{cases}
$$

B1(r) ist hier die sogenannte Bedingung der Regel, die die notwendige Besetzung der Anforderungsparameter definiert, und das Prädikat "lesen" $\in M(\sigma 2, o)$ bedeutet, daß das Leserecht für das Subjekt $\sigma 2$ auf das Objekt o in der aktuellen Zugriffsmatrix vorhanden sein muß. Die beiden zusätzlichen Voraussetzungen für die Ausführung der Regel beziehen sich auf die von Bell und LaPadula geforderten Sicherheitseigenschaften; im Anschluß an das Beispiel wird darauf noch detailliert eingegangen.

Sind alle Voraussetzungen erfüllt, so besteht die Wirkung der Regel darin, der Menge der aktuellen Zugriffe b im momentanen Zustand v den Zugriff ($\sigma 2$,o,lesen) hinzuzufügen, ansonsten bleibt der Zustand v unverändert.

Zusammenspiel der Grundkomponenten

Die Wirkung der Regeln ergibt sich aus ihrem Effekt auf die Zustände im Modell. Dies soll an einem konkreten Beispiel anhand der Regel für Leseanforderungen noch erläutert werden. Eine wichtige Rolle in der Arbeit von Bell und LaPadula spielt der Sicherheitsbegriff. Wie erwähnt unterscheidet er sich wesentlich von dem der Modelle von Harrison und Popek. Bell und LaPadula unterscheiden zwei Bedingungen für Sicherheit:

- Die normale Sicherheitsbedingung (SB)
  Die normale Sicherheitsbedingung verhindert lesenden Zugriff eines Subjekts auf ein Objekt mit höherer Sicherheitsstufe: Ein Subjekt hat nur dann Lesezugriff zu einem Objekt, wenn seine Klassifikation größer oder gleich der des Objekts ist, und seine Kategorienmenge eine Obermenge der des Objekts ist, oder formaler:
  Ein Tripel (s,o,z) erfüllt die SB relativ zu f, gdw.
  $z \in \{\text{ausführen, anhängen, besitzen}\} \vee (z \in \{\text{lesen, schreiben}\} \wedge s >=^* 0)$
  Ein Zustand v = (b,M,f) heißt sicher, wenn jedes $(s,o,z) \in b$ die SB relativ zu f erfüllt.

- Die *-Eigenschaft
  Die *-Eigenschaft verhindert das Kopieren von Information in Objekte mit niedrigerer Sicherheitsstufe: Hat ein Subjekt gleichzeitig Lese- und Schreibzugriff zu Objekten, so müssen die Klassifikationen der Objekte mit Schreibzugriffen grösser oder gleich denen der Objekte mit Lesezugriffen sein. Analoges gilt für die Kategorienmengen. Formal ausgedrückt heißt dies:
  Ein Zustand v = (b,M,f) erfüllt die *-Eigenschaft, wenn für jedes $s \in S$ gilt
  $\forall o1\ ((s,o1,\text{schreiben}) \in b \vee (s,o1,\text{anhängen}) \in b)$
  $\forall o2\ ((s,o2,\text{lesen}) \in b \vee (s,o2,\text{schreiben}) \in b) \quad (o1 >=^* o2)$

Durch diese beiden Bedingungen ist im Modell von Bell und LaPadula gewährleistet, daß Information immer nur von niedrigeren zu höheren Sicherheitsebenen fliessen kann. Bell und LaPadula beweisen, daß die angegebenen Regeln die SB und *-Eigenschaft erhalten, d.h. daß, ausgehend von einem Zustand, der die SB und *-Eigenschaft erfüllt, durch Anwendung der Regeln nur Zustände erreicht werden können, die die beiden Bedingungen ebenfalls erfüllen.

Bevor nun das Bell-LaPadula-Modell zusammemfassend bewertet werden soll, wird im folgenden noch an einem konkreten Beispiel die Wirkungsweise der Regeln auf Zustände

und die Berücksichtigung der Sicherheitsbedingungen in den Regeln dargestellt:

Der momentane Zustand des Systems sei vt = (b,M,f) mit
b = {(s1,o3,anhängen),(s2,o2,lesen)}

M:

	o1	o2	o3
s1	{besitzen,lesen, schreiben,an- hängen}	0	{lesen,anhän- gen}
s2	{anhängen}	{besitzen,lesen, schreiben,an- häng.,ausführ.}	0
s3	0	0	{besitzen,lesen, schreiben,an- hängen}

f:

	fc	fkat
s1	3	{a,b}
s2	2	{a,b}
s3	3	{b}
o1	3	{a,b}
o2	1	{a,b}
o3	1	{b}

s1 will nun aus o1 lesen und stellt die Leseanforderung
r1 = (λ ,bekommen,s1,o1,lesen)

Die Antwort auf diese Anforderung ist
$\varrho 1(r1,vt) = (nein,vt)$, weil s1 noch einen Anhänge-Zugriff auf o3 hat und o3 <* o1, d.h. die *-Eigenschaft würde verletzt werden (würde die Anforderung in der Regel nicht zurückgewiesen, könnte s1 Information von o1 lesen und in o3 anhängen). s1 muß also zuerst seinen Anhänge-Zugriff auf o3 freigeben, bevor r1 erlaubt wird, d.h.

$r1` = (\lambda, freigeben, s1, o3, anhängen)$
$r2` = (\lambda, bekommen, s1, o1, lesen)$

Der Zustand v ändert sich dabei in zwei Stufen:

$\varrho 5(r1`,vt) = (ja, vt+1=(b`,M,f))$ mit $b`=\{(s2,o2,lesen)\}$
$\varrho 1(r2`,vt+1) = (ja, vt+2=(b``,M,f))$ mit
$b``=\{(s1,o1,lesen),(s2,o2,lesen)\}$

Die Zugriffsmatrix und die Zuordnungsfunktion f ändern sich bei den angegebenen Regeln nicht.

Bemerkungen zum Modell

Ziel des Modells von Bell und LaPadula ist das Erreichen von Informationsschutz im militärischen Sinn durch Objektschutzmethoden: Information darf nur von niedrigeren zu höheren Sicherheitsebenen fliessen. Dies wird durch zwei Zusicherungen, die einfache Sicherheitsbedingung (SB) und die *-Eigenschaft, erreicht.

Der so definierte Sicherheitsbegriff von Bell und LaPadula ist völlig anders als der bei Harrison oder Popek, er bezieht sich nicht auf Veränderungen der Zugriffsmatrix oder die Einhaltung von Zugriffsspezifikationen, sondern betrachtet Zugriffe ausschließlich in Bezug auf Informationsfluß zwischen Sicherheitsebenen. Die von Bell definierte policy (Information nur von unten nach oben) ist aus verschiedenen Gründen sehr eingeschränkt und unrealistisch:

- Information kann nur in einer Richtung fließen. Es kann aber durchaus sein, daß die Verknüpfung von schutzwürdigen Informationen bzw. eine geeignete Extraktion aus einer schutzwürdigen Information nicht mehr eines Schutzes bedarf. Darüberhinaus gibt es Beispiele für Systemfunktionen, die unter dieser policy nicht lösbar sind. Das hierfür eingeführte Konzept der "trusted processes" brachte allerdings statt eines Lösungsansatzes nur eine Flut von weiteren ungelösten Problemen (vgl.4.2.4).

- Ein Informationsfluß von unten nach oben kann immer stattfinden, d.h. die Integrität von Objekten einer höheren Sicherheitsebene kann nicht gewährleistet werden.

- Jeder potentielle Informationsfluß wird betrachtet, nicht nur tatsächlich beabsichtigte Informationsflüsse (vgl. 4.3)

- Es ist keine Selektivität für Informationsfluß möglich, d.h. hat ein Subjekt Lesezugriff auf ein bestimmtes Objekt, so kann es die darin enthaltene Information in jedes einem anderen Subjekt gehörende Objekt übertragen, sofern dies eine höhere Sicherheitsstufe besitzt.

- Es sind keine wertabhängigen Entscheidungen möglich

- Die von Bell und LaPadula definierte policy ist fest, es sind keine Änderungen möglich

- Die definierte policy bezieht sich nur auf potentiellen Informationsfluß. Der Fluß von Rechten im System und die eigentlichen Zugriffe der Subjekte auf die Objekte werden nicht betrachtet.

- Das Modell kennt nur die festen Zugriffstypen lesen, schreiben, anhängen, ausführen und Besitzer

- Der Sicherheitsbegriff des Modells basiert darauf, daß sich die Subjekte - außerhalb von Objekten - nichts "merken" können, da immer nur gleichzeitige Zugriffe betrachtet werden.

Im Modell von Bell und LaPadula wird zwar ein Schutzmechanismus zur Durchsetzung der definierten policy nicht explizit angegeben, er ist aber aufbauend auf den Operationsregeln leicht definierbar. Deshalb besteht auch die Möglichkeit, Sicherheit im Sinn des Modells zu beweisen, d.h. zu beweisen, daß der definierte Schutzmechanismus die angegebene policy durchsetzt. Allerdings ist dies wegen der eingeschränkten policy und dem daraus resultierenden Sicherheitsbegriff von geringer Bedeutung für die Praxis. Weitere Nachteile des Modells sind wie bei Harrison die fehlenden Spezifikations- und Abstraktionsmöglichkeiten und der Widerspruch zum Lokalitätsprinzip und zur Modifizierbarkeit.

Das Bell-LaPadula-Modell war das erste formale Modell, das auch Informationsschutz betrachtete, ist aber heute nur mehr von historischem Interesse. In der Zwischenzeit entstanden einige Neuformulierungen des Modells, die teilweise die oben aufgeführten Nachteile zu vermeiden versuchen. Eines dieser Modelle - das Modell von Feiertag - soll nun im nächsten Abschnitt genauer betrachtet werden.

4.2.2 Das Modell von Feiertag

Seit Mitte der siebziger Jahre arbeiteten am Stanford Research Institute (SRI International) Feiertag, Neumann und andere an Methoden zur Spezifikation, Implementierung und Verifikation sicherer Betriebssysteme. Ergebnis dieser Arbeiten war der sogenannte HDM-Ansatz. HDM bedeutet "Hierarchical Development Methodology", d.h. es handelt sich hierbei um eine hierarchische Entwurfsmethode für sichere Betriebssysteme. Die grundsätzliche Vorgehensweise ähnelt sehr stark der Methode, die von Popek bei der Spezi-

fikation, Implementierung und Verifikation des UCLA-Sicherheitskerns angewandt wurde (vgl. 4.1.2), d.h. ausgehend von einem formalen Sicherheitsmodell wird das Betriebssystem in mehreren Abstraktionsebenen (Top-Level, Intermediate-Level, Bottom-Level) spezifiziert. Zwischen den verschiedenen Abstraktionsebenen werden Abbildungen angegeben, die als Basis für den Beweis der korrekten Implementierung einer Ebene durch die nächstniedrigere Ebene dienen. Zusammen mit dem Beweis, daß die Top-Level-Spezifikation die im formalen Modell geforderte policy erfüllt, kann so die Erfüllung der policy durch die Implementierung gezeigt werden.

Eine praktische Anwendung fand HDM bei der Spezifikation von PSOS (Provably Secure Operating System), einem Betriebssystem mit einer Hierarchie von abstrakten Maschinen, und bei der Spezifikation und Verifikation von KSOS (Kernelized Secure Operating System), einem Betriebssystem mit nur einer Spezifikationsebene. In beiden Fällen war die eingesetzte Spezifikationssprache SPECIAL (<Rob77>), eine nicht prozedurale Sprache, die eine Spezifikation in der von Parnas vorgeschlagenen Art und Weise (vgl. Kapitel 3) ermöglicht. Im Rahmen von HDM wurden auch zahlreiche Verifikationstools - wie z.B. der "Boyer-Moore theorem prover" - geschaffen. Eine Zusammenfassung des HDM-Ansatzes findet sich in <Che81>.

Das der Hierarchical Development Methodology zugrundeliegende Sicherheitsmodell wurde von Feiertag, Levitt und Robinson in <Fei77> beschrieben. In dieser Arbeit präsentiert Feiertag zwei Modelle für militärische Sicherheit. Das erste Modell ist eine verallgemeinerte Version der Bell-LaPadula- und Walter-Modelle. Das zweite ist eine Präzisierung des ersten Modells und hat in einigen Punkten Ähnlichkeit mit dem Bell-LaPadula-Modell. Wichtigstes Ziel bei dieser Neuformulierung war, das Modell leicht anwendbar für die Sicherheitsbeweise von Systemspezifikationen zu machen: Mit diesem Modell wurde z.B. die militärische Sicherheit der Top-Level-Spezifikationen von KSOS und PSOS bewiesen.

Der Sicherheitsbegriff von Feiertag unterscheidet sich wegen dieser Anforderungen an das Modell von dem des Bell-LaPadula-Modells. Militärische Sicherheit wird hier nicht durch Lese- oder Schreibzugriffe von Subjekten auf Objekte definiert, sondern über die funktionale Abhängigkeit von Operationen verschiedener Sicherheitsebenen. Aus diesem Grund muß sich Feiertag auch nicht auf die Zugriffsarten "lesen", "schreiben", "anhängen" und "ausführen" auf Objekte beschränken, sondern kann beliebige Operationen betrachten.

Das Modell befaßt sich ausschließlich mit der policy für militärische Sicherheit, d.h. die Zugriffsschutzanforderungen der "discretionary models" werden nicht behandelt; deshalb basiert es auch nicht auf einer Zugriffsmatrix, sondern ist in Form einer abstraktan Maschine gegeben. Natürlich können in eine Spezifikation, deren militärische Sicherheit bewiesen werden soll, zusätzlich Zugriffsschutzmechanismen, wie z.B. capabilities, eingebaut werden. Die militärische Sicherheit ist jedoch unabhängig davon und auch ohne die Semantik der Zugriffsschutzmechanismen beweisbar. Obwohl das Modell von Feiertag nur als Neuformulierung des Bell-LaPadula - Modells präsentiert wird, beinhaltet es wegen des veränderten Sicherheitsbegriffs schon Konzepte der Informationsflußmodelle für militärische Sicherheit (vgl. 4.3).

Modellkomponenten

Feiertag definiert sein Modell mit Hilfe der abstrakten Maschine <S,so,L,"<",I,K,R,Nr,Ns>. Hierbei ist S die Menge aller möglichen Zustände der Maschine und so der Anfangszustand. L ist die Menge der Sicherheitsebenen, wobei nicht mehr - wie bei Bell-LaPadula - explizit zwischen Klassifikationen und Kategorienmengen unterschieden wird. Die Relation ">" bzw. ">=" definiert eine Halbordnung auf L, entspricht also der Relation ">=*" bei Bell und LaPadula. I ist die Menge aller möglichen Operationsaufrufe im System; benötigt eine Operation Parameter, dann ist die Operation

mit jeder unterschiedlichen Parameterbelegung der Operation ein eigenes Element von I. (Im dieser Arbeit zugrundeliegenden Modell wird das Analogon zu I als Aktivitätenmenge, ein Element daraus als Aktivität bezeichnet (vgl. auch <Ker82>)).

K ist eine Funktion, die jedem Operationsaufruf eine Sicherheitsstufe zuordnet, d.h. K : I -> L, und R ist die Menge der Resultate der Operationsaufrufe. Die Funktionen Nr und Ns definieren die Zustandsübergänge der Maschine und das entsprechende Resultat bei Anwendung eines Operationsaufrufs in einem Zustand, d.h. Nr : I x S -> R und Ns : I x S -> S; sie bestimmen also die Dynamik der Maschine. Aufbauend auf der so definierten abstrakten Maschine soll nun zuerst das Grundmodell und anschließend das präzisierte Modell mit den dazugehörigen Sicherheitsbegriffen definiert werden.

Zusammenspiel der Grundkomponenten

a) allgemeines Modell

Im allgemeinen Modell bleiben die Komponenten der abstrakten Maschine noch uninterpretiert. Für die Definition von Sicherheit werden noch einige nützliche Abkürzungen eingeführt:
T bezeichnet die Menge aller möglichen Folgen von Operationsaufrufen, d.h. $T = I^*$.
M : S x T -> S ist als Erweiterung von Ns auf T definiert, d.h. M ist die reflexive, transitive Erweiterung von Ns.
E : T x L -> T ist eine Funktion, die aus einer gegebenen Folge von Operationsaufrufen diejenigen "ausblendet", die eine höhere Sicherheitsstufe als die angegebene besitzen.
Mit diesen Definitionen wird nun Sicherheit wie folgt definiert:

(P1) $(\forall i \in I, s \in S, t1, t2 \in T)\ E(t1,K(i)) = E(t2,K(i))$
$\Longrightarrow Nr(i,M(s,t1)) = Nr(i,M(s,t2))$

Informal heißt dies: werden zwei Operationsaufrufsequenzen, die sich nur durch Operationsaufrufe einer höheren Sicherheitsstufe als einer gegebenen Stufe l unterscheiden, auf einen Zustand der Maschine angewandt, so ergibt ein anschließender Aufruf einer Operation mit dieser Sicherheitsstufe l in beiden Fällen das gleiche Resultat. Anders ausgedrückt, Operationen können nur Resultate einer Operation der gleichen oder höheren, nie einer niedrigeren Sicherheitsebene beeinflussen.

Diese Definition drückt auf einfache Art und Weise die Erfordernisse der militärischen Sicherheit, daß Information nur von unten nach oben fließen darf, aus. Der Vorteil gegenüber dem Bell-LaPadula-Modell ist, daß beliebige Zugriffsarten betrachtet werden können. Leider ist durch die abstrakte Natur dieser Definition ein Beweis der Sicherheit einer Spezifikation im Sinne der policy des Modells sehr erschwert. Dies liegt daran, daß im Beweis eine Induktion über alle möglichen Operationsaufruffolgen durchgeführt werden muß. Aus diesem Grund wird von Feiertag das Modell weiter interpretiert und präzisiert.

b) präzisiertes Modell

Hier werden gegenüber dem Grundmodell die Zustände der abstrakten Maschine als Kreuzprodukt aller Zustandsvariablen interpretiert. Jeder Variablen v wird durch die Funktion H eine Sicherheitsstufe H(v) zugeordnet. Feiertag führt drei Funktionen als Abkürzungen ein, mit deren Hilfe dann Sicherheit definiert werden kann:

$Pl : S \to \prod_{\forall v(H(v)=l)} v$ bildet einen Zustand in das geordnete Tupel von Werten von Variablen der Sicherheitsstufe l ab.

$Ql : S \to \prod_{\forall v(H(v)<=l)} v$ Analog zu Pl definieren Ql und Dl die Abbildung in Tupel von Werten von Variablen der Sicherheitsstufe <= l und $\neg(>=l)$

$Dl : S \to \prod_{\forall v[\neg(H(v)>=l)]} v$

Feiertag definiert nun für sein präzisiertes Modell drei Sicherheitseigenschaften, deren Konjunktion etwas stärker ist, als (P1) des allgemeinen Modells. Man beachte, daß bei der Formulierung "j" irgendeine Funktion des entsprechenden Definitions- und Wertebereichs meint.

(P2a) $(\forall i \in I)(\exists j)(\forall s \in S) \quad Nr(i,s) = j(QK(i)(s))$

Dies heißt, daß das Resultat eines Operationsaufrufs [Nr(i,s)] nur abhängig ist von Variablen mit einer niedrigeren oder gleichen Sicherheitsstufe wie der Operationsaufruf [QK(i) blendet die Variablen mit höherer Sicherheitsstufe als der Operationsaufruf i aus].

(P2b) $(\forall i \in I, l \in L)(\exists j)(\forall s \in S)\ Pl(Ns(i,s)) = j(Ql(s))$

Dies bedeutet, daß die Werte aller Variablen einer bestimmten Sicherheitsstufe nach irgendeinem Operationsaufruf [Ns(i,s) ist das Tupel der Werte aller Variablen, Pl blendet alle mit Sicherheitsstufe $\neq$ l aus] nur von Variablen mit einer niedrigeren oder gleichen Sicherheitsstufe abhängig sind [j(Ql(s))].

(P2c) $(\forall i \in I, s \in S) \quad DK(i)(s) = DK(i)(Ns(i,s))$

Diese Bedingung sagt aus, daß durch die Ausführung einer Operation einer bestimmten Sicherheitsstufe Variablen einer niedrigeren Sicherheitsstufe nicht geändert werden [Dk(i)(s) , d.h. die Werte der Variablen mit Stufe < K(i) werden durch die Ausführung von i, d.h. Ns(i,s) nicht beeinflußt] .

Der Beweis, daß (P2a), (P2b) und (P2c) (P1) implizieren, findet sich in <Neu77>. Setzt man Variable mit den Objekten bzw. Operationen mit den Subjekten des Bell-LaPadula-Modells gleich, so entspricht die Bedingung (P2a) der einfachen SB und (P2b) der *-Eigenschaft. (P2c) ist eine Eigenschaft, die im Bell-LaPadula-Modell nicht direkt behandelt wird: Ein Beispiel hierfür wäre, daß ein Operationsaufruf der Stufe TOP SECRET einer Variablen der Stufe SECRET den Wert einer Variablen der Stufe CONFIDENTIAL zuordnet. Diese Operation ist im militärischen Sinn nicht sicher, da eine Operation der TOP SECRET-Ebene Daten der SECRET-Ebene modifiziert (auch wenn der Wert aus der CONFIDENTIAL-Ebene stammt).

Bemerkungen zum Modell

Das Hauptziel des Modells von Feiertag war seine einfache und problemlose Anwendbarkeit beim Beweis der militärischen Sicherheit von Systemspezifikationen. Der Sicherheitsbegriff von Feiertag basiert wegen dieser Anforderungen auf der funktionalen Abhängigkeit von Operationsaufrufen, nicht auf den potentiellen Zugriffen von Subjekten auf Objekte. Beim dabei betrachteten Informationsfluß handelt es sich nicht um potentiellen Fluß wie im Bell-LaPadula-Modell, sondern nur um tatsächlich beabsichtigte Flüsse zwischen verschiedenen Sicherheitsebenen.

Durch seine Spezifikations-, Abstraktions- und Verifikationsmöglichkeiten innerhalb des HDM-Ansatzes gewinnt das Modell sehr stark an Bedeutung. Die Trennung zwischen Spezifikation und Implementierung und der Beweis der Sicherheit der Spezifikation zusammen mit dem Beweis der Korrektheit der Implementierung bezogen auf die Spezifikation ermöglichen einen Sicherheitsbeweis im militärischen Sinn. Aus der Möglichkeit der Verwendung von rekursiven Funktionen und Existenz- und Allquantoren bei der Spezifikation resultiert aber im allgemeinen die Unentscheidbarkeit der funktionalen Abhängigkeit und damit auch der militärischen Sicherheit der Spezifikation. Feiertag gibt jedoch entscheidbare Ableitungsregeln zum Beweis der funktionalen Abhängigkeit an und räumt ein, daß diese Regeln bei bisherigen Spezifikationen zum Beweis genügt hätten. Weitere Vorteile des Modells von Feiertag im Vergleich zum Bell-LaPadula-Modell bestehen in folgenden Punkten:

- Feiertag beschränkt sich nicht auf bestimmte Zugriffsarten, sondern kann aufgrund des anderen Sicherheitsbegriffs beliebige Operationen betrachten.

- Durch die Betrachtung aller möglichen Operationsaufrufe sind wertabhängige Entscheidungen (abhängig vom Wert der Parameter) möglich; auch Selektivität ist hierdurch in gewisser Weise formulierbar.

Der wohl gewichtigste Mangel des Modells von Feiertag besteht darin, daß es sich auf Anwendungen der militärischen Sicherheit beschränkt (vgl. 4.2.4). Die schon bei der Beschreibung des Bell-LaPadula-Modells aufgezeigten Probleme der "trusted processes", der fehlenden Integrität von Objekten und der festen, nicht änderbaren, policy bleiben also bestehen. Darüber hinaus fehlen bei der Formulierung des Modells Ansätze für die Integration von Zugriffskontrollmechanismen und Mechanismen zur Kontrolle der Weitergabe von Zugriffsrechten, auch wenn dies natürlich prinzipiell zu-

sätzlich in das Modell eingebaut werden kann. Das Modell ist von seiner Konzeption her - ähnlich wie das Modell von Popek - besonders geeignet für den Einsatz bei der Spezifikation und Verifikation von Sicherheitskernen in Betriebssystemen.

Insgesamt betrachtet liegt die Bedeutung des Modells durch seine Spezifikations-, Abstraktions- und Verifikationsmöglichkeiten im Einsatz für praktische Anwendungen, wie z.B. bei den Projekten PSOS und KSOS. Die Beschränkung auf militärische Sicherheit bedeutet jedoch eine wesentliche Einschränkung für die allgemeine Anwendbarkeit des Modells.

4.2.3 Weitere Modelle für militärische Sicherheit

Das Modell von Bell und LaPadula ist wohl das bekannteste der Modelle für militärische Sicherheit, und Grundlage für alle Modelle mit diesen Sicherheitsanforderungen. Zusammen mit dem Modell von Feiertag, das seine Bedeutung aus der Anwendbarkeit für Sicherheitsbeweise von Systemspezifikationen gewinnt, wurde es deshalb detailliert vorgestellt.

Aufbauend auf dem Bell-LaPadula-Modell entwickelte Biba in <Bib77> ein Modell, das die Integrität von Objekten berücksichtigt. Die Arbeit von Grohn <Gro76> behandelt ebenfalls die Integrität von Objekten, untersucht aber zusätzlich die Anwendbarkeit des Bell-LaPadula-Modells auf Datenbanken, worauf hier jedoch nicht eingegangen wird.

Auch in neuester Zeit entstanden noch einige Artikel, in denen Modelle für militärische Sicherheit präsentiert werden. An erster Stelle wären hier die Arbeiten von Bishop <Bis81> und Wu <Wu 81> zu nennen; sie befassen sich mit der Integration von militärischen Sicherheitspolicies in die bekannten Take-Grant-Modelle (vgl. 4.1.3). Wegen der Ähnlichkeit der beiden Ansätze soll hier nur das Modell von

Bishop vorgestellt werden. Eine weitere Veröffentlichung von Dion <Dio81> hat wie die Arbeiten von Biba und Grohn eine Erweiterung des ursprünglichen Bell-LaPadula-Modells in Richtung Integrität von Objekten zum Thema; da hier jedoch nichts grundlegend neues gebracht wird, wird dieser Artikel hier nicht näher betrachtet.

Insgesamt können alle erwähnten Ansätze - um den Rahmen nicht zu sprengen - nur kurz diskutiert werden.

Datenintegritätsmodelle

Als eine der grundlegenden Aufgaben von Schutzmechanismen wurden in Kapitel 3 dieser Arbeit der Schutz vor unerlaubter Modifikation von Daten, d.h. die Forderung nach Datenintegrität, präsentiert. Im Modell von Bell und LaPadula wird dieser Gesichtspunkt leider überhaupt nicht berücksichtigt, es wird ausschließlich die Datengeheimhaltung, d.h. der Schutz vor unerlaubter Enthüllung von Information, betrachtet. Diesen Mangel versucht Biba in <Bib77> durch die Einführung von Integritätsebenen und einer Integritätspolicy aufzuheben. Integritätsebenen werden dabei analog zu Sicherheitsebenen durch Klassifikationen und Kategorienmengen definiert, und eine Relation >=* wird auf den Integritätsebenen eingeführt. Biba definiert nun verschiedene Möglichkeiten für Integritätspolicies, wobei die sogenannte "policy of strict integrity" das duale Gegenstück zur security-policy von Bell und LaPadula ist, d.h. ein Subjekt kann nur Objekte lesen, die eine höhere oder gleiche Integritätsstufe besitzen, und nur solche Objekte mit niedrigerer oder gleicher Integritätsstufe modifizieren.

Auch in der Arbeit von Grohn werden zum Schutz vor unerlaubter Modifikation Integritätsebenen eingeführt, die durch Funktionen gs und go den Subjekten bzw. Objekten zugeordnet werden. Zusammen mit den Funktionen fs und fo, die die Sicherheitsstufen der Subjekte bzw. Objekte definieren, entstehen so die sogenannten Schutzstufen Πs und Πo, wobei

$\pi = f \times g$ ist. Eine Relation $>=\grave{}$ für die Schutzstufen wird definiert als:

$$\pi(A) >=\grave{} \pi(B) \text{ gdw. } fc(A) >= fc(B) \wedge fkat(A) \supseteq fkat(B) \wedge gc(A) <= gc(B) \wedge gkat(A) \subseteq gkat(B)$$

(fc bzw. gc sind hierbei die Klassifikations-, fkat und gkat die Kategorienfunktionen).
Analog zu Bell und LaPadula wird nun die einfache Schutzbedingung und die *-Eigenschaft für Schutz definiert, d.h. Information fließt nur von niedrigeren zu höheren Schutzebenen, bzw. von niedrigeren zu höheren Sicherheitsebenen, aber nur dann, wenn gleichzeitig von höheren zu niedrigeren Integritätsebenen.

Die Erweiterung gegenüber dem ursprünglichen Bell-LaPadula-Modell besteht bei beiden Ansätzen nur in der Einführung der Integritätsebenen. Anders als beim Modell von Feiertag werden hier Spezifikations- und Verifikationsmöglichkeiten nicht behandelt. An dieser Stelle wäre noch zu erwähnen, daß die neueste Version des Modells von Feiertag - die bei KSOS zum Einsatz kam (vgl. <Cau79>) - ebenfalls Integritätsebenen berücksichtigt; hierbei wird allerdings nur zwischen "system administrator" (höchste), "operator" und "user" (niedrigste) unterschieden. Allgemein muß zu dem Integritätsebenen-Ansatz gesagt werden, daß zur Zeit noch wenig praktische Erfahrung damit gewonnen werden konnte. Wie Landwehr bemerkt <Lan81>, ist es noch unklar, wie Integritätsebenen den einzelnen Systemobjekten in der Praxis zugeordnet werden müssen (bzw. sollten).

Hierarchische Take-Grant-Modelle

In 4.1.3 wurden die Take-Grant-Modelle als Objektschutzmodelle präsentiert, die den Transport von Rechten in einem System betrachten. In der Arbeit <Bis79> erweiterten Bishop und Snyder diese Modelle durch die Einführung der de-facto Regeln auf Betrachtungen des Flusses von Informa-

tion. Die neueste Arbeit von Bishop <Bis81> befaßt sich nun mit den sogenannten hierarchischen Take-Grant-Modellen. Es handelt sich hierbei um Untersuchungen über den Informationsfluß in Take-Grant-Modellen im Zusammenhang mit den Anforderungen der militärischen Sicherheit. Bishop bildet die Sicherheitsebenen (definiert durch Klassifikationen und Kategorienmengen) der Modelle für militärische Sicherheit in sogenannte "rwtg-Ebenen" der Take-Grant-Modelle ab. In einer rwtg-Ebene (read, write, take, grant-Ebene) besteht dabei für jedes darin enthaltene Subjekt die Möglichkeit, sämtliche Rechte aller anderen Subjekte dieser Ebene und auch sämtliche - allen anderen Subjekten dieser Ebene de-facto zugänglichen - Informationen ebenfalls zu bekommen. Zwischen den verschiedenen rwtg-Ebenen wird nun durch eine Relation "higher" eine Halbordnung definiert. Bishop untersucht, welche Einschränkungen den de-jure Regeln zur Übertragung von Rechten aufzuerlegen sind, um der Forderung der militärischen Sicherheit - Information darf nur von niedrigeren zu höheren Ebenen fließen - gerecht zu werden. Diese Einschränkungen beziehen sich auf den Transport von "read" und "write"-Rechten und entsprechen genau der einfachen Sicherheitsbedingung ("no read up") und der *-Eigenschaft ("no write down") von Bell und LaPadula.

Bemerkungen

Die hier vorgestellten Ansätze von Biba und Grohn erweitern das ursprüngliche Bell-LaPadula-Modell zwar um Integritätsbetrachtungen, die sonstigen Nachteile des Modells (vgl. 4.2.1) bleiben jedoch bestehen. Der Ansatz von Bishop ist zwar rein formal recht interessant, bringt jedoch gegenüber dem Bell-LaPadula-Modell überhaupt keine Erweiterungen; es wird lediglich eine andere Notation - die der Take-Grant-Modelle - benutzt.

Insgesamt fehlen allen diesen Modellen Spezifikations- und Verifikationsmöglichkeiten, wie z.B. beim Modell von Feiertag. Den wohl wichtigsten Nachteil aller Modelle für

militärische Sicherheit, nämlich die militärische Sicherheitspolicy selbst, soll im nächsten Abschnitt noch genauer diskutiert werden.

4.2.4 Zusammenfassung

Alle hier beschriebenen Modelle zur militärischen Sicherheit basieren direkt oder indirekt auf dem von Bell und LaPadula angegebenen militärischen Sicherheitsbegriff, der durch die einfache Sicherheitsbedingung und die *-Eigenschaft repräsentiert wird. Bereits bei den Bemerkungen zum Bell-LaPadula-Modell wurden einige Mängel aufgezeigt, die sich nicht speziell auf dieses Modell, sondern allgemein auf das Konzept der militärischen Sicherheit beziehen.

An dieser Stelle soll noch einmal auf das Hauptproblem dieses Sicherheitsbegriffs, nämlich die strikte Beschränkung des Informationsflusses auf eine Richtung, und die daraus resultierenden Konsequenzen, eingegangen werden, um seine Nicht-Adäquatheit für Anwendungen bei realen Systemen aufzuzeigen. Dies ist bereits an einem einfachen Beispiel (vgl. <Rus81a>) zu sehen:

Man betrachte ein Programm zum Ausdruck von Spooldateien. Ordnet man dem Spooler und allen Spooldateien die höchste Sicherheitsstufe zu, so können Anwender mit einer niedrigeren Sicherheitsstufe ihre eigenen Spooldateien nicht mehr lesen, nicht einmal, um z.B. den Fortschritt ihrer Jobs zu kontrollieren. Aus diesem Grund ist es üblich, Spooldateien die Sicherheitsstufe ihrer Besitzer zuzuordnen. Der Spooler selbst muß dann die höchste Sicherheitsstufe bekommen, damit er alle Spooldateien lesen kann. Dann kann aber der Spooler nach dem Ausdruck des Inhalts der Spooldateien diese nicht löschen, ja nicht einmal eine Nachricht für den Besitzer der Spooldateien über die Beendigung seines Jobs geben, da dies der *-Eigenschaft widersprechen würde. Um eine akzeptable Benutzeroberfläche zur Verfügung zu

stellen, muß der Spooler die Erlaubnis bekommen, die *-Eigenschaft zu verletzen, er wird zum "trusted process".

In realen Systemen existieren sehr viele Funktionen, die sich nicht in das starre Schema der militärischen Sicherheitsanforderungen pressen lassen, und die deshalb mit den Privilegien der trusted processes ausgestattet werden müssen (vgl. <Ber79> p.365). Probleme entstehen nun natürlich bei der Verifikation der militärischen Sicherheit eines Systems; um militärische Sicherheit garantieren zu können, genügt es dann nämlich nicht mehr, den Sicherheitskern zu verifizieren. Durch die Verlagerung von Sicherheitsfragen aus dem Kern in die trusted processes muß die Kombination aus Kern und trusted process verifiziert werden. Hierfür bieten jedoch die existierenden Modelle keinerlei Unterstützung an. Durch das Fehlen jeglicher präziser Formulierung der Rolle der trusted processes in den militärischen Sicherheitsmodellen, und das Fehlen jedes formalen Verständnisses, wie Eigenschaften von trusted processes mit Eigenschaften des Kerns zum Beweis der Sicherheit des Gesamtsystems kombiniert werden können, fehlt eigentlich auch die Berechtigung, von der "Verifikation" der Sicherheit solcher Systeme zu sprechen. Selbst Berson und Barksdale räumen in ihrer Beschreibung von KSOS zum Problem der trusted processes ein:

"To a large extent, they (Anm.: trusted processes) represent a mismatch between the idealisations of the multilevel security model and the practical needs of a real user environment" (vgl. <Ber79> p.365).

Aus den bisherigen Ausführungen wurde deutlich, daß das Konzept der militärischen Sicherheit bei der Anwendung in der Praxis so schwerwiegende Probleme aufwirft, daß es für den Einsatz bei realen Systemen - insbesondere für die Verifikation von Sicherheit - nicht geeignet ist. Außerdem werden - abgesehen vom Modell von Feiertag - in den existierenden Modellen für militärische Sicherheit Aspekte der Spezifikation und Verifikation von Sicherheit völlig außer acht gelassen. Im folgenden Abschnitt sollen nun - um den

Überblick über formale Sicherheitsmodelle zu vervollständigen - Informationsflußmodelle betrachtet werden.

4.3 Informationsflußmodelle

In Abschnitt 4.2 wurden Modelle diskutiert, die zur Durchsetzung der militärischen Sicherheitspolicy eine Kontrolle des Informationsflusses mit Objektschutzmethoden durchführten. Zwei wesentliche Punkte der Kritik an diesen Modellen waren, daß zum einen jeder potentielle Informationsfluß, nicht nur tatsächlich beabsichtigte Flüsse, und zum anderen nur feste Zugriffsarten wie z.B. "lesen" und "schreiben" betrachtet werden. Eine Ausnahme bildet hier allerdings schon das Modell von Feiertag; in ihm werden bereits die Grundprinzipien der hier zu behandelnden Informationsflußmodelle berücksichtigt.

Informationsflußmodelle untersuchen den Fluß von Information in Programmen im allgemeinen unabhängig von festen Sicherheitspolicies - im Gegensatz zu militärischen Sicherheitsmodellen. Sie betrachten Programme als Medien für Informationsübertragung, wobei nur tatsächliche, durch die Statements der Programme beabsichtigte, Informationsflüsse berücksichtigt werden. Natürlich können auf diese Art beliebige Zugriffsarten untersucht werden.

Die hierzu existierenden Modelle lassen sich in zwei Klassen einteilen, syntaktische und semantische Modelle. Syntaktische Modelle untersuchen den Informationsfluß in Statements von Programmen rein syntaktisch, d.h. ohne Berücksichtigung des momentanen Systemzustands; diese syntaktische Analyse kann z.B. von einem Compiler zusätzlich vorgenommen werden. Semantische Modelle berücksichtigen darüberhinaus noch den Systemzustand, z.B. gegeben durch die Werte aller Variablen des Systems. Aus diesem Grund sind semantische Modelle präziser als syntaktische in dem Sinn, daß bei gewissen Systemzuständen semantisch betrachtet (und auch tatsächlich) kein Informationsfluß stattfindet, obwohl eine rein syntaktische Betrachtungsweise Informationsfluß für das betreffende Programm diagnostiziert (Beispiele hierfür finden sich in den folgenden Abschnitten).

Das älteste Informationsflußmodell ist syntaktischer Natur und stammt von D.E.Denning (vgl. <Den75>, <Den76>, <Den77>). Dieses Modell ist gewissermaßen das Basismodell für Informationsflußbetrachtungen; alle anderen Informationsflußmodelle wurden von ihm stark beeinflußt. Aus diesem Grund soll hier eine genauere Darstellung des Modells von Denning präsentiert werden. Die neuesten Untersuchungen auf dem Gebiet des syntaktischen Informationsflusses stammen von Andrews und Reitman (<Rei79>, <And80>); in diesen Arbeiten werden Erweiterungen des Grundmodells von Denning angegeben, auf die kurz eingegangen werden wird. Das Modell von Jones und Lipton (vgl. <Jon75>, <Jon78b>) ist ein allgemeiner formaler Ansatz für Informationsflußmodelle und bringt einige wichtige theoretische Ergebnisse; es soll ebenfalls genauer vorgestellt werden. Ein weiterer sehr interessanter - semantischer - Ansatz, der näher präsentiert werden soll, stammt von Cohen (<Coh77>, <Coh78>); hier werden informationstheoretische Grundlagen zum Erkennen von Informationsfluß in Programmen herangezogen. Die nächste Methode zur semantischen Betrachtung von Informationsfluß ist von Furtek und Millen (vgl. <Fur78>, <Mil78>); ihre Theorie der "constraints" wird hier jedoch nur kurz betrachtet. Auch auf das Modell von Landauer und Crocker <Lan82> soll nur kurz eingegangen werden.

4.3.1 Das Verbandsmodell von Denning

Das von Dorothy Denning entwickelte syntaktische Informationsflußmodell betrachtet den Fluß von Information in Statements und Programmen einer Programmiersprache. Denning definiert eine Informationsflußpolicy durch eine Menge von Sicherheitsklassen, eine Flußrelation, die den erlaubten Fluß zwischen Sicherheitsklassen spezifiziert, und eine Zuordnungsmethode, die für jedes Objekt eine Sicherheitsklasse angibt. Die Sicherheitsklassen bilden zusammen mit der Flußrelation einen Verband. Diese Definition ähnelt in gewisser Weise der militärischen Sicherheitspolicy; allerdings sind die Methoden zur Bestimmung von Informationsfluß und auch der Sicherheitsbegriff bei Denning völlig anders als in den Modellen zur militärischen Sicherheit: Eine Operation oder eine Folge von Operationen, die den Wert eines Objekts x benutzt, um daraus den Wert eines anderen Objekts y abzuleiten, verursacht einen Fluß von x nach y. Ein solcher Informationsfluß ist nur dann zulässig, wenn die Flußrelation den Informationsfluß von der Sicherheitsklasse von x nach der Sicherheitsklasse von y erlaubt.

Durch die Betrachtung des Informationsflusses in Statements oder Folgen von Statements wird - im Gegensatz zu den militärischen Sicherheitsmodellen - nur tatsächlich beabsichtigter Informationsfluß berücksichtigt. Der Mechanismus von Denning untersucht den Informationsfluß rein syntaktisch, d.h. ohne Berücksichtigung des Systemzustands wie z.B. Variablenwerte, er kann deshalb bereits zur Kompilierungszeit auf Programme angewandt werden; dies bedingt allerdings auch eine statische Bindung von Objekten zu Sicherheitsebenen.

Modellkomponenten

Im Verbandsmodell von Denning wird eine Informationsflußpolicy definiert durch <S,"->">, wobei S eine Menge von Sicherheitsklassen ist und "->" eine Flußrelation, die erlaubte Informationsflüsse zwischen Paaren von Sicherheitsklassen spezifiziert. Jedem Speicherobjekt x (Konstante, Variable, Feld oder File) wird eine Sicherheitsklasse zugeordnet, die durch $\underline{x}$ bezeichnet wird. Die Notation $\underline{x} \rightarrow \underline{y}$ bedeutet, daß ein Fluß von Objekt x zu Objekt y gemäß Flußpolicy erlaubt ist; die Zuordnung der Sicherheitsklassen zu Objekten wird dabei als statisch angenommen. Unter den durchaus vernünftigen Annahmen, daß S eine endliche Menge ist und "->" reflexiv und transitiv ist, werden von Denning für <S,"->"> Verbandseigenschaften gefordert, d.h. für jedes Paar von Sicherheitsklassen gibt es je eine einzige Sicherheitsklasse als Supremum und Infimum. Die Symbole $\oplus$ und $\odot$ bezeichnen hierbei die Verbandsoperationen für die kleinste obere Schranke und die größte untere Schranke. Für die kleinste obere Schranke und die größte untere Schranke ergibt sich die folgende Bedeutung im Modell:

$\forall i \in \{1,\ldots,m\}$ $\underline{x_i} \rightarrow \underline{y}$ gdw. $\underline{x_1} \oplus \ldots \oplus \underline{x_m} \rightarrow \underline{y}$ und
$\forall j \in \{1,\ldots,n\}$ $\underline{x} \rightarrow \underline{y_j}$ gdw. $\underline{x} \rightarrow \underline{y_1} \odot \ldots \odot \underline{y_n}$

Es gibt eine höchste Klasse H, die kleinste obere Schranke aller Klassen und eine niedrigste Klasse L, die größte untere Schranke aller Klassen. Am Beispiel des Boole`schen Verbands für n=3 läßt sich das Modell von Denning so darstellen:

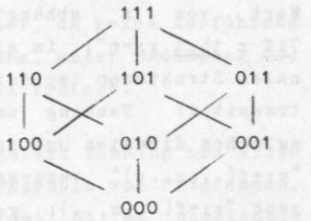
S = {000,001,...,111}

A -> B gdw. OR(A,B) = B

A ⊕ B = OR(A,B)

A ⊗ B = AND(A,B)

L = 000 H = 111



Bei der Spezifikation eines Programms wird nun im Deklarationsteil allen Grössen des Programms eine Sicherheitsklasse zugeordnet; Konstanten sind in der Klasse L enthalten, da sie ja keine Information über irgendein anderes Objekt enthalten.

Aufbauend auf dem so definierten Verbandsmodell für Sicherheitsklassen erläutert Denning nun die Begriffe Informationsfluß und Sicherheit und gibt einen Verifikationsmechanismus für die Sicherheit von Programmen an.

Zusammenspiel der Grundkomponenten

Der zentrale Punkt im Modell von Denning ist der Begriff "Informationsfluß": ein Informationsfluß von einem Objekt x zu einem Objekt y - abgekürzt als x => y - findet statt, wenn eine Information, die in x gespeichert ist, nach y übertragen wird, oder dazu benutzt wird, eine Information abzuleiten, die nach y übertragen wird. Ein Statement spezifiziert einen Fluß x => y, wenn die Ausführung des Statements in einem Fluß x => y resultieren könnte. Denning unterscheidet zwischen zwei Arten von Informationsfluß, explizitem und implizitem Fluß. Ein expliziter Fluß x => y findet statt, wenn die Operationen, die ihn verursachen, unabhängig vom Wert von x sind; Wertzuweisungen, I/O-Statements und Prozeduren mit Aufrufparametern verursachen z.B. explizite Flüsse. Ein impliziter Fluß x => y findet statt, wenn ein Statement einen Fluß von einem beliebigen z nach y

spezifiziert, die Ausführung dieses Statements jedoch vom Wert von x abhängt. Ein Beispiel hierfür wäre "if x then y:=z"; im allgemeinen verursachen alle konditionalen Strukturen implizite Flüsse. Die Relation "=>" ist transitiv; Denning unterscheidet in diesem Zusammenhang zwischen direkten und indirekten Flüssen. Eine Zuordnung "y:=f(..,x,..)" verursacht einen direkten Fluß x => y während "z:=f(..,x,..); y:=g(..,z,..)" einen indirekten Fluß x => y verursacht.

Mit dem so definierten Begriff des Informationsflusses ist Denning nun in der Lage, den Begriff "Sicherheit eines Programms" einzuführen: ein Programm P ist sicher, gdw. keine Ausführung von P einen Fluß x => y verursacht, für den x -> y nicht gilt. Eine notwendige und hinreichende Bedingung für die Sicherheit von P ist dann also

(1) x => y für eine beliebige Ausführung von P nur dann, wenn x -> y

Leider ist diese Bedingung (1) unentscheidbar, wie das einfache Statement "if f(x) hält then y:=0" zeigt; das Halteproblem für eine beliebige rekursive Funktion ist ja bekanntlich unentscheidbar, und deshalb auch der Informationsfluß x => y im angegebenen Beispielstatement.

Ersetzt man bei der Definition der Sicherheit eines Programms die unentscheidbare Bedingung (1) durch

(2) x => y ist spezifiziert durch P nur dann, wenn x -> y

so wird die Sicherheit von P entscheidbar. Allerdings ist die Bedingung (2) weniger präzise als (1), da der Fluß rein syntaktisch durch die Spezifikationen des Programms betrachtet wird, und durch die syntaktische Analyse unter Umständen Flüsse in Programmen diagnostiziert werden, die in Wirklichkeit gar nicht stattfinden können. Man betrachte z.B. das Programm "if x=0 then if x≠0 then y:=z" zusammen mit einer Flußrelation, die nur z => y verbietet. Dieses

Programm ist unter Bedingung (1) sicher, da keine Ausführung einen Fluß z => y zur Folge haben kann, unter Bedingung (2) jedoch nicht sicher, da es z => y spezifiziert.

Aufbauend auf Bedingung (2) definiert Denning nun einen Verifikationsmechanismus für die Sicherheit von Programmen, der für jedes gegebene Programm bestimmt, ob es unerlaubte Flüsse spezifiziert. Die Transitivität der Flußrelation spielt hierbei eine wichtige Rolle: da Folgen von sicheren direkten Informationsflüssen ebenfalls sicher sind, muß bei der Verifikation jeweils nur der direkte Fluß für die verschiedenen syntaktischen Typen von Statements betrachtet werden. An dieser Stelle sollen nicht die Verifikationsregeln für sämtliche Statementtypen präsentiert werden, sondern an einem Beispielprogramm die Vorgehensweise dargelegt werden (vgl. <Den77>).

Bei der Verifikation von Selektions- und Iterationsstatements ist hier natürlich der implizite Informationsfluß von den Kontrollausdrücken der Statements zu den Variablen, in die im Geltungsbereich dieser Statements Information fließt, nicht zu vernachlässigen. Das folgende Programm ist in einer PASCAL-ähnlichen Sprache formuliert, die Deklarationsliste enthält zusätzlich für jede Variable die jeweilige Sicherheitsklasse; aus Gründen der Einfachheit wird nur zwischen den Klassen H und L unterschieden. In der Spalte rechts neben den Programmstatements findet sich der jeweilige Verifikationsschritt.

```
begin
  i,n    :  integer security class L;
  flag   :  Boolean security class L;
  f1,f2  :  file security class L;
  x,sum  :  integer security class H;
  f3,f4  :  file security class H;

  begin
    i:=1;                          1 -> i (L -> L)
    n:=0;                          0 -> n (L -> L)
    sum:=0;                        0 -> sum (L -> H)
    while i<=100 do
      begin
        input flag from f1;        f1 -> flag (L -> L)
        output flag to f2;         flag -> f2 (L -> L)
        input x from f3;           f3 -> x (H -> H)
        if flag then
          begin
            n:=n+1;                n ⊕ 1 -> n (L -> L)
            sum:=sum+x;            sum ⊕ x -> sum (H -> H)
          end;                     flag -> n ⊛ sum (L -> L)
        i:=i+1;                    i ⊕ 1 -> i (L -> L)
      end;                         i ⊕ 100 -> flag ⊛ f2 ⊛ x
                                   ⊛ n ⊛ sum ⊛ i (L -> L)
    output n,sum,sum/n to f4;      n ⊕ sum ⊕ sum ⊕ n -> f4
                                   (H -> H)
  end;
end;
```

In <Den77> zeigt Dorothy Denning die Korrektheit des angegebenen Verifikationsmechanismus durch den Beweis des Satzes "Ein Programm wird durch den vorgegebenen Mechanismus nur verifiziert, wenn es sicher ist". Anschliessend wird die von Denning angegebene einfache Sprache, die sich auf Wertzuweisungen, I/O-Statements und einfache Kontrollstrukturen beschränkte, um zusätzliche Kontrollstrukturen, Datenstrukturen und Prozeduraufrufe erweitert. Der Einsatz des angegebenen Verifikationsmechanismus zur Beantwottung der Confinement-Frage eines Programms wird abschließend aufge-

zeigt.

Bemerkungen zum Modell

Ziel des Modells von Denning und aller anderen Informationsflußmodelle des Abschnitts 4.3 ist das Erkennen von Informationsfluß in Statements und Programmen, wobei im Gegensatz zu den Modellen für die militärische Sicherheit nur tatsächlich durch die Statements beabsichtigte Flüsse betrachtet werden, und sich die Flußbetrachtungen nicht auf feste Zugriffsarten oder Anwendungen der militärischen Sicherheit beschränken. Die Informationsflußmodelle des Abschnitts 4.3 sind jedoch keine Objektverwaltungsmodelle im Sinn von Abschnitt 3.5, befassen sie sich doch ausschließlich mit Fragen des Informationsflusses unter dem Aspekt der Datengeheimhaltung; es werden weder Fragen der Datenintegrität noch Probleme des Zugriffsschutzes oder der Weitergabe von Rechten betrachtet. Auch die in den Modellen vorhandenen Möglichkeiten der Spezifikation und Verifikation beschränken sich auf das Teilgebiet des Informationsflusses.

Das Modell von Denning unterscheidet sich allerdings von den anderen Modellen dieses Abschnitts in einigen wesentlichen Punkten, die hier kurz dargestellt werden sollen. Da es den Informationsfluß in Statements und Programmen rein syntaktisch betrachtet, kann der von Denning angegebene Verifikationsmechanismus bereits zur Kompilierungszeit eingesetzt werden; dies bedingt allerdings eine statische Bindung von Sicherheitsklassen an Objekte bzw. Variablen. Auch die Möglichkeiten, in das Laufzeitverhalten von Programmen einzugreifen - z.B. um wertabhängige Entscheidungen zu treffen - sind dadurch natürlich nicht gegeben. Ein weiterer Nachteil des Modells ist seine Beschränkung auf eine, zwar relativ allgemeine, aber dennoch feste policy und die Anwendbarkeit des gegebenen Verifikationsmechanismus nur für diese policy. Diese Einschränkungen führen allerdings zu einem sehr einfach anwendbaren und sehr klaren Verifikationsmechanismus und ermöglichen die Entscheidbarkeit von

Sicherheit im Modell von Denning.

Insgesamt betrachtet brachte das Modell von Denning, verglichen mit den Modellen zur militärischen Sicherheit, einen großen Fortschritt auf dem Gebiet der Problematik des Informationsflusses in Systemen. Dies zeigt auch die Tatsache, daß es heute noch als Basis für neuere Informationsflußmodelle - wie z.B. das in 4.3.4 vorzustellende Modell von Andrews und Reitman - dient.

4.3.2 Das Modell von Jones und Lipton

Jones und Lipton betrachten in ihrer Arbeit <Jon78b> den Informationsfluß von den Inputvariablen eines Programms zu dessen Outputvariablen. Sie definieren für Programme ihres Modells Sicherheitspolicies in nichtprozeduraler Form, die als eine Art von Informationsfilter für Inputwerte wirken. Ein Schutzmechanismus für ein Programm und eine gegebene Sicherheitspolicy wird dann in prozeduraler Form angegeben und als zuverlässig definiert, wenn er die, durch die policy dem Programm auferlegten Informationsflußbeschränkungen durchsetzt. Jones und Lipton führen anschließend als Vergleichskriterium für zwei Schutzmechanismen zu einem Programm und einer Sicherheitspolicy deren "Vollständigkeit" ein und untersuchen die Frage nach der effektiven Konstruierbarkeit eines maximal vollständigen, zuverlässigen Schutzmechanismus. Als Anwendungsbeispiel geben sie für Programme in einer einfachen Flußdiagramm-Sprache einen sogenannten "Überwachungs-Schutzmechanismus" an, dessen Anwendung allerdings auf einfache policies beschränkt wird; außerdem liefert dieser Schutzmechanismus nur eine syntaktische Analyse des Informationsflusses. Eine interessante Erweiterung dieses Überwachungsmechanismus auf die Betrachtung der Laufzeit eines Programms als verdeckten Informationsflußkanal wird zuletzt präsentiert.

Dieser Ansatz von Jones und Lipton ist vor allem deswegen von Bedeutung, weil das angegebene Modell als einziges eine formale Definition der Begriffe "Sicherheitspolicy" und "Schutzmechanismus" gibt, und auch die Sicherheit von Programmen präzise formuliert.

Modellkomponenten

Im Modell von Jones und Lipton wird ein Programm als totale Funktion Q: D1 x ... x Dk -> E definiert, wobei Di der Wertebereich der i-ten Inputvariablen und E der Wertebereich des Programmoutputs ist. Ein Schutzmechanismus für ein Programm Q ist eine totale Funktion
M: D1 x ... x Dk -> E$\cup$F, wobei für alle (d1,...,dk) aus D1 x ... x Dk gilt: entweder ist
M(d1,...,dk) = Q(d1,...,dk) oder M(d1,...,dk)$\in$F. Die Menge F besteht aus den "Verletzungsanzeigen" des Mechanismus M. Ein Schutzmechanismus ist also eine Art "Türwächter", der den normalen Output eines Programms in allen Fällen, in denen ein unzulässiger Informationsfluß stattfände, durch eine Verletzungsanzeige ersetzt. Eine Sicherheitspolicy für ein Programm Q: D1 x ... x Dk -> E ist definiert als eine totale Funktion I: D1 x ... x Dk -> $\mathcal{W}$, wobei $\mathcal{W}$ eine neue Menge ist; die Sicherheitspolicy arbeitet gewissermaßen als Informationsfilter, d.h. der Wert von I(d1,...,dk) enthält nur mehr einen Teil der ursprünglichen Information (d1,...,dk). Jones und Lipton definieren nun durch "allow(i1,...,im)" eine spezielle Form von Sicherheitspolicies, nämlich das "Herausfiltern" der Komponenten di1,...,dim aus (d1,...,dk). Diese policies sind dadurch charakterisiert, daß sie über jede Inputvariable entweder Information erlauben, oder nicht erlauben. Obwohl die Autoren einräumen, daß die allgemeine Form der Sicherheitspolicies auch wertabhängige policies (content-dependent policies) beinhaltet, werden im weiteren Verlauf ihrer Arbeit im wesentlichen nur mehr die wertunabhängigen policies der Form allow(...) betrachtet.

Zusammenspiel der Grundkomponenten

Der zentrale Punkt im Modell von Jones und Lipton ist die Verbindung zwischen Sicherheitspolicy und Schutzmechanismus, d.h. die Frage, ob ein Schutzmechanismus die durch eine gegebene policy definierten Informationsflußbeschränkungen durchsetzt; diese Beziehung wird "Zuverlässigkeit" des Mechanismus genannt.

Sei $I: D1 \times \ldots \times Dk \to \mathcal{W}$ eine Sicherheitspolicy und $M: D1 \times \ldots \times Dk \to E \cup F$ ein Schutzmechanismus für ein Programm $Q: D1 \times \ldots \times Dk \to E$. M heißt zuverlässig für I und Q, falls es eine Funktion $M': \mathcal{W} \to E \cup F$ gibt, so daß für alle $(d1,\ldots,dk)$ gilt $M(d1,\ldots,dk) = M'(I(d1,\ldots,dk))$.

Ein Mechanismus ist also genau dann zuverlässig, wenn er sich so verhält, als ob er statt des Inputs $(d1,\ldots,dk)$ nur $I(d1,\ldots,dk)$ bekommen hätte.

Ein kleines Beispiel soll diesen Sachverhalt näher erläutern. Für ein Programm Q gibt es zwei triviale Schutzmechanismen. Der erste ist das Programm selbst, der zweite ist gegeben durch $M: D1 \times \ldots \times Dk \to E \cup \{\lambda\}$, wobei $m(d1,\ldots,dk)$ immer gleich λ ist (und $\lambda \notin E$). Der zweite Mechanismus, der als Output immer λ liefert, ist trivialerweise für jede gegebene policy zuverlässig, währenddessen das Programm selbst als Schutzmechanismus dies im allgemeinen nur in Ausnahmefällen sein wird.

Wie das Beispiel des immer zuverlässigen Schutzmechanismus mit $M(d1,\ldots,dk) = \lambda$ zeigt, ist die Zuverlässigkeit kein Kriterium für die Güte eines Schutzmechanismus. Von ganz entscheidender Bedeutung ist vielmehr die Frage nach der "Vollständigkeit" eines gegebenen Schutzmechanismus. Jones und Lipton definieren die Vollständigkeit als Halbordnung auf der Menge aller zuverlässigen Schutzmechanismen für ein Programm Q und eine Sicherheitspolicy I:

Seien M1 und M2 zwei zuverlässige Schutzmechanismen für das gleiche Programm Q und die gleiche policy I, wobei die Verletzungsanzeigen bei M1 und M2 aus der gleichen einelementigen Menge bestehen; M1 heißt mindestens so vollständig wie M2 - abgekürzt M1 >= M2 - wenn für alle Inputvektoren d gilt, wenn M2(d) = Q(d), dann M1(d) = Q(d). M1 heißt vollständiger als M2 (M1 > M2), wenn M1 >= M2 und es ein d gibt mit M1(d) = Q(d) und M2(d) $\neq$ Q(d).

Ein Mechanismus M1 ist also vollständiger als ein Mechanismus M2, wenn er für mehr Inputwerte die ursprünglichen Programmwerte angibt als M2. Für den trivialen Schutzmechanismus M(d) = λ für alle d gilt also, daß er der unvollständigste zuverlässige Schutzmechanismus für ein Programm Q und eine Sicherheitspolicy I ist. Von praktischem Interesse sind natürlich nur solche zuverlässigen Schutzmechanismen, die so wenige Verletzungsanzeigen wie möglich angeben, die also so vollständig wie möglich sind. In diesem Zusammenhang interessiert besonders die Frage, ob es für jedes Programm Q und jede policy I einen maximal vollständigen, zuverlässigen Schutzmechanismus gibt. Diese Frage läßt sich zwar mit "ja" beantworten, jedoch ist dieser maximale Schutzmechanismus, wie Jones und Lipton zeigen, nicht effektiv konstruierbar, d.h. es gibt keinen Algorithmus, der für jedes Programm Q und jede policy I den zugehörigen maximal zuverlässigen Schutzmechanismus ausgibt. Wie Ruzzo in einer privaten Mitteilung an die Autoren dieses Artikels feststellte, muß der maximal zuverlässige Schutzmechanismus für ein Programm Q und eine policy I nicht einmal eine rekursive Funktion sein, auch wenn Q und I rekursiv sind. Ein weiteres Ergebnis von Ruzzo ist die Unentscheidbarkeit der Frage, ob ein gegebener Schutzmechanismus M für ein Programm Q und eine policy I zuverlässig ist.

Diese Resultate über die Nicht-Konstruierbarkeit bzw. Nicht-Rekursivität eines maximal zuverlässigen Schutzmechanismus veranlassen Jones und Lipton als ein Beispiel für einen zuverlässigen, aber nicht maximalen Schutzmechanismus den sogenannten "Überwachungs-Schutzmechanismus" anzugeben.

Dieser Überwachungsmechanismus beschränkt sich auf Programme mit nur einem Output in einer einfachen Flußdiagrammsprache und policies der Form "allow(...)". Er führt - wie der Mechanismus von Denning - eine syntaktische Analyse des Informationsflusses durch. Aus diesem Grund hat er sehr viel Ähnlichkeit mit diesem, auch wenn es sich um einen Laufzeitmechanismus handelt, der zudem nicht auf einem Verbandsmodell mit dementsprechender policy basiert.

Beim Überwachungsmechanismus von Jones und Lipton wird für jede Variable (Input-, Output-, Programmvariable und Befehlszähler) des Originalprogramms eine entsprechende "Überwachungsvariable" hinzugefügt: Der Wert der Überwachungsvariablen ist die Menge aller Indizes der Inputvariablen, die diese Variable beeinflußt haben (Die Überwachungsvariable für den Befehlszähler wird bei konditionalen Strukturen beeinflußt; dies entspricht dem impliziten Fluß bei Denning). Jede Komponente des ursprünglichen Flußdiagrammprogramms wird durch eine modifizierte ersetzt, die neben der ursprünglichen Funktion noch den jeweiligen Überwachungsvariablen entsprechende Werte zuweist. Am Ende des Programms wird jedes Haltestatement durch eine Struktur ersetzt, die prüft, ob die Überwachungs-Ausgabevariable nur die Indizes der durch die policy allow(...) gegebenen Inputvariablen - also auch nur die entsprechende Information - enthält, und wenn nötig, den Wert der Ausgabevariablen auf λ, die Verletzungsanzeige, setzt.

Von Jones und Lipton wird nun gezeigt, daß der angegebene Überwachungsmechanismus für jedes Flußdiagramm-Programm und jede policy allow(...) zuverlässig ist. Eine Erweiterung des angegebenen Mechanismus ist sogar unter der Voraussetzung, daß die Laufzeiten der Programme beobachtbar sind und eventuell als verdeckte Kanäle für Informationsübertragung benutzt werden, zuverlässig.

Bemerkungen zum Modell

Das Modell von Jones und Lipton ist das einzige Informationsflußmodell, das die Begriffe "Sicherheitspolicy", "Schutzmechanismus" und "Zuverlässigkeit" (als die Brücke zwischen Sicherheitspolicy und Schutzmechanismus) formal, präzise und in allgemeingültiger Form definiert. Durch die Einführung der Vollständigkeitsrelation zwischen zwei Schutzmechanismen wird eine Bewertung der Güte und Präzision von Schutzmechanismen ermöglicht. In diesem Zusammenhang ist das Ergebnis der Existenz, aber Nicht-Konstruierbarkeit, eines maximal vollständigen, zuverlässigen Schutzmechanismus von besonderer Bedeutung. Der Überwachungsschutzmechanismus als ein zwar zuverlässiger, aber nicht maximaler Schutzmechanismus ist ein gutes Beispiel für syntaktische Laufzeit-Schutzmechanismen. Von besonderem Interesse für zukünftige Forschungen - auch in dieser Arbeit - ist nun die Frage nach der effektiven Konstruierbarkeit von zuverlässigen Mechanismen, die vollständiger sind als der angegebene syntaktische Überwachungsmechanismus; es wird sich zeigen, daß hier semantische Methoden interessante Ergebnisse liefern (vgl. 4.3.3 und 5.3.3).

Insgesamt erscheint das Modell von Jones und Lipton als Ausgangspunkt für die Definition der Begriffe policy, Mechanismus und Sicherheit sehr geeignet, auch wenn es an einigen Stellen - wie z.B. Untersuchungen bezüglich der Integrität von Variablen und Objekten, oder semantische Betrachtungen des Informationsflusses - noch erweiterungsbedürftig ist. Im nächsten Abschnitt soll nun eine Methode zur semantischen Analyse des Informationsflusses, das Modell von Cohen, näher betrachtet werden.

4.3.3 Das Modell von Cohen

Wie Denning und Jones und Lipton betrachtet Cohen in seinen Arbeiten <Coh77> und <Coh78> den Informationsfluß in Statements und Programmen einer Programmiersprache. Er stellt seine Untersuchungen allerdings unabhängig von irgendwelchen Sicherheitspolicies oder einer bestimmten Definition von Sicherheit an, sondern betrachtet ausschließlich die Möglichkeiten der Informationsübertragung in einem Programm. Die Bedeutung des Ansatzes von Cohen liegt darin, daß er wegen der mangelnden Präzision einer syntaktischen Betrachtungsweise (vgl 4.3.1 und 4.3.2), semantische Methoden zur Bestimmung des Informationsflusses benutzt.

Dieser semantische Ansatz von Cohen basiert auf Ideen der klassischen Informationstheorie: Informationsübertragung von einem Sender zu einem Empfänger findet nur statt, wenn eine Mannigfaltigkeit (d.h. verschiedene Zustände) des Senders zum Empfänger übermittelt werden kann (d.h. verschiedene Zustände des Empfängers zur Folge hat). Während sich die klassische Informationstheorie jedoch mit der Frage beschäftigt, wieviel Information über einen Kanal übertragen werden kann, interessiert beim Konzept der "strengen Abhängigkeit" von Cohen nur, ob Information übertragen werden kann, oder nicht; mit Hilfe dieses Konzepts gelingt es Cohen nun, Informationsfluß durch die funktionale Semantik der Operationen zu bestimmen, d.h. neben der syntaktischen Form der Operationen wird hier noch der Systemzustand (die Werte der Programmvariablen) bei der Untersuchung von Informationsfluß berücksichtigt. Insbesondere betrachtet Cohen auch den Einfluß von Zusicherungen auf den Informationsfluß; sie schränken die möglichen Informationspfade ein bzw. können sie sogar ganz eliminieren. Ein Beispiel hierfür wäre ein Konditionalstatement mit der Zusicherung, daß der Boole`sche Test immer den gleichen Wahrheitswert annimmt; hier kann der potentiell mögliche Informationsfluß über den Zweig des Statements, der laut Zusicherung nie ausgeführt wird, ignoriert werden.

In <Coh78> zeigt der Autor, daß das Konzept der strengen Abhängigkeit nur für eine Untermenge aller möglichen Zusicherungen geeignet ist, und führt die Ansätze "definitive Abhängigkeit" und "verbundene Abhängigkeit" ein, die die Frage der Informationsübertragung von einem deduktiven Standpunkt aus betrachten. Hier wird allerdings nur kurz auf diese Ansätze eingegangen, da im Kapitel 5 dieser Arbeit bei der Vorstellung des hier entwickelten Ansatzes eine detaillierte Betrachtung erfolgt.

Im Anhang von <Coh78> bringt Cohen schließlich Beweisregeln auf der Basis der strengen Abhängigkeit für die programmiersprachlichen Konstrukte Wertzuweisung, Sequenz, Alternation und Iteration, die die syntaktischen Regeln von Denning bzw. Andrews und Reitman bei einer semantischen Informationsflußkontrolle ersetzen können.

Modellkomponenten

Cohen geht in seinem Modell von einer einfachen abstrakten Maschine $\langle \mathcal{O}, \Sigma, \Delta \rangle$ aus. Hier ist $\mathcal{O}$ die Menge aller Objekte (Variablen des Programms), Σ die Menge aller Systemzustände und Δ die Menge der Operationen des Systems. Der Wert einer Variablen $a \in \mathcal{O}$ im Systemzustand $\sigma \in \Sigma$ wird als $\sigma.a$ bezeichnet. Jede Ausführung einer Operation $\delta \in \Delta$ ändert den momentanen Zustand des Systems, die Operationen aus Δ sind also Funktionen auf Σ. Mit $\delta(\sigma)$ wird der Zustand bezeichnet, der durch Anwendung der Operation $\delta \in \Delta$ auf den Zustand $\sigma \in \Sigma$ erreicht wurde. Dies läßt sich in der üblichen Weise auf Folgen von Operationen, d.h. Programme erweitern. Cohen schreibt $\sigma 1 \underset{a}{=} \sigma 2$, wenn die Zustände $\sigma 1$ und $\sigma 2$ bis auf den Wert der Variablen a gleich sind. Erweitert auf eine Menge von Variablen $A \subseteq \mathcal{O}$ gilt

$\sigma 1.A = \sigma 2.A$ gdw. $(\forall a \in A)\ (\sigma 1.a = \sigma 2.a)$ und

$\sigma 1 \underset{A}{=} \sigma 2$ gdw. $(\forall a \notin A)\ (\sigma 1.a = \sigma 2.a)$.

$\sigma 1 \,/\overline{A}\, \sigma 2$ ist derjenige Zustand σ^* mit $\sigma^* \underset{A}{=} \sigma 1$ und $\sigma^*.A = \sigma 2.A$, d.h. der Zustand, der für die Variablen aus A die Werte aus $\sigma 2$ und die restlichen Variablen die Werte aus $\sigma 1$ enthält. Mit diesen Definitionen ist Cohen nun in der Lage, den Begriff der strengen Abhängigkeit zu definieren.

Zusammenspiel der Grundkomponenten

Bei der Definition der strengen Abhängigkeit geht Cohen von Ideen der klassischen Informationstheorie aus: Information kann von einem Sender a zu einem Empfänger b übertragen werden, wenn eine Mannigfaltigkeit von a nach b übermittelt werden kann. Man betrachte ein Programm P und Variablen a und b. Haben verschiedene Anfangswerte von a nach der Ausführung von P verschiedene Endwerte von b zur Folge, so wird die Mannigfaltigkeit von a nach b übermittelt und somit Information übertragen. Um zu zeigen, daß eine Informationsübertragung von a nach b stattfindet, müssen also nur zwei verschiedene Werte von a gefunden werden, die nach der Ausführung von P verschiedene Werte von b zur Folge haben. Diese zwei verschiedenen Werte von a sind durch das Finden zweier Zustände $\sigma 1$ und $\sigma 2$ mit $\sigma 1 \underset{a}{=} \sigma 2$ gegeben. Man beachte, daß sich $\sigma 1$ und $\sigma 2$ für den Beweis einer Informationsübertragung von a nach b nur im Wert von a unterscheiden dürfen, weil ja sonst die verschiedenen Werte von b auch aus anderen Unterschieden zwischen $\sigma 1$ und $\sigma 2$ resultieren könnten. Formal stellt sich dies folgendermaßen dar:

Eine Variable b hängt bei der Ausführung eines Programms P streng von einer Variablen a ab, kurz $a \,||>^P b$, wenn gilt

$$(\exists\, \sigma 1, \sigma 2)\ (\sigma 1 \underset{a}{=} \sigma 2 \wedge P(\sigma 1).b \neq P(\sigma 2).b);$$

dies läßt sich auch auf Mengen von Variablen A verallgemeinern $(A \,||>^P b)$, d.h. auf die Informationsübertragung von einer Menge von Variablen A zu einer Variable b.

Man beachte, daß im Konzept der strengen Abhängigkeit keine Unterscheidung zwischen "explizitem" und "implizitem" Fluß notwendig ist. Im einfachen Beispiel "P: if a then b:=x" kann leicht $a \;||>^{P} b$ gezeigt werden, d.h. es wird Information von a nach b übertragen (natürlich gilt auch $x \;||>^{P} b$).

Von besonderem Interesse sind Untersuchungen des Informationsflusses in Programmen, wenn Zusicherungen über die Werte von Variablen berücksichtigt werden. Solche Zusicherungen schränken den möglichen Informationsfluß im allgemeinen ein, sie können ihn jedenfalls nie vergrössern. Zum Beispiel wird im Programm "P: if m=0 then b:=a+1" keine Information von a nach b übertragen, wenn die Zusicherung m=0 vor der Ausführung von P erfüllt ist. Cohen erweitert seinen Ansatz um die Möglichkeit, Zusicherungen mit zu berücksichtigen. Er definiert

$$\sigma 1 \overset{\Phi}{\underset{A}{=}} \sigma 2 \text{ gdw. } (\sigma 1 \overset{}{\underset{A}{=}} \sigma 2) \wedge \Phi(\sigma 1) \wedge \Phi(\sigma 2) \text{ und}$$

$$A \;||>^{P}_{\Phi} b \text{ gdw. } (\exists\, \sigma 1, \sigma 2)\; (\sigma 1 \overset{\Phi}{\underset{A}{=}} \sigma 2 \wedge P(\sigma 1).b \neq P(\sigma 2).b).$$

Der Unterschied zu den bisherigen Definitionen besteht darin, daß nur Zustände berücksichtigt werden, die die Eingangszusicherung Φ erfüllen. Im Beispiel "P: if m=0 then b:=a+1" gilt nun unter der Zusicherung

Φ: $m \neq 0$, daß $\neg(a \;||>^{P}_{\Phi} b)$, obwohl $a \;||>^{P} b$ gilt.

Zur Untersuchung der Informationsübertragung in Programmen müssen natürlich auch Regeln für die Informationsübertragung bei der sequentiellen Ausführung von Programmstatements existieren. Diese Regeln sollen hier vorstellt und an einigen Beispielen demonstriert werden, weil sich hier der Unterschied zwischen syntaktischer und semantischer Betrachtungsweise besonders deutlich zeigt.

Cohen definiert zunächst $A \,||\!>^{P}* := \{m \mid A \,||\!>^{P} m\}$ und

$A \,||\!>^{P*}_{\Phi} := \{m \mid A \,||\!>^{P}_{\Phi} m\}$, d.h. $A \,||\!>^{P*}_{\Phi}$ ist die Menge aller Variablen, die von A bei der Ausführung von P unter der Eingangszusicherung Φ streng abhängen. Mit diesen Definitionen läßt sich nun der folgende Satz beweisen:

"Sei $P = [S;S']$ und $\Phi \rightarrow \Phi' \circ S$ (d.h. P ist die sequentielle Ausführung der Programmstücke S und S', und falls Φ vor der Ausführung von S gilt, so gilt Φ' nach der Ausführung von S), und gelte $M = A \,||\!>^{S*}_{\Phi}$. Dann folgt aus

$\neg M \,||\!>^{S'}_{\Phi'} b$, daß $\neg A \,||\!>^{P}_{\Phi} b$ gilt".

Damit läßt sich also die Abwesenheit eines Informationspfades in einem Gesamtprogramm $P = [S;S']$ durch den Beweis der Abwesenheit eines solchen Pfads im Teilprogramm S' zeigen (dies läßt sich analog auch für das Teilprogramm S zeigen). Leider gilt die Umkehrung dieses Satzes - dies

wäre "aus $M \,||\!>^{S'}_{\Phi'} b$ folgt $A \,||\!>^{P}_{\Phi} b$"- nicht; d.h. für die Informationsübertragung bei semantischer Betrachtungsweise gilt - im Gegensatz zu einer syntaktischen Betrachtung - die Transitivität nicht. Es gibt also Programme $P = [S;S']$ mit

$M = A \,||\!>^{S*}_{\Phi}$, so daß für jedes Φ' mit $(\Phi \rightarrow \Phi' \circ S)$ gilt

$M \,||\!>^{S'}_{\Phi'} b$, obwohl $\neg A \,||\!>^{P}_{\Phi} b$, d.h. es findet eine Informationsübertragung von A nach M in S statt und von M nach b in S', und trotzdem fließt keine Information von A nach b. Ein einfaches Beispiel hierfür ist das Programm $P = [S;S']$ mit "S: if $q \neq 0$ then $m := a$" und "S': if $q = 0$ then $b := m$"; hier gilt zwar $a \,||\!>^{S} m$ und $m \,||\!>^{S'} b$, aber nicht $a \,||\!>^{P} b$.

Um in solchen Fällen die Abwesenheit eines Informationspfades nachweisen zu können, führt Cohen die Technik der "Trennung der Mannigfaltigkeit" (separation of variety) ein. Diese Methode soll nun zu ihrem besseren Verständnis erst am angegebenen Beispiel erläutert werden, bevor eine formale Definition gegeben wird.

Man betrachte die beiden Zusicherungen $\Phi 1$: q=0 und $\Phi 2$: q≠0. Man sieht deutlich, daß sowohl

$$\neg a \;||>^{P_b}_{\Phi 1}, \text{ als auch } \neg a \;||>^{P_b}_{\Phi 2}$$

gilt, da bei Gültigkeit von $\Phi 1$ in S keine Information von a nach m und bei Gültigkeit von $\Phi 2$ in S` keine Information von m nach b übertragen wird; dies läßt sich formal durch den angeführten Satz für die sequentielle Ausführung zweier Teilprogramme zeigen. Da $\Phi 1$ und $\Phi 2$ alle möglichen Fälle abdecken, gilt für dieses Beispiel auch $\neg a \;||>^{P_b}$. Natürlich läßt sich diese Schlußfolgerung im angegebenen Beispiel nur deswegen treffen, weil $\Phi 1$ und $\Phi 2$ gewisse Voraussetzungen erfüllen. Die erste Voraussetzung ist, daß die Zusicherungen $\{\Phi i\}$ eine sogenannte Hülle bilden, d.h. sämtliche möglichen Fälle abdecken; formal läßt sich dies durch $(\forall \sigma\ \exists i)\ (\Phi i(\sigma))$ ausdrücken. Die zweite Voraussetzung ist, daß die Zusicherungen $\{\Phi i\}$ unabhängig von der zu untersuchenden Informationsquelle formuliert sind. Man definiert hierfür:

Eine Zusicherung ρ ist A-unabhängig, wenn gilt

$$(\forall \sigma 1, \sigma 2)\ (\sigma 1 \underset{A}{=} \sigma 2 \rightarrow \rho(\sigma 1) = \rho(\sigma 2)).$$

Die Technik der Trennung der Mannigfaltigkeit läßt sich nun folgendermaßen formulieren:

Sei $\{\Phi i\}$ eine Hülle und sei für alle i Φi A-unabhängig,

dann gilt $(\forall i)(\neg A \;||>^{P_b}_{\Phi i}) \rightarrow (\neg A \;||>^{P_b})$.

An dieser Stelle sei erwähnt, daß der Autor im Anhang der Arbeit <Coh78>, analog zu den Regeln für eine sequentielle Ausführung von Teilprogrammen, Beweisregeln für die Betrachtung eines semantischen Informationsflusses in konditionalen Statements und Iterationen gemäß den Kriterien der strengen Abhängigkeit angibt (vgl. <Coh78> S 328 ff).

Das Konzept der strengen Abhängigkeit basiert stillschweigend darauf, daß die betrachteten Zusicherungen eine weitere Voraussetzung, die der Autonomie, erfüllen. Informal heißt dies, daß bei der Untersuchung, ob $A \;||\!>^{P}_{\Phi}\; b$ gilt, die Zusicherung Φ nicht Variablen aus A mit solchen nicht aus A verknüpft; es läßt sich zeigen, daß eine A-autonome Zusicherung Φ von der Form $\Phi 1 \wedge \Phi 2$ sein muß, wobei $\Phi 1$ nur Variablen aus A betrifft und $\Phi 2$ nur andere (d.h. A-unabhängig ist).

In <Coh78> definiert Cohen nun die zwei alternativen Konzepte "definitive Abhängigkeit" und "verbundene Abhängigkeit", die beide auf einem deduktiven Standpunkt basieren: Information kann während der Ausführung eines Programms P von A nach b übertragen werden, wenn ein Wert von b nach der Ausführung von P dazu benutzt werden kann, Eigenschaften der ursprünglichen Werte von A abzuleiten. Eine definitive Abhängigkeit der Variablen b von den Variablen aus A bei der Ausführung von P liegt dann vor, wenn aus einem Wert von b nach der Ausführung des Programms P etwas definitives über die Werte der Variablen aus A geschlossen werden kann.

Oft ist ein definitiver Schluß über die Werte von A nur dann möglich, wenn eine zusätzliche Information, die nicht A betrifft, gegeben ist. Eine solche Zusicherung über A, die nur in Verbindung mit einer weiteren Zusicherung einen definitiven Schluß über A erlaubt, heißt A-verbunden. Darauf aufbauend wird nun von Cohen die "verbundene Abhängigkeit" definiert. Cohen zeigt dann, daß bei der Verwendung von autonomen Zusicherungen die Konzepte der strengen Abhängigkeit und der verbundenen Abhängigkeit äquivalent sind. Ein kleines Beispiel soll abschließend die Unterschiede der drei

vorgestellten Konzepte verdeutlichen:

Sei "P: b:=a1 + a2" und Φ: a1 = a3. In allen drei Konzepten läßt sich zeigen, daß bei der Ausführung von P unter der Zusicherung Φ Information von {a1,a2,a3} nach b übertragen wird. Betrachtet man die Frage, ob Information von a1 allein nach b übertragen wird, so lautet die Antwort nur beim Ansatz der verbundenen Abhängigkeit "ja". Das Konzept der strengen Abhängigkeit versagt, da es sich bei Φ um eine nicht autonome Zusicherung handelt, und eine definitive Abhängigkeit liegt nicht vor, da der Wert von b nach der Ausführung von P nichts definitives über den Wert von a1 vor der Ausführung von P aussagt.

Bemerkungen zum Modell

Die Bedeutung des Modells von Cohen liegt in der semantischen Betrachtungsweise von Informationsfluß. Semantische Methoden ermöglichen eine größere Präzision bei der Beantwortung der Frage nach Informationsfluß als syntaktische - in dem Sinn, daß sie bei Zugrundelegung des entsprechenden Begriffs von Jones und Lipton vollständiger sind. Dieses Mehr an Vollständigkeit ergibt sich aus der zusätzlichen Betrachtung des Systemzustands (d.h. der Werte der Variablen) und der Untersuchung des Einflusses von Zusicherungen auf den Informationsfluß in Programmen.

Insgesamt betrachtet müssen die Arbeiten von Cohen als äußerst wichtiger Beitrag zum Thema Informationsfluß in Programmen angesehen werden. Leider fanden die hier vorgestellten Ansätze - wohl wegen der recht komplexen Darstellung - noch keine Anwendungen in der Praxis. Auch wenn die Analyse von Informationsfluß in Programmen mit syntaktischen Methoden wesentlich einfacher ist, wird man sicher in Zukunft bei der Konstruktion zuverlässiger Systeme an semantischen Methoden nicht vorbeikommen (vgl. Kapitel 5 dieser Arbeit).

4.3.4 Weitere Informationsflußmodelle

Die letzten drei Abschnitte präsentierten sehr unterschiedliche Ansätze, das Problem der Bestimmung von Informationsfluß in Programmen zu lösen. Die Verschiedenheit der Methoden zur Untersuchung von Informationsflüssen setzt sich auch in den, in diesem Abschnitt kurz vorzustellenden, Modellen fort. Aufbauend auf dem Verbandsmodell von Denning entwickelten Andrews und Reitman eine weitere syntaktische Methode zur Informationsflußbestimmung. Das nächste Modell stammt von Furtek und Millen; in diesem semantischen Modell wird die Informationsflußfrage von einem deduktiven Standpunkt aus betrachtet. Landauer und Crocker entwickelten schließlich ebenfalls einen semantischen Ansatz zur Bestimmung von Informationsfluß, bei dem der Informationsgehalt der einzelnen Variablen und dessen Veränderung bei Zustandsübergängen im Vordergrund steht.

Die Methode von Andrews und Reitman

Andrews und Reitman präsentieren in ihren Arbeiten <Rei79> und <And80> Regeln zum Nachweis von Informationsfluß in Programmen. Diese Regeln ähneln vom Aufbau her sehr stark den Regeln für Korrektheitsbeweise, wie sie von Hoare <Hoa69> angegeben wurden; die mit ihnen durchzuführende Informationsflußanalyse ist syntaktischer Natur, und ist, von einigen kleinen Erweiterungen abgesehen, prinzipiell gleich dem Verbandsmodell von Denning.

Die Autoren betrachten Programme, die drei relevante Komponenten besitzen: Variable, die Information enthalten, einen "Informationszustand", der die aktuelle Sicherheitsklassifikation der Variablen angibt, und Statements, die Variable modifizieren und damit auch den Informationszustand ändern. Auch in diesem Modell formen die Sicherheitsklassen einen Verband, jedoch betrachten Andrews und Reitman nicht nur eine statische Bindung von Sicherheitsklassen an Variablen, sondern lassen eine dynamische Bindung zu.

Bei der Untersuchung des Informationsflusses wird wie bei Denning zwischen implizitem und explizitem Informationsfluß (bei Andrews und Reitman "direkter" und "indirekter" Informationsfluß genannt) in Programmstatements unterschieden. Beim impliziten Fluß wird allerdings nochmals in zwei Untertypen differenziert, lokale und globale Flüsse. Ein lokaler Fluß ist ein impliziter Fluß innerhalb eines Statements, wie z.B. der Informationsfluß von der Bedingung zu den Zweigen eines Konditionalstatements. Globale Flüsse sind dagegen Informationsflüsse zwischen verschiedenen Statements. Als Beispiel hierfür wäre der Informationsfluß von einer Schleife zu den nachfolgenden Statements - abhängig von der Terminierung der Schleife oder deren Nichtterminierung - zu nennen. Nimmt man in einem Programm an, oder beweist man, daß alle Schleifen terminieren, so fällt diese Art von globalem Fluß weg.

Die Autoren geben nun auf dieser Basis für die verschiedenen Statementtypen Beweisregeln an. Interessant ist an dieser Stelle, daß sie auch den Informationsfluß betrachten, der sich durch Synchronisation mehrerer Programme über eine Semaphore (es handelt sich hier auch um einen globalen Fluß) ergibt. Zuletzt stellen die Verfasser in <And80> einen Vergleich zwischen den in ihren Arbeiten präsentierten Regeln zum Nachweis von Informationsfluß und den Regeln zum Beweis der Korrektheit eines Programms auf, und diskutieren kurz die Möglichkeit einer Verbindung dieser beiden Techniken, die eventuell eine Informationsflußanalyse auf semantischer Basis durchführen könnte.

Das Modell von Furtek und Millen

In <Fur78>, <Mil78> und <Mil81> stellen die beiden Autoren ihre Theorie der "constraints" vor; in diesem Ansatz wird versucht, mit semantischen Methoden (d.h. unter Berücksichtigung des Systemzustands) Informationsfluß zu untersuchen, wobei ein deduktiver Standpunkt eingenommen wird: Furtek und Millen betrachten Programme als Folgen von

Transitionen und unterscheiden zwischen Eingabe-, Ausgabe- und Systemvariablen. Wie bei den Modellen zur militärischen Sicherheit bzw. im Verbandsmodell von Denning werden nun den Variablen bestimmte Sicherheitsstufen eines Verbands zugeordnet. Die Autoren stellen dann die Frage, ob sich durch die Beobachtung von Ein- und Ausgabevariablen einer bestimmten Sicherheitsstufe s (oder einer niedrigeren) und die Beeinflussung von Eingabevariablen einer beliebigen Sicherheitsstufe etwas über die Werte von Eingabevariablen einer höheren Stufe als s (bzw. einer zu s inkomparablen Stufe) ableiten läßt; ein Programm (bzw. System) wird als sicher bezeichnet, wenn dies nicht der Fall ist. Der hier verwendete Sicherheitsbegriff ist also identisch mit dem der militärischen Sicherheit, durch die semantische Analyse wird jedoch bei diesem Ansatz eine wesentlich präzisere Definition ermöglicht, als bei den klassischen Modellen zur militärischen Sicherheit oder dem Verbandsmodell von Denning.

Zur Klärung der Sicherheitsfrage benutzen Furtek und Millen nun ihre Theorie der constraints. Ein constraint ist dabei eine Folge von Zuständen bei der Ausführung eines Programms, die aufgrund der Programmdefinition nicht vorkommen kann. Von besonderem Intetesse sind hier die sogenannten "prime constraints", da bei ihnen eine Kenntnis der Werte aller bis auf eine Variablen, die darin vorkommen, einen Schluß auf den Wert dieser einen Variablen ermöglicht. Millen gibt nun mit Hilfe der entwickelten Notation der constraints eine notwendige und hinreichende Bedingung für die Sicherheit eines Systems an, die allerdings die Untersuchung sämtlicher prime constraints des Systems erfordert.

In der Praxis ist dies jedoch kaum anwendbar, da die prime constraints - selbst für einfache Systeme - beliebig lang werden können und auch beliebig viele existieren können. Millen entwickelt zwar noch eine einfachere hinreichende Bedingung für die Sicherheit von Systemen, die durch constraints einer bestimmten Form charakterisiert werden können, jedoch erscheint auch diese Bedingung für praktische Anwendungen als zu kompliziert.

Das Modell von Landauer und Crocker

Der momentan neueste Ansatz auf dem Gebiet der Informationsflußmodelle stammt von Landauer und Crocker (vgl. <Lan82>). Die beiden Autoren präsentieren einen Mechanismus zur Informationsflußanalyse mit dem Ziel, diesen Mechanismus in ein Programm-Verifikationssystem zu integrieren. Die vorgestellte Methode ist im Prinzip eine Verbindung der semantischen Informationsflußbetrachtungen von Cohen und der axiomatischen Methoden von Andrews und Reitman.

Landauer und Crocker betrachten in ihrem Modell den abstrakten Informationsgehalt von Variablen und dessen Veränderung bei Zustandsübergängen; sie gehen hierbei von einer abstrakten Maschine mit einer endlichen Menge von Zustandsvariablen aus. Wie Cohen definieren die Autoren den Begrif "Information" als Variation, d.h. eine Informationsübertragung findet statt, wenn eine Mannigfaltigkeit des Senders zum Empfänger übertragen werden kann. Der Informationsgehalt von Variablen wird nun als eine Menge enthalten in einem abstrakten "Informationsraum" definiert; es handelt sich hier um eine rein abstrakte Definition, ohne Festlegung der Art der Elemente der Informationsgehalt-Menge einer Variablen. Aufbauend auf dieser Definition führen Landauer und Crocker dann den Informationsgehalt von Ausdrücken (z.B. Konstante, Variable, Feldelemente oder Funktionsauswertungen wie Boole`sche und arithmetische Ausdrücke) ein. Sie betrachten nun den Informationsfluß in Programmstatements durch Untersuchungen der Veränderung des Informationsgehalts der Variablen bzw. Ausdrücke des Programms. Hierbei greifen sie auf einen Mechanismus zur Beschreibung der Semantik von abstrakten Maschinen - die sogenannten "state deltas" - zurück.

Ein state delta ist eine Formel "P -> (M)Q", die einen Zustandübergang der abstrakten Maschine beschreibt; P ist hier die Precondition, M die Liste der Variablen, die vom Zustandsübergang verändert werden, und Q die Postcondition für den Zustandsübergang, die sich nur auf Variablen aus M

bezieht. Diese state deltas - ursprünglich von Crocker nur zur Beschreibung der Semantik von Programmsegmenten entwickelt - werden nun in <Lan82> um die Möglichkeit, Informationsfluß in Programmsegmenten zu analysieren, erweitert. Ähnlich wie bei Andrews und Reitman definieren auch Landauer und Crocker zwei zusätzliche "Informationsvariable" für lokale und globale Flüsse (die hier allerdings semantisch betrachtet werden).

Mit Hilfe des eingeführten Mechanismus - einer Kombination aus Informationsflußanalyse und Zusicherungen über Werte von Variablen bzw. Ausdrücken - untersuchen nun die Autoren den Informationsfluß in Beispielprogrammen einer einfachen Programmiersprache; bei den gegebenen Beispielen handelt es sich durchwegs um solche Statements, in denen eine rein syntaktische Analyse versagt, wie sie z.B. auch von Cohen angegeben werden.

Das Modell von Landauer und Crocker ist ähnlich wie das Modell von Cohen ein recht interessanter Ansatz zur semantischen Informationsflußanalyse. Wie allerdings schon die Beispiele in <Lan82> zeigen, wird die Analyse bei komplexeren Statements sehr kompliziert und umständlich, so daß erst eine Integration in ein rechnerunterstütztes Verifikationssystem - wie es das Vorhaben der Autoren ist - eine Anwendung dieses Verfahrens in der Praxis ermöglichen wird.

Bemerkungen

Die in diesem Abschnitt vorgestellten Modelle zur Informationsflußanalyse sind - bis auf das Modell von Furtek und Millen - im Prinzip Erweiterungen der in 4.3.1 bis 4.3.3 vorgestellten Methoden. Die Modelle von Andrews und Reitman und von Landauer und Crocker sind dabei mit dem Ziel einer späteren Integration der vorgestellten Ansätze in rechnerunterstützte Programm-Verifikationssysteme entwickelt worden, so daß eine Anwendungsmöglichkeit in der Praxis für diese beiden Methoden zu einem späteren Zeitpunkt durchaus

gegeben ist. Das Modell von Furtek und Millen erscheint dafür nicht besonders geeignet, da die Theorie der Constraints sich nicht unbedingt am normalen und intuitiven Verständnis von Informationsfluß orientiert.

4.3.5 Zusammenfassung

Die ersten Modelle, die eine Kontrolle des Informationsflusses versuchten, waren die in 4.2 vorgestellten Modelle für militärische Sicherheit. Da diese jeden potentiellen Informationsfluß in einem System und nur feste Zugriffsarten betrachteten, und sich außerdem auf Anwendungen der militärischen Sicherheitspolicy beschränkten, eigneten sie sich nicht für eine präzise Kontrolle des Informationsflusses in Programmen. Erst die in diesem Abschnitt präsentierten Informationsflußmodelle brachten hier Ansätze zur Lösung dieses Problems.

Die ersten Informationsflußmodelle arbeiteten mit syntaktischen Methoden, wie das Verbandsmodell von Denning. Die Arbeiten von Cohen und Furtek und Millen führten dann wegen der mangelnden Präzision der rein syntaktischen Analyse semantische Methoden ein. Die neuesten Arbeiten auf dem Gebiet der Informationsflußmodelle - die Modelle von Andrews und Reitman sowie Landauer und Crocker - wurden teilweise schon mit dem Ziel einer Integration in rechnerunterstützte Programm-Verifikationssysteme konstruiert - eine wichtige Voraussetzung für deren Anwendbarkeit in der Praxis.

Das einzige Modell, das die Begriffe "Sicherheitspolicy", "Schutzmechanismus" und "Sicherheit" (d.h. die Zuverlässigkeit eines Schutzmechanismus für ein gegebenes Programm und eine gegebene policy) formal und allgemeingültig definiert, ist jedoch das Modell von Jones und Lipton. Aus diesem Grund könnte es als Basis für den Teilbereich Informationsfluß in einem formalen Modell zur Objektverwaltung herangezogen werden.

Insgesamt sind alle im Abschnitt 4.3 vorgestellten Informationsflußmodelle unvollständig im Sinne eines allgemeinen Objektverwaltungsmodells, da z.B Aspekte des Zugriffsschutzes, der Weitergabe von Rechten, der Datenintegrität oder gar der Synchronisation nicht berücksichtigt werden. Auch die Spezifikations- und Verifikationsmöglichkeiten dieser Modelle beziehen sich nur auf das Gebiet des Informationsflusses.

Im folgenden Abschnitt sollen nun zwei Modelle vorgestellt werden, die sich nicht mehr in eine der drei bisher vorgestellten Modellkategorien einordnen lassen, sondern versuchen, das Thema Schutz und Sicherheit allgemeiner anzugehen; natürlich sind diese Modelle dann auch vollständiger im Sinn von Abschnitt 3.5. Beiden Ansätzen ist gemeinsam, daß sie nicht älter als 1981 sind; dies deutet darauf hin, daß sich erst jetzt langsam eine Betrachtungsweise weg vom speziellen Schutzproblem und hin zu einem allgemeinen Modell durchsetzt.

4.4 Allgemeine Modelle

Die bisher in den Abschnitten 4.1 bis 4.3 vorgestellten Modelle lassen sich alle eindeutig in eine der drei Klassen von Schutzmodellen Objektschutzmodelle, Modelle zur militärischen Sicherheit und Informationsflußmodelle einordnen. Zwar liegen gewisse Modelle, wie z.B. die hierarchischen Take-Grant-Modelle von Bishop oder das Modell von Feiertag, an der Grenze zwischen zwei Gebieten, und decken diese beiden Gebiete auch teilweise ab, jedoch bieten selbst diese Ansätze - zum Teil auch noch durch andere Mängel bedingt - nur eine eingeschränkte Sicht der Problemkreise Schutz und Sicherheit.

Allgemein läßt sich über Objektschutzmodelle sagen, daß sie keinerlei Informationsbetrachtungen bieten, während Informationsflußmodelle weder Objektschutz noch Schutz vor unerlaubter Modifikation von Information berücksichtigen. Die Modelle für militärische Sicherheit hingegen betrachten nur eine äußerst eingeschränkte policy, die weder für ein allgemeines Modell noch für die Praxis brauchbar ist. Die bisher vorgestellten Modelle sind also alle mehr oder weniger unvollständig und eingeschränkt, was den gesamten Problemkreis Schutz und Sicherheit betrifft.

In diesem Abschnitt werden nun zwei Modelle präsentiert, die auf unterschiedliche Art und Weise versuchen, einen allgemeinen Ansatz für Schutzmodelle - weg von einer Betrachtung spezieller Schutzprobleme - darzulegen. In <Gog82> präsentieren Goguen und Meseguer einen Ansatz, der es erlaubt, allgemeine Sicherheitspolicies zu definieren; die speziellen Sicherheitspolicies der bisher vorgestellten Modelle lassen sich mit dem dargelegten Formalismus sehr einfach formulieren. Einen völlig anderen Weg schlägt Stoughton in seiner Arbeit <Sto81> ein: er präsentiert ein Modell, das sowohl Aspekte der Zugriffskontrolle als auch der Informationsflußkontrolle berücksichtigt.

Man sollte sich allerdings an dieser Stelle im klaren darüber sein, daß beide Ansätze nur ein erster Schritt dahin sind, die ganze Problematik des Gebiets Schutz und Sicherheit in einem Modell abzuhandeln, und daß zur Erreichung dieses Ziels noch sehr viel Forschungsarbeit zu leisten ist.

4.4.1 Das Modell von Goguen und Meseguer

Goguen und Meseguer stellen in ihrer Arbeit einen einfachen automatentheoretischen Ansatz zur Modellierung sicherer Systeme vor, der sich nicht auf die Anwendungen bei Betriebssystemen beschränkt, sondern auf allgemeine SW-Systeme übertragbar ist. Von besonderer Bedeutung ist, daß das präsentierte Modell ein allgemeines Konzept für Sicherheitspolicies beinhaltet, und somit eine Spezifikation beliebiger policies gestattet. Die Autoren unterscheiden in diesem Zusammenhang strikt zwischen der Sicherheitspolicy und dem eigentlichen System, das durch ein Modell (wie z.B. eine High-level Spezifikation einer abstrakten Maschine) gegeben ist. Daraus resultiert logischerweise ein klarer Sicherheitsbegriff, d.h. ein System ist sicher, wenn das zugeordnete Modell, bzw. die entsprechende abstrakte Maschine, die gegebene Sicherheitspolicy erfüllt. Der eigentliche Verifikationsschritt wird allerdings von Goguen und Meseguer nicht näher ausgeführt, sie schlagen nur den HDM-Ansatz (vgl. 4.2.2) als eine Möglichkeit der Vorgehensweise vor, und verweisen auf zukünftige Arbeiten in dieser Richtung.

Modellkomponenten

Grundlage des Modells von Goguen und Meseguer ist eine abstrakte Maschine

M = (S,U,SC,Out,Capt,CC,out,do,cdo,t0,s0).

S ist hierbei eine Menge von (Daten-) Zuständen (die möglichen Werte aller Objekte des Systems), U eine Menge von

Usern (Subjekte). SC bezeichnet eine Menge von "state commands", d.h. Operationen zur Veränderung des Datenzustands, während Out die Menge der Outputs des Systems ist. Capt ist eine Menge von "capability tables", die die Schutzkomponenten bzw. den Schutzzustand des Systems darstellen, und CC sind "capability commands", d.h. Operationen zur Veränderung der capability tables bzw. des Schutzzustands. sO und tO bezeichnen dann die Anfangszustände von S und Capt, d.h $sO \in S \wedge tO \in Capt$.

Die Dynamik des Systems bestimmen nun die drei Funktionen out, do und cdo, wobei gilt

$out: S \times Capt \times U \rightarrow Out$
$do : S \times Capt \times U \times SC \rightarrow S$
$cdo: Capt \times U \times CC \rightarrow Capt.$

out ist also eine Ausgabefunktion, die jedem Zustand (Datenzustand und Schutzzustand) und User eine bestimmte Ausgabe zuordnet. Die Funktionen do und cdo bewirken nun die Zustandsübergänge von S und Capt, je nachdem, ob ein state command aus SC oder ein capability command aus CC vom Benutzer $u \in U$ aufgerufen wurde. Erweitert man die beiden Funktionen do und cdo auf den kombinierten Zustandsraum S x Capt, faßt sie dann zusammen und definiert $C := SC \cup CC$, so ergibt sich eine System-Zustandsübergangsfunktion csdo mit

$csdo: S \times Capt \times U \times C \rightarrow S \times Capt$
$csdo(s,t,u,c) := (do(s,t,u,c),t)$ falls $c \in SC$
$csdo(s,t,u,c) := (s,cdo(t,u,c))$ falls $c \in CC$

Die von Goguen und Meseguer definierte abstrakte Maschine besitzt also als Zustandsraum S x Capt, als Inputraum U x C und als Outputraum Out. Die Autoren erweitern nun die Zustandsübergangsfunktion csdo in der üblichen Weise auf die Menge $(U \times C)^*$ und definieren

[[w]] := csdo(s0,t0,w) für $w \in (U \times C)^*$,
d.h [[w]] ist derjenige Zustand, der ausgehend vom Anfangszustand (s0,t0) durch Eingabe des Worts $w \in (U \times C)^*$ erreicht wird.

Zusammenspiel der Grundkomponenten

Mit den bisherigen Definitionen sind Goguen und Meseguer nun in der Lage, den Begriff "Sicherheitspolicy" einzuführen; sie definieren eine Sicherheitspolicy als eine Menge von "Nichtbeeinflussungs-Zusicherungen", wobei jede solche Zusicherung angibt, daß eine bestimmte Gruppe von Usern (Subjekten) durch die Ausführung bestimmter Kommandos keinen Einfluß darauf hat, was eine bestimmte andere Usergruppe sieht. Die Autoren unterscheiden dabei zwischen statischen policies (die sich nicht verändern, also unabhängig vom Zustand der capability tables sind), und dynamischen policies. Im folgenden sollen nun diese Definitionen präsentiert und zu ihrem besseren Verständnis an einigen konkreten Beispielen demonstriert werden.

Zuerst werden dafür noch einige Hilfsdefinitionen benötigt:

- Sei $w \in (U \times C)^*$; [[w]] u := out([[w]] ,u) bezeichnet die Ausgabe an das Subjekt u nach der Ausführung von w.

- Sei $G \subseteq U$ eine "Gruppe" von Subjekten, $A \subseteq C$ eine Untermenge von Kommandos (von den Autoren "ability" genannt) und sei $w \in (U \times C)^*$ ein Eingabewort.
  * pG(w) ist die Unterfolge von w, in der alle Paare (u,c) mit $u \in G$ eliminiert sind.
  * pA(w) ist die Unterfolge von w, in der alle Paare (u,c) mit $c \in A$ eliminiert sind.
  * pG,A(w) ist die Unterfolge von w, in der alle Paare (u,c) mit $u \in G \wedge c \in A$ eliminiert sind

Ein Beispiel hierfür wäre (wenn G={u,v} und A={c1,c2})
pG,A((u`,c1)(u,c3)(u,c2)(v`,c1)(v,c1)) =
(u`,c1)(u,c3)(v`,c1).

Aufbauend auf den Hilfsdefinitionen definieren die Autoren nun drei verschiedene Formen von Nichtbeeinflussungszusicherungen für eine gegebene abstrakte Maschine M:

- Seien G und G` Usergruppen. G beeinflußt G` nicht, oder kurz G:|G`, gdw. für alle w ∈ (U x C)* und alle u ∈ G` gilt [[w]] u = [[pG(w)]] u.

- Sei A eine ability und G` eine Usergruppe. A beeinflußt G` nicht, oder kurz A:|G`, gdw. für alle w ∈ (U x C)* und alle u ∈ G` gilt [[w]] u = [[pA(w)]] u.

- Sei G eine Usergruppe mit ability A und G` eine Usergruppe. Die Gruppe G mit ability A beeinflußt G` nicht, oder kurz A,G:|G`, gdw. für alle w ∈ (U x C)* und alle u ∈ G` gilt [[w]] u = [[pG,A(w)]] u.

Diese Nichtbeeinflussungszusicherungen sind nun die Grundlage für die Einführung des Begriffs "Sicherheitspolicy"; eine (statische) Sicherheitspolicy wird nämlich einfach als eine Menge von Nichtbeeinflussungszusicherungen definiert.

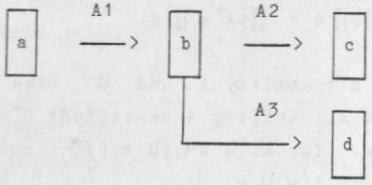
Um diese Definition von Sicherheitspolicies näher zu erläutern, insbesondere auch um die Allgemeinheit des Ansatzes von Goguen und Meseguer zu zeigen, soll nun im folgenden die bekannte policy für militärische Sicherheit und ein Beispiel für eine Informationsflußpolicy mit den eingeführten Formalismen dargestellt werden.

Für das Beispiel der militärischen Sicherheitspolicy werde von einer geordneten Menge von Sicherheitsebenen L (z.B unclassified, confidential, secret, top-secret) mit einer Ordnungsrelation "<" ausgegangen (vgl. 4.2). Eine Zuordnungsfunktion Level : U --> L ordne den Usern des Systems eine Sicherheitsstufe zu. Sei für x ∈ L

$U[-\infty,x] := \{u \in U \mid Level(u) \leq x\}$ und
$U[x,+\infty] := \{u \in U \mid Level(u) \geq x\}$.

Ein System M ist dann militärisch sicher, wenn für alle Sicherheitsstufen $x > x'$ in L die Nichtbeeinflussungszusicherung $U[x,+\infty] :| U[-\infty,x']$ gilt, d.h. User einer höheren Sicherheitsstufe können solche einer niedrigeren Sicherheitsstufe nicht beeinflussen.

Als ein weiteres Beispiel für die Darstellung einer Sicherheitspolicy in der von Goguen und Meseguer präsentierten Notation soll hier der durch die folgende Skizze erlaubte Informationsfluß formalisiert werden:



Seien a,b,c,d User und A1,A2,A3 Operationenmengen. Information soll ausschließlich in der im Bild angegebenen Weise fliessen, also von a über A1 nach b und von b über A2 nach c oder über A3 nach d. Die entsprechenden Nichtbeeinflussungszusicherungen hierfür sind:

$\{b,c,d\} :| \{a\}$, $\{c,d\} :| \{b\}$, $\{c\} :| \{d\}$, $\{d\} :| \{c\}$ und
$\neg A1, \{a\} :| \{b,c,d\}$, $\neg A2, \{b\} :| \{c\}$, $\neg A3, \{b\} :| \{d\}$,
wobei hier $\neg Ai := C - Ai \quad (i=1,2,3)$.

Neben den bisher definierten statischen policies führen die Autoren noch dynamische policies, das sind policies, die vom Zustand der capability tables des Systems abhängen, ein. Solche dynamischen policies werden durch sogenannte konditionale Nichtbeeinflussungszusicherungen formuliert, d.h. Nichtbeeinflussungszusicherungen mit einer Bedingung, wobei

die Bedingung ein Prädikat über die Sequenz der Operationen ist, die angewandt wurden, um den momentanen Zustand zu erreichen. Es wird an dieser Stelle nicht näher auf dynamische policies eingegangen; die Definition und Beispiele dafür finden sich in <Gog82>.

Bemerkungen zum Modell

Goguen und Meseguer stellen mit ihrem Modell einen Ansatz vor, der sich nicht an speziellen Schutzproblemen orientiert, sondern versucht, möglichst allgemein die Problematik zu behandeln. Das präsentierte Modell ähnelt vom formalen Aufbau her etwas dem Ansatz von Feiertag (vgl. 4.2.2), beschränkt sich aber nicht auf die dort ausschließlich behandelte militärische Sicherheitspolicy. Die Autoren stellen vielmehr eine Methode zur Definition allgemeiner Sicherheitspolicies vor, mit der sowohl access control policies als auch information flow policies formal dargestellt werden können. Die strikte Trennung zwischen policy und mechanism (bzw. abstrakter Maschine im gegebenen Modell) ermöglicht zudem eine klare Definition von Sicherheit. Leider gehen die Verfasser auf die Verifikation von Sicherheit in ihrem Modell nicht näher ein, sie geben nur mögliche Ansätze hierfür an.

Die Entscheidungen zur Veränderung des Schutzzustands fallen durch die Werte der capability tables. Dieser Name ist deswegen etwas unglücklich gewählt, weil das vorgestellte Modell nicht von einer speziellen Darstellung des Schutzzustands mit Hilfe von capabilities abhängt. Beim Übergang eines Schutzzustands in einen anderen fällt die Entscheidung allerdings unabhängig vom Daten bzw. Objektzustand, was die Formulierung von content-dependent policies unmöglich macht. Überhaupt ist das Modell von Goguen und Meseguer nicht objektorientiert; Objekte werden nur in ihrer Gesamtheit als Datenzustand gesehen, so daß die Möglichkeit, bestimmten Objekten bestimmte Operationen zu ihrer Manipulation zuzuordnen - deren Ausführbarkeit dann

natürlich wertabhängig sein muß - nicht gegeben ist.

Nichtsdestoweniger ist das präsentierte Modell - verglichen mit den bisherigen Ansätzen - ein großer Schritt in Richtung einheitliche Behandlung von Schutz- und Sicherheitsproblemen.

An dieser Stelle muß noch ein dem hier vorgestellten Modell verwandter Ansatz von Rushby (vgl. <Rus81a>, <Rus81b>, <Rus81c>, <Rus82 >) erwähnt werden. Allerdings beschränkt sich Rushby auf eine "policy of isolation", die gewisse Usergruppen völlig isoliert. Aus diesem Grund wird hier auf die Arbeiten von Rushby nicht näher eingegangen, jedoch sei bemerkt, daß dort gerade der Beweis, daß die geforderte policy vom System erfüllt wird, sehr ausführlich und klar dargestellt wird, so daß vielleicht eine Anwendung jener Techniken auf das Modell von Goguen und Meseguer Impulse für die Verifikation von Sicherheit in diesem Modell geben könnte.

4.4.2 Das Modell von Stoughton

In seiner Arbeit <Sto81> präsentiert Stoughton einen Ansatz, der die beiden Teilgebiete Zugriffskontrolle und Informationsflußkontrolle in einem Modell zusammemfaßt. Chronologisch gesehen ist dies der erste Ansatz in dieser Richtung, da das Modell von Goguen und Meseguer erst später entstand. Der Autor begründet in seiner Arbeit zunächst die Notwendigkeit der beiden Schutzmechanismen zu Lösung realer Schutzprobleme und belegt anschließend anhand einiger Beispiele, daß die militärische Sicherheitspolicy für ein allgemeines Informationsflußkonzept nicht geeignet ist (vgl. auch 4.2.4). Er führt deshalb in seinem Modell ein Informationsflußkonzept ein, das auf dem "kontrollierten Teilen" von Information basiert. Bei der formalen Darstellung des Modells lehnt sich Stoughton sehr stark an die "denotational semantics"-Methode von Scott und Stratchey (vgl. <Ten76>)

zur Beschreibung von Systemen an. Dies führt leider zu einer ziemlich komplexen und deshalb auch etwas unübersichtlichen formalen Darstellung, so daß hier nur eine informale Beschreibung der wichtigsten Aspekte des Modells - die aber zum Verständnis seiner Wirkungsweise völlig ausreicht - vorgestellt wird.

Modellkomponenten

Aufgrund der gewählten Beschreibungsmethode definiert Stoughton sein Modell in zwei Schritten. Er legt zunächst die Syntax eines Systems und anschliessend dessen Semantik fest. Syntaktisch gesehen besteht ein System aus einer Menge von Kommandos (einfache Kommandos oder zusammengesetzte Kommandos, d.h. Programme), von denen jedes mit einem Usernamen (Subjekt, dem das Programm "gehört") versehen ist. Der syntaktische Aufbau der Kommandos wird durch eine rekursive Definition (einfache Operationsaufrufe, sequentielle Ausführung zweier Kommandos, Konditionalstatement, while-Schleife, Semaphore-Operationen) gegeben, die möglichen Identifikatoren (Subjektnamen, Objektnamen, Operationsnamen) durch einen entsprechenden (syntaktischen) Definitionsbereich.

Durch die Angabe der Semantik wird dann der bisher rein syntaktischen Systembeschreibung eine Bedeutung zugeordnet, die erst die Dynamik des Systems und damit auch die Menge aller möglichen Berechnungen festlegt. Im einzelnen werden durch die Angabe der Semantik die möglichen Systemzustände, d.h. die Struktur und Werte der Objekte, und die möglichen Zustandsübergänge, d.h. die Art und Interpretation der Operationen bzw. Kommandos definiert.

Grundlage eines Systems im Modell von Stoughton sind dabei dessen Objekte, deren Werte auch den Systemzustand definieren. Ein Objekt besteht aus vier Komponenten
Obj = (Data, Type, current accesses, potential accesses),
wobei "Data" die Daten des Objekts, "Type" dessen Typ und

die beiden anderen Komponenten die Schutzattribute des Objekts sind. Der Typ eines Objekts gibt an, welche Zugriffsarten prinzipiell für das Objekt definiert sind, d.h. mit welchen Operationen auf das Objekt zugegriffen werden kann, an welcher Stelle des entsprechenden Operationsaufrufs das Objekt stehen darf, und ob es bei diesem Aufruf nur gelesen, modifiziert, oder gelesen und modifiziert werden kann. Jedes Objekt besitzt zudem zwei Schutzattribute:

- die "current accesses" beschreiben, welche Operationen jedes Subjekt (User) auf das Objekt anwenden darf, d.h. sie definieren für jedes Subjekt eine Menge von erlaubten Zugriffen auf das Objekt; sie geben also an, in welchem Operationsaufruf des betreffenden Subjekts und an welcher Argumentenstelle des Aufrufs das Objekt vorkommen darf, und ob es gelesen oder modifiziert (oder beides) werden kann. Die current accesses sind somit die Zugriffskontrollkomponente des Objekts.

- die "potential accesses" geben stattdessen an, welche Operationen jedes Subjekt auf die im Objekt enthaltene Information anwenden darf, auch wenn die Information zu einem zukünftigen Zeitpunkt in einem anderen Objekt (auch eines andern Typs) enthalten sein wird. Die potential accesses sind also die Informationsfluß-Kontrollkomponente des Objekts.

Aufgrund der obigen Definition ist es unmittelbar klar, daß die current accesses immer eine Untermenge der potential accesses eines Objekts sind. Außerdem sind die current accesses eines Objekts immer eine Untermenge des Objekttyps - also der für dieses Objekt definierten Zugriffsarten - während die potential accesses dies nicht sein müssen, da sich die Information ja zu einem späteren Zeitpunkt in einem anderen Objekt mit einem anderen Typ und damit auch anderen Zugriffsarten befinden kann.

Neben den Objekten, die den Zustand des Systems festlegen, sind die Operationen eine weitere Grundkomponente eines Systems. Operationen führen Zustandsübergänge aus, sie modifizieren also Objekte. Im wesentlichen sind Operationen somit Funktionen von den gelesenen (und gelesenen und modifizierten) Objekten zu den modifizierten (und gelesenen und modifizierten) Objekten der Operation.

Zusammenspiel der Grundkomponenten

Wichtig für das Zusammenspiel der Modellkomponenten Objekte und Operationen ist nun, daß sich die Operationen "legal" in dem Sinn verhalten, daß die durch die current und potential accesses definierte policy eingehalten wird. Im einzelnen muß dabei für jede Operation gelten

- der potential access jedes modifizierten Objekts ist eine Untermenge des Durchschnitts aller potential accesses der gelesenen Objekte.

- der current access jedes modifizierten Objekts ist eine Untermenge des Typs dieses Objekts und dessen potential accesses.

Die erste Bedingung setzt hierbei die Informationsflußkontrolle des Systems, die zweite die Zugriffskontrolle durch. Um diese beiden Regeln für die Legalität einer Operation verständlicher zu machen, sollen sie allgemein begründet und anschliessend ihre Wirkungsweise anhand eines kleinen Beispiels demonstriert werden.

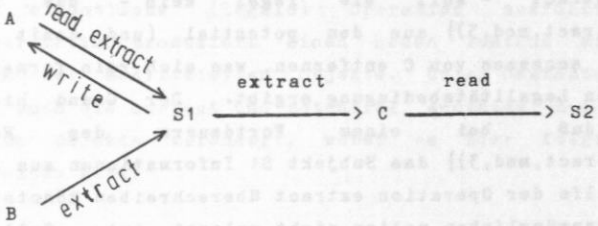
Man betrachte zuerst eine Operation, die aus den Werten von Objekten X1,X2,...,Xn den neuen Wert eines Objekts Y berechnet. Die neuen potential accesses von Y müssen dann eine Untermenge des Durchschnitts aller potential accesses der Xi sein; wären nämlich die neuen potential accesses von Y eine Obermenge des Durchschnitts aller potential accesses der Xi, so gäbe es mindestens ein Xj, auf dessen Information

- die ja nach der Operationsausführung auch mit in Y enthalten ist - nach der Ausführung der Operation mehr potentielle Zugriffe erlaubt wären, als vorher. Gerade dies soll aber durch die Einführung der potential accesses als Informationsfluß-Kontrollinstrument verhindert werden.

Die neuen current accesses des Objekts müssen natürlich laut Definition wieder eine Untermenge der neuen potential accesses von Y sein. Da aber die potential accesses nicht unbedingt typkonform sind, muß für die current accesses noch gefordert werden, daß sie zugleich eine Untermenge der durch den Objekttyp definierten Zugriffsarten sind.

Das folgende Beispiel soll nun den Sachverhalt weiter verdeutlichen. An dieser Stelle sollte erwähnt werden, daß Stoughton auf die Angabe eines Beispiels - wohl wegen der erforderlichen komplexen Darstellung - verzichtet. Hier wird ebenfalls aus diesem Grund das gegebene Beispiel nicht exakt in dem von Stoughton angegebenen Formalismus präsentiert, sondern eine verständlichere Darstellung gewählt.

Das Beispielsystem bestehe aus zwei Subjekten S1 und S2, drei Objekten A, B und C, und drei Operationen read, write und extract, wobei "read" die Daten eines Objekts liest, "write" neue Werte in ein Objekt schreibt, und "extract" die Werte zweier Objekte miteinander verknüpft und in ein drittes schreibt; die Operationentypen lassen sich dann als "read(ref)", "write(mod)" und "extract(ref,ref,mod)" schreiben. Alle drei Objekte seien vom gleichen Typ
{(read,ref,1), (write,mod,1), (extract,ref,1), (extract,ref,2), (extract,mod,3)}, d.h. die Operationen read, write und extract (sowohl lesend als auch schreibend) seien auf A, B und C anwendbar. Das folgende Diagramm stelle nun die erlaubten Datenflüsse und Operationsaufrufe dar:



S1 soll also aus A lesen und in A schreiben können, und die mit extract verknüpften Werte aus A und B nach C schreiben dürfen. S2 soll die Werte des Objekts C lesen können. Die für die Objekte A, B und C erforderlichen current und potential accesses lassen sich am besten anhand einer Tabelle darstellen:

	current accesses	potential accesses
A	S1->{(read,ref,1), (write,mod,1), (extract,ref,1)}	S1->{(read,ref,1), (write,mod,1), (extract,ref,1)} S2->{read,ref,1)}
B	S1->{(extract,ref,2)}	S1->{(extract,ref,2)} S2->{(read,ref,1)}
C	S1->{(extract,mod,3)} S2->{(read,ref,1)}	S1->{(extract,mod,3)} S2->{(read,ref,1)}

Die Zugriffskontrollkomponente current accesses definiert hier, daß S1 auf A mit read und write zugreifen darf, und die Operation extract(A,B,C) aufrufen darf; S2 darf mit read auf C zugreifen. Zusätzlich darf S2 Informationen, die aus A oder B abgeleitet wurden, lesen. Wäre z.B. das Recht S2->{(read,ref,1)} nicht unter den potential accesses von A oder B, so wäre eine Operation extract nur dann legal, wenn

sie aus C das Recht S2->{(read,ref,1)} entfernen würde.

Selbst bei der angegebenen Konfiguration muß die Operation extract - soll sie legal sein - das Recht S1->{(extract,mod,3)} aus den potential (und damit auch current) accesses von C entfernen, was sich rein formal aus der ersten Legalitätsbedingung ergibt. Der Grund hierfür ist, daß bei einem Fortdauern des Rechts S1->{(extract,mod,3)} das Subjekt S1 Informationen aus A und B mit Hilfe der Operation extract überschreiben könnte, was in der ursprünglichen policy nicht erlaubt ist. Soll das Recht S1->{(extract,mod,3)} jedoch in C verbleiben, weil z.B der Extraktionsvorgang wiederholt werden können soll, so muß in den potential accesses (aber nicht unbedingt in den current accesses) von A und B das Recht S1->{(extract,mod,3)} ebenfalls enthalten sein.

Der Autor gibt nun für die verschiedenen syntaktischen Kommandotypen

- einfacher Operationsaufruf "op(...)"
- sequentielle Ausführung zweier Kommandos "C1;C2"
- Konditionalstatement "If bool Then C1 Else C2 Fi"
- Schleife "While bool Do C Od"
- Semaphore-Operationen "Psem sema" und "Vsem sema"

semantische Interpretationsfunktionen an, die die jeweiligen Zustandsübergänge des Modells festlegen. Wie bei Denning oder Andrews und Reitman (vgl. 4.3) definiert Stoughton dabei sogenannte lokale und globale "implicit accesses", die den lokalen und globalen impliziten Informationsfluß kontrollieren; neben den Objektwerten werden auch diese implicit accesses durch die Interpretationsfunktionen beeinflußt.

Die Interpretationsfunktionen führen zunächst einen access-control check durch (der entsprechende Operationsaufruf darf Objekte nur an solchen Stellen enthalten, die in den current accesses dieser Objekte aufgeführt sind!); an-

schliessend wird eine "apply"-Funktion angewandt, die dann die eigentliche (legale) Operation ausführt. Die apply-Funktion produziert einen neuen Zustand mit neuen Werten der modifizierten Objekte. Unter Umständen werden dabei auch die current und potential accesses der modifizierten Objekte verändert, wobei es hier folgende Möglichkeiten gibt:

- Erweiterung der current accesses in Objekten
- Verminderung der current accesses in Objekten
- Verminderung der potential accesses in Objekten

Bei Konditionalstatements werden zusätzlich noch (vgl. 4.3.4) die lokalen impliziten accesses modifiziert. Man beachte, daß in diesem dynamischen Laufzeitmodell im Gegensatz zu statischen Modellen der Informationsfluß von der Bedingung eines Konditionalstatements zum Then- und Else-Teil nicht betrachtet werden kann, sondern nur der Informationsfluß in dem Zweig, der aktuell eingeschlagen wird. While-Schleifen werden ähnlich wie Konditionalstatements behandelt, d.h. die lokalen impliziten accesses werden modifiziert. Da es sich um einen Laufzeitmechanismus handelt, kann auch hier der globale implizite Informationsfluß von einer nichtterminierenden Schleife zum Rest des Systems nicht betrachtet werden. Die einzigen Kommandos, die also die globalen impliziten accesses manipulieren, sind die Semaphore-Kommandos, da sie Informationsfluß zwischen verschiedenen Prozessen über die Semaphore herstellen.

An dieser Stelle sei noch bemerkt, daß das von Stoughton präsentierte Modell keine Erweiterungen der einmal definierten potential accesses ermöglicht; der Autor stellt aber abschliessend kurz eine Erweiterungsmöglichkeit seines Ansatzes vor, die dies gestatten würde. Hierauf wird allerdings im Rahmen dieser Arbeit nicht näher eingegangen.

Bemerkungen zum Modell

Wie Goguen und Meseguer stellt Stoughton mit seinem Modell einen Ansatz vor, der versucht, die Problematik Schutz und Sicherheit allgemein zu betrachten. Im Gegensatz zu jenem Modell orientiert sich Stoughton jedoch nicht an irgendeinem bestehenden Schutzmodell, sondern präsentiert einen völlig neuen Ansatz. Leider führt die gewählte Beschreibungsmethode zu einer recht komplexen Darstellung, die dem an sich klar aufgebauten Modell viel an Übersichtlichkeit nimmt.

Der Autor stellt mit seiner Arbeit ein Modell vor, das gleichermaßen Zugriffsschutz und Informationsflußkontrolle ermöglicht; durch die Angabe der current accesses und potential accesses werden die Sicherheitspolicies für beide Teilgebiete definiert. Der angegebene Laufzeit-Schutzmechanismus ist so konstruiert, daß er automatisch die gegebenen policies durchsetzt, d.h. die Sicherheit eines Systems - die Durchsetzung der policies durch den Mechanismus - ist immer gewährleistet, und eine Verifikation ist nicht für jedes nach diesem Modell aufgebaute System gesondert durchzuführen. Das Modell ist allerdings ein rein formales Modell, d.h. es wird nicht auf Implementierungsfragen oder die Verifikation der Korrektheit der Spezifikation bzw. der Implementierung einer Spezifikation (die Frage, welche Objekte durch eine Operation wirklich gelesen und modifiziert werden) eingegangen.

Stoughton behandelt in seinem Ansatz auch den globalen Informationsfluß über Semaphore. Wie im Abschnitt 3.5.1 dargestellt, führt jedoch eine Synchronisation über Semaphore zu "unstrukturierter Programmierung" und in der Praxis oft zur Blockade eines Systems (vgl. 3.4.1), und ist deshalb in einem modernen Objektverwaltungskonzept für die Synchronisation der Zugriffe zu Objekten nicht geeignet.

Das Modell von Stoughton ist ein objektorientiertes Modell und die Schutzentscheidungen fallen aufgrund von Daten, die im Objekt selbst enthalten sind, was dem Lokalitätsprinzip entspricht. Allerdings sind die Entscheidungen nur von den current accesses und potential accesses des Objekts abhängig, eine Abhängigkeit vom Wert des Objekts ist vom Autor nicht vorgesehen. Das Modell ist auch nicht datentyporientiert in dem Sinn, daß Objekttypen und die entsprechenden Operationen zu ihrer Manipulation zu einem abstrakten Datentyp zusammengeschlossen sind; durch den Objekttyp wird allerdings definiert, in welchen Operationen und an welcher Stelle des Operationsaufrufs das entsprechende Objekt vorkommen darf.

Insgesamt ist das Modell von Stoughton jedoch ein wertvoller Beitrag in Richtung einheitliche Behandlung von Schutzproblemen, da in ihm erstmals die beiden Problemkreise Zugriffsschutz und Informationsflußkontrolle in einem gemeinsamen Modell präsentiert werden. Der Ansatz von Stoughton geht auch noch über den von Goguen und Meseguer behandelten Umfang hinaus, da von Stoughton nicht nur die Definition von policies behandelt wird, sondern eine allgemeine abstrakte Maschine zur Behandlung von Schutzproblemen in Betriebssystemen angegeben wird.

Im nächsten Abschnitt soll nun eine kurze Gegenüberstellung der bisher existierenden Ansätze und der aufgestellten Anforderungen an ein allgemeines Objektverwaltungsmodell (vgl. 3.4.3) präsentiert werden, um dann zum in dieser Arbeit gewählten Ansatz überzuleiten.

4.5 Bemerkungen zu formalen Schutzmodellen

In diesem Abschnitt sollen zunächst die Eigenschaften der verschiedenen existierenden Modelle kurz zusammengefaßt und gegenübergestellt werden; diese Gegenüberstellung ist auch in tabellarischer Form (siehe S 181) zusammengestellt.

Bekanntlich lassen sich die Modelle aufgrund des gewählten Ansatzes für Schutz und Sicherheit in die drei Klassen Objektschutzmodelle, Modelle für militärische Sicherheit und Informationsflußmodelle einteilen, wobei die Modelle von Goguen und Meseguer und Stoughton allgemeine Modelle sind, die die beiden Gebiete Objektschutz und Informationsflußkontrolle gleichzeitig abdecken.

Eine andere Möglichkeit der Typisierung wäre danach, ob das Modell objektorientiert ist - also auf Systemstrukturen wie Objekten, Subjekten, Operationen etc. basiert - oder sprachorientiert, d.h Programme einer Programmiersprache betrachtet, wobei sprachorientierte Ansätze ausschließlich bei Informationsflußmodellen zu finden sind. Die objektorientierten Modelle lassen sich wiederum in zwei Unterklassen einteilen, in solche, die vor allem die Weitergabe von Rechten im System betrachten, wie die Zugriffsmatrix - und Take-Grant-Modelle, und in die Klasse derer, die die eigentlichen Zugriffe der Subjekte auf Objekte untersuchen, wie die Modelle von Popek, Feiertag, Jones und Lipton oder Goguen und Meseguer. Zwei der Modelle - die Hierarchischen Take-Grant-Modelle und das Modell von Stoughton - berücksichtigen beide Aspekte.

So unterschiedlich die einzelnen Ansätze der Modelle sind, so verschieden sind auch ihre Eigenschaften. So führen nur wenige Modelle eine Trennung zwischen policy und Mechanismus im angegebenen Sinn durch, d.h die policy spezifiziert das gewünschte Verhalten des Systems und ein angegebener Mechanismus setzt diese policy durch. Die Sicherheit des Systems ist dann das Bindeglied zwischen policy und Mechanismus, d.h. ein System ist sicher, wenn der ange-

gebene Mechanismus die definierte policy durchsetzt. Dieses Sicherheitskonzept ist also als relatives Konzept zu sehen - Sicherheit wird nämlich immer relativ zu einer frei angebbaren policy betrachtet.

In den meisten Modellen ist der policy-Begriff aber viel eingeschränkter; hier versteht man unter policy ein fest für das Modell vorgegebenes Verhaltensmuster. Sicherheit bezieht sich in diesen Fällen auf die feste policy, wird also zum absoluten, starren Konzept. In einigen Modellen wird auf die Einführung von Sicherheitspolicies sogar ganz verzichtet; hieraus resultiert allerdings dann auch ein sehr eingeschränkter Sicherheitsbegriff, bzw. der Begriff Sicherheit wird überhaupt nicht definiert. Beispiele hierfür sind die Modelle von Cohen und Landauer und Crocker, die sich auf die reine Betrachtung des Informationsflusses in Programmstrukturen beschränken.

Selbst bei Modellen mit einer sauberen Trennung zwischen policy und Mechanismus findet man kaum die Möglichkeit, wertabhängige policies oder sogar völlig beliebige policies - z.B Zugriffsschutz- und Informationsfluß-policies - zu definieren. Eine Ausnahme machen hier nur die beiden Modelle von Goguen und Meseguer und von Stoughton, die ja Objektschutz und Informationsflußkontrolle - und auch beliebige policies - betrachten. Auch das Modell von Jones und Lipton gestattet eine freie Definition von policies, beschränkt sich aber auf Informationsflußpolicies, und deckt daher nicht das komplette Spektrum ab.

Eine weitere Einschränkung der Allgemeinheit besteht bei fast der Hälfte der Modelle darin, daß sie nur eine feste Menge von Zugriffstypen (z.B. read, write, take, grant etc.) betrachten; solche Modelle sind für die Betrachtung von realen Schutzproblemen von vorneherein ungeeignet.

Ein sehr wichtiger Punkt beim Themenkreis Schutz und Sicherheit ist die Verifikation der Sicherheit, d.h. es genügt für ein System nicht einfach sicher zu sein, sondern dies muß auch beweisbar sein. Auch diesen grundlegenden Punkt behandeln eine Reihe von Modellen nur in sehr unzureichendem Maße. Die Verifikation von Sicherheit erfordert zwei Schritte: den Nachweis, daß der Mechanismus, bzw. eine High-Level Spezifikation des Mechanismus, die angegebene policy erfüllt (design verification), und den Beweis, daß der Implementierungscode die High-Level Spezifikation korrekt implementiert (program verification) (vgl. <Mil81>).

Bis auf die Modelle von Popek und Feiertag wird in allen Modellen, die Verifikationsaspekte betrachten, nur der erste Schritt berücksichtigt. Dies ist vor allem dadurch bedingt, daß in den Modellen Abstraktionsmöglichkeiten, d.h. die Einteilung eines Systems in verschiedene Abstraktionsebenen - von der Spezifikation bis hin zur Implementierung - nicht berücksichtigt sind. Man kann an dieser Stelle einwenden, daß dies in rein formalen Modellen nicht nötig sei, dem sei jedoch hier entgegengehalten, daß für einen Einsatz eines Modells in der Praxis - und dies sollten ja alle Modelle als Ziel haben - solche Abstraktionsmöglichkeiten unumgänglich sind.

Ein weiterer wichtiger Gesichtspunkt für den praktischen Einsatz ist die Erfüllung der Lokalitätseigenschaft durch ein Modell, d.h. die Entscheidung über Zugriffe auf ein Objekt sollte in räumlicher Nähe des Objekts, nicht durch einen globalen Mechanismus, getroffen werden. Es gibt allerdings nur zwei Modelle, die Ansätze von Minsky und Stoughton, die sich an dieses sehr wichtige Prinzip halten.

Die vorstehenden Ausführungen sollten nochmals eines kurzen Überblick über die momentan existierenden formalen Schutzmodelle und ihre Eigenschaften geben. Im folgenden werden nun die an ein allgemeines Objektverwaltungsmodell zu stellenden Forderungen nochmals kurz zusammengefaßt (vgl.

3.4.3) und daraus die Anforderungen an den hier zu erstellenden Ansatz abgeleitet.

Im Abschnitt 3.4.3 wurde gefordert, daß einem Objektverwaltungsmodell ein einheitliches Konzept für allgemeine Objektverwaltungsaufgaben, Synchronisation und Schutz zugrundeliegen muß. Da die vorgestellten Modelle reine Schutzmodelle sind, decken sie die Gebiete allgemeine Objektverwaltung und Synchronisation natürlich nicht mit ab. Es gibt auch noch kein allgemeines Modell, das diese drei Anforderungen gleichzeitig erfüllen würde. Aber auch auf dem Gebiet Schutz ergeben sich aus der Forderung nach Allgemeinheit des Modells einige notwendige Eigenschaften, die von keinem der bisherigen Schutzmodelle gleichzeitig erfüllt werden:

- Behandlung von Zugriffsschutz und Informationsfluß
- Möglichkeit der Behandlung der Manipulation von "Rechten" und der eigentlichen Zugriffe auf Objekte für beliebige Zugriffstypen
- Strikte Trennung zwischen Sicherheitspolicy und Schutzmechanismus
- Möglichkeit der Formulierung beliebiger - auch wertabhängiger - policies
- Klare Definition des Begriffs "Sicherheit" mit Angabe von Verifikationsmöglichkeiten
- Spezifikations- und Abstraktionsmöglichkeiten
- Einhaltung des Lokalitätsprinzips

Nicht nur für den Bereich Schutz, sondern auch für die beiden anderen Gebiete der der Objektverwaltung müssen in einem allgemeinen Objektverwaltungsmodell - gerade im Hinblick auf dessen Einsatzmöglichkeiten in der Praxis - Abstraktionsmöglichkeiten bestehen. Die Trennung zwischen Spezifikation und Implementierung ist bekanntlich eine entscheidende Voraussetzung für die Verständlichkeit, Modifizierbarkeit und auch Verifizierbarkeit eines Systems. Ein Modell, das diese Möglichkeiten nicht bietet, erweist sich deshalb von vorneherein als nicht geeignet für praktische

Anwendungsfälle.

Aufgabe dieser Arbeit ist es daher, einen ersten Schritt in Richtung einheitliche Behandlung von Objektverwaltungsproblemen durch die Angabe eines allgemeinen formalen Modells für Objektverwaltung zu tun. In den Kapiteln 5 und 6 dieser Arbeit wird dieser Ansatz ausführlich präsentiert und dann im Kapitel 7 an einem Praxisbeispiel seine Anwendungsmöglichkeiten unter Beweis gestellt.

	ZM	UCLA	TG	Minsky	BLP	Feiert	Integr	HTG	Denn	A&R	J&L	Cohen	F&M	L&C	G&M	Stough
Objektschutz	x	x	x	x				x							x	x
milit.Sicherheit					x	x	x	x					x			
Inf.fluß-syntakt.									x	x	x					
Inf.fluß-semant.						(x)					x	x	x	x	x	x
Obj.-Recht-orient.	x	(x)	x	x	x		x	x								x
Obj.-Zugriff-orien		x				x		x			x				x	x
Sprachstrukt.orien									x	x		x	x	x		
policy-Mechanismus		x		x	(x)	x	(x)	(x)	(x)	(x)	x		(x)		x	x
beliebige policies											(x)				x	x
wertabhäng. pol.				x							x					
belieb. Zugr.typen		x				x			x	x	x	x	x	x	x	x
Sicherheitsdefin.	(x)	x	(x)		x	x	x	x	x	x	x		x		x	x[1]
Verifik. v. Sich.		x				x			x	x	x	x[2]	x	x[2]		x[3]
Spezifik. Abstr.		x				x										
Lokalität				x												x
Praxisanwendungen	x	x				x	x									

1 Sicherheit ist durch die Definition des Mechanismus gewährleistet

2 Verifikation des Informationsflusses, nicht von Sicherheit

3 Verifikation der Sicherheit des Mechanismus nicht durchgeführt (aber vgl. 1)

5. Ein formales Modell für die Objektverwaltung zur Definition der Semantik der zu entwickelnden Spezifikations- und Implementierungsmethode

Nachdem in den vorhergehenden Kapiteln die Anforderungen an ein Objektverwaltungsmodell, und insbesondere die Notwendigkeit der Einbeziehung von Schutz- und Sicherheitsaspekten, erläutert wurden, wird nun im folgenden ein formales Modell entwickelt, das diesen Anforderungen genügt. Ansätze für dieses Modell finden sich bereits in der Betriebsmittelmaschine (BMM) <Ker77>, als Basis diente die in <Ker82> vorgestellte Betriebssystemmaschine (BSM), die in dieser Arbeit um Schutzkonstrukte erweitert wird.

In einem nächsten Schritt entsteht dann durch eine Erweiterung des Konzepts der Abstrakten Datentypen und die Integration dieses erweiterten Ansatzes in die BSM die abstrakte Objektverwaltungsmaschine, die dann die der Methode zur Spezifikation und Implementierung zuverlässiger SW-Systeme zugrundeliegende Semantik definiert.

Nach einer kurzen Einführung wird im zweiten Abschnitt des Kapitels die Entwicklung des formalen Modells aus der BMM bzw. BSM dargestellt. Eine Skizze des Modells soll dabei bereits einen ersten Einblick in die eingeschlagene Vorgehensweise geben.

Im Abschnitt 5.3 wird dann das Modell der um Schutzkonstrukte erweiterten Betriebssystemmaschine durch die Angabe von Syntax und Semantik formal definiert. Hierbei wird detailliert auf verschiedene Möglichkeiten für die Definition der Abfrage und Veränderung einer Variablen durch eine Operation - die Basis für die Betrachtung von Schutz- und Synchronisationsfragen - eingegangen, und schließlich ein neuer Ansatz präsentiert, der auch für relationale Operationen Gültigkeit besitzt. Aufbauend auf diesem Ansatz werden anschliessend Fragen der Synchronisation und des Schutzes auf der Ebene der Betriebssystemmaschine diskutiert.

Der Wunsch nach einer Einführung von Daten- und prozeduraler Abstraktion in die BSM führt dann zu einer Integration des Konzepts der Abstrakten Datentypen in das bisherige Modell, und schließlich zur Definition der abstrakten Objektverwaltungsmaschine als formales Modell für die Verwaltung der Zugriffe auf abstrakte Objekte. Nach der Angabe der Syntax und Semantik der OV-Maschine wird eine Möglichkeit zur Einbettung von OV-Maschinen in eine übergeordnete Supervisor-BSM als formales Modell für ein Gesamt-SW-System diskutiert. Anschließend werden für die OV-Maschine Möglichkeiten zur Definition von Schutzstrategien (policies) präsentiert, und der für die Zuverlässigkeit eines Systems grundlegende Begriff der Sicherheit eines ADT`s präzisiert.

Den Abschluß dieses Kapitels bildet dann im fünften Abschnitt eine Übersicht über die Leistungen und Möglichkeiten des präsentierten formalen Modells.

5.1 Einführung und Problemstellung

Die Ausführungen in den vorhergehenden Kapiteln haben die Notwendigkeit der Forderung nach einem allgemeinen, formalen Modell für Objektverwaltung - zur Konsistenzerhaltung der Objekte, Schutz vor unerlaubtem Zugriff und Synchronisation der Zugriffe auf Objekte - unterstrichen. Bei der Erstellung eines solchen Modells sind darüberhinaus noch wichtige Nebenbedingungen einzuhalten, die im folgenden kurz aufgeführt werden sollen.

Die wohl wesentlichste Forderung an das Modell betrifft die Einhaltung bzw. Unterstützung der im Abschnitt 3.2 aufgezeigten SW-Entwurfsprinzipien. Hier wären an erster Stelle das Prinzip der Abstraktion mit den Möglichkeiten der Datenabstraktion und prozeduralen Abstraktion zu nennen, desweiteren die Erfüllung des Lokalitätsprinzips und damit die Unterstützung der Modifizierbarkeit. Das OV-Modell

sollte die Modularisierung und Strukturierung eines auf ihm aufbauenden Gesamtsystems unterstützen, und anwendbar auf hierarchische Systeme sein.

Von großer Bedeutung für die Anwendbarkeit ist die Forderung, daß das formale Modell direkt als Basis für die Spezifikation und Implementierung zuverlässiger SW-Systeme einsetzbar ist, so daß ein einheitlicher Ansatz "formales Modell -- Spezifikation -- Implementierung" entsteht. Das Modell muß weiterhin die Formulierung beliebiger OV-Strategien gestatten und die Durchsetzung der Strategien garantieren. Hierfür muß im Modell eine klare Trennung zwischen policy und Mechanismus erfolgen; dies ist auch die Voraussetzung für einen klaren Sicherheitsbegriff, der wiederum die Verifizierbarkeit im Modell erhöht.

Eine weitere Forderung an ein formales Modell zur Objektverwaltung ist seine Problemadäquatheit; d.h. das Modell darf nicht zu speziell sein, und somit nur für eine eingeschränkte Klasse von Anwendungsfällen einsetzbar sein (z.B. nur für eine spezielle Sicherheitspolicy wie militärische Sicherheit). Andererseits darf es nicht zu allgemein sein (z.B. endlicher Automat, Turing-Maschine, vgl. 5.2.1), d.h. es muß gestatten, anwendungsbezogene Fragen wie Verträglichkeit oder Informationsfluß schon im formalen Modell zu formulieren bzw. zu klären.

Natürlich muß sich das Modell auch an den bisherigen Ansätzen zur Formulierung und Lösung von Teilgebieten der Objektverwaltung orientieren. Für das Gebiet Schutz wurden deshalb im vorhergehenden Kapitel formale Schutzmodelle intensiv betrachtet, und untersucht, inwieweit sie die hier aufgestellten Forderungen an ein OV-Modell erfüllen. Für die Synchronisation wurde auf die umfangreichen Betrachtungen in <Ker82> zurückgegriffen, was schließlich auch zur Entscheidung führte, die dort entwickelte Betriebssystemmaschine als Basiskonzept für das zu entwickelnde Modell einzusetzen.

Ein wesentlicher Gesichtspunkt für die Einhaltung der dieser Arbeit zugrundeliegenden Zielsetzung ist die Anwendbarkeit des Ansatzes bei der Erstellung von Software für den Einsatz im Produktionsbereich - mit den in Kapitel 2 dieser Arbeit aufgezeigten Besonderheiten. Dies gilt insbesondere für das Schema zur Spezifikation und Implementierung; da dessen Semantik durch das formale Modell jedoch exakt vorgegeben ist, ist schon bei dessen Erstellung eine Eignung für den o.g. Einsatz zu berücksichtigen.

Eng damit zusammenhängend, und großen Einfluß auf die Anwendbarkeit des Ansatzes in der Praxis ausübend, ist die Forderung nach dessen leichter Verständlichkeit, wobei auch hier wieder der Schwerpunkt bei dem für den Benutzer ausschlaggebenden Schema für die Spezifikation und Implementierung liegt. Aus diesem Grund wird der gesamte Ansatz in einem mehr theoretischen Teil (formales Modell - Kapitel 5) und einem anwendungsorientierten Teil (Spezifikations- und Implementierungskonzept - Kapitel 6/7) mit Anwendungsbeispielen aus der Praxis beschrieben.

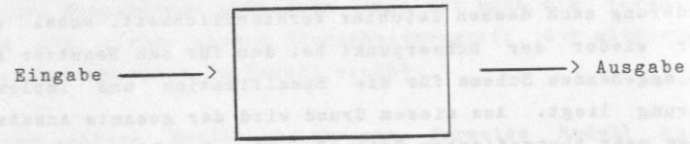
5.2 Einordnung des Modells

5.2.1 Formale Modelle in der Informatik

Formale bzw. mathematische Modelle dienen der abstrakten Darstellung von realen Systemen und Sachverhalten in diesen Modellen. Ausschlaggebend für die Qualität, d.h. die Problemadäquatheit, eines formalen Modells ist dabei die Konzentration auf das Wesentliche, die es gestattet, Systemeigenschaften ohne Festlegung auf bestimmte Interpretationen zu untersuchen. Ziel der Erstellung eines formalen Modells ist dann die Ausnutzung der im formalen Modell gewonnenen Erkenntnisse in der Praxis.

Bei der Erstellung eines formalen Modells taucht automatisch die Frage nach seiner Universalität auf. Dies sei an einigen Beispielen aus der Informatik erläutert:

Das einfachste formale Modell für dynamische Systeme, d.h. Systeme, die auf Umwelteinflüsse reagieren können, ist der endliche Automat. Man kann sich einen endlichen Automaten als "Black Box" vorstellen, in die man "etwas eingeben kann", worauf sie aufgrund der Eingabe und eines internen Systemzustands "auf die Eingabe antwortet".



Formal läßt sich ein endlicher Automat als Tupel (I,O,S,do,out,s0) definieren, wobei gilt

I endliches Eingabealphabet
O endliches Ausgabealphabet
S endliche Zustandsmenge
s0 Anfangszustand (s0 ∈ S)
do Zustandsübergangsfunktion do: I x S --> S
out Ausgabefunktion out: I x S --> O

Bei Eingabe eines Elements i ∈ I im Zustand s ∈ S geht der Automat in den Folgezustand do(i,s) ∈ S über und gibt out(i,s) ∈ O aus.

Es ist aufgrund der Definition unmittelbar einsichtig, daß es sich hierbei um ein sehr allgemeines Modell handelt, das (fast) auf jedes dynamische System anwendbar ist. Leider besitzt der endliche Automat nur eingeschränkte Berechnungsmöglichkeiten, und ist somit kein universelles

Modell.

Das einfachste universelle Modell (in dem Sinn, daß auf dem Modell jede berechenbare Funktion berechnet werden kann) ist die Turing-Maschine (vgl. <Her78>). Informal funktioniert dieses Modell so, daß in die Turing-Maschine (Modell für Rechner) ein Turing-Programm (Rechenvorschrift) eingegeben wird, und so (durch Eingabe des entsprechenden Programms) von der Turing-Maschine jede beliebige berechenbare Funktion berechnet wird.

Die Turing-Maschine ist somit zwar ein universelles Modell, aber leider für die Modellierung von Rechnersystemen nicht problemadäquat, da selbst die Darstellung von einfachen Sachverhalten so komplex wird, daß der Blick auf das Wesentliche verschleiert wird.

Man muß also - gerade zur Modellierung anwendungsorientierter Systeme - auf spezielle, für die jeweilige Anwendung problemadäquate, Modelle zurückgreifen. Als Beispiele wären hier Petri-Netze zur Modellierung paralleler Abläufe oder parallele Programmschemata für die Parallelisierung von Programmen zu nennen.

Als formales Modell zur Darstellung der Abläufe in Betriebssystemen bzw. allgemeinen asynchronen Systemen wurde von Keramidis die Betriebsmittelmaschine (BMM) <Ker77> entwickelt, die dann in <Ker82> zur Betriebssystemmaschine (BSM) erweitert wurde.

Beide Modelle sind im o.g. Sinn universell, jedoch ist das Modell der BSM wesentlich breiter einsetzbar und zudem für asynchrone Systeme problemadäquater als die BMM.

5.2.2 Formale Modelle für asynchrone Systeme

Unter einem asynchronen System (bzw. asynchronem Prozeßsystem) versteht man ein System mit einer endlichen Anzahl von Prozessen (aktive Einheiten), die mit Hilfe von Operationen private oder gemeinsame Größen (Objekte, Variablen) manipulieren können. Bezüglich der relativen Geschwindigkeiten, mit denen die einzelnen Prozesse ihre Operationen ausführen, werden keine Einschränkungen gemacht (Asynchronität).

Im folgenden werden nun informal die beiden Ansätze von Keramidis zur Modellierung asynchroner Systeme, die BMM und die BSM, beschrieben:

In der Betriebsmittelmaschine wird der gemeinsame Zugriff von Prozessen auf Betriebsmittel (Ressourcen) modelliert - mit dem Schwerpunkt auf der Synchronisation der Prozesse. Das formale Modell besteht aus einer Menge von Prozessen, einer Menge von Betriebsmitteln und sogenannten Elementaroperationen (die P- und V-Operationen auf Semaphore-Feldern entsprechen), mit denen die Prozesse auf die Betriebsmittel zugreifen, und so das Belegen bzw. Freigeben der Betriebsmittel modellieren. Zwischen den Prozessen existiert eine Prioritätsrelation, die die Formulierung von Scheduling-Strategien ermöglicht. Die BMM kennt vier mögliche Prozeßzustände - "laufend", "laufbereit", "wartend" und "blockiert" - die den Prozessen je nach momentan beabsichtigtem Zugriff zugeordnet werden.

Die BMM ist eine universelle Basis für Betriebssysteme, da sie rekursiv äquivalent zur Turing-Maschine ist, d.h. jede Turing-Maschine kann durch eine BMM simuliert werden und umgekehrt. Allerdings kann die BMM nicht als allgemeines problemadäquates Modell für asynchrone Systeme gelten, da die Verwendung von Elementaroperationen eine zu große Einschränkung bedeutet.

Während also die BMM vor allem den Zugriff von Prozessen auf Betriebsmittel in einem asynchronen System modelliert, ist die BSM ein allgemeines Berechnungsmodell für asynchrone Systeme. Hier soll nur ein kurzer Überblick über das Modell gegeben werden, da in Abschnitt 5.3 bei der Präsentation einer Erweiterung der BSM eine detaillierte Beschreibung erfolgt.

Die BSM besteht aus mehreren sequentiellen Prozessen, die über gemeinsame Variablen kommunizieren. Diese Vorstellung erfordert zum einen ein Ersetzen der Betriebsmittelmenge der BMM durch einen allgemeinen globalen Datenbereich, zum anderen die Ersetzung der Elementaroperationen durch beliebige relationale Operationen auf diesen Datenbereich. Neben der Prioritätsrelation für Prozesse existiert eine sogenannte Verträglichkeitsrelation, die explizit angibt, wann zwei Operationen verträglich sind, d.h. wann sie gleichzeitig ausgeführt werden können, ohne daß inkonsistente Datenzustände auftreten.

Das dynamische Verhalten der BSM ist durch zwei mögliche Schritte definiert, das Starten und Beenden einer Operation eines Prozesses. Wie bei der BMM werden die vier möglichen Prozeßzustände "laufend", "laufbereit", "wartend" und "blockiert" unterschieden.

Die BSM ist ein universelles, problemadäquates Berechnungsmodell für asynchrone Systeme; sie wird in <Ker82> als Basis für die Spezifikation und Implementierung asynchroner Systeme herangezogen. Jedoch fehlen in der BSM Ansätze für Datenschutz und -sicherheit sowie eine Unterstützung von Daten- und prozeduraler Abstraktion: Der Datenbereich ist ein unstrukturierter n-dimensionaler Zustandsraum und die Operationen sind Übergänge in diesem Zustandsraum.

In dieser Arbeit werden nun die fehlenden Aspekte in die BSM integriert. Daraus entsteht schließlich das Modell der abstrakten Objektverwaltungsmaschine.

5.2.3 Eine Erweiterung der BSM und deren Interpretation

Aufbauend auf der BSM wird in diesem Kapitel ein formales Modell präsentiert, das neben Schutzaspekten auch eine Unterstützung von prozeduraler und Datenabstraktion beinhaltet. Die Entwicklung dieses Modells geschieht dabei in zwei Schritten:

Zuerst wird die BSM um Schutzkonstrukte erweitert; es handelt sich bei dieser Erweiterung immer noch um ein allgemeines, uninterpretiertes Modell, das sich noch sehr stark an der ursprünglichen BSM orientiert, d.h. der Datenbereich ist unstrukturiert und die Operationen sind keine Prozeduren, sondern Übergänge im Zustandsraum des Datenbereichs.

Die eingeführten Datenschutzmechanismen müssen, wegen der Definition der Operationen als abstrakte Zustandsübergänge, in den Operationen verborgen werden. Dies wird so modelliert, daß jede Operation neben der normalen Reaktion (Veränderungen von Variablen im gemeinsamen Datenbereich) eine Reaktion im Fehlerfall beinhaltet, bei der ein für jeden Prozeß privater Fehlerbereich verändert wird.

Außer diesen Datenschutzmechanismen wurde gegenüber der BSM bewußt nichts verändert, da gezeigt werden sollte, daß auch auf der niedrigen Ebene eines völlig allgemeinen, uninterpretierten Modells Schutzprobleme betrachtet werden können.

Für das Modell der erweiterten BSM werden dann verschiedene mögliche Ansätze zur Definition der Abfrage und Veränderung diskutiert und ein neuer Ansatz präsentiert. Dieser Unterschied zur BSM wird notwendig, da der dortige Ansatz nicht in alle Fällen eine exakte Definition für semantische Abfrage bietet. Aufbauend auf Abfrage und Veränderung werden dann schließlich Schutzprobleme in der erweiterten BSM untersucht und ein Informationsflußprädikat zwischen Prozessen definiert.

Im zweiten Schritt wird dann das vorliegende Modell so interpretiert, daß Daten- und prozedurale Abstraktion bereits vom formalen Modell unterstützt werden. Dies ist eine wichtige Voraussetzung für die Uniformität von formalem Modell, Spezifikations- und Implementierungskonzept.

Der Begriff des Abstrakten Datentyps (vgl. Abschnitt 3.2) wird formal in das Modell eingeführt - als Strukturierungskonzept und als Muster für die Repräsentation abstrakter Objekte. Das ADT-Konzept unterstützt dabei sowohl Daten- als auch prozedurale Abstraktion: Der Datenbereich des Gesamtsystems besteht nicht aus unstrukturierten Variablen, sondern aus einer Menge von abstrakten Objekten eines ADT`s, wobei neben den gemeinsamen Objekten jeder Prozeß private Objekte besitzen kann. Kommunikation zwischen Prozessen kann jedoch nur über gemeinsame Objekte stattfinden. Die Operationen sind ebenfalls keine Übergänge im Zustandsraum mehr, sondern Prozeduren mit E/A-Parametern.

Ein ADT definiert die abstrakte Repräsentation (Wertebereich) der Objekte des Typs, und stellt für den Zugriff auf Objekte des Typs Operationen zur Verfügung. Nur mit diesen Operationen ist ein Zugriff auf abstrakte Objekte möglich. Außerdem werden im ADT für dessen Operationen Zugriffseinschränkungen (Synchronisation, Schutz, allgemeiner Art) definiert.

Für jedes gemeinsame abstrakte Objekt wird eine abstrakte Maschine zur Verwaltung der Zugriffe auf das Objekt - die OV-Maschine - definiert. Diese OV-Maschine setzt genau die im ADT gegebenen Zugriffseinschränkungen auf das Objekt durch, und definiert somit einen Laufzeit-Objektverwaltungsmechanismus für gemeinsame Objekte.

Eine Einbettung der gemeinsamen Objekte und der zugehörigen OV-Maschinen in ein Modell für ein Gesamtsystem erfordert das Vorhandensein einer übergeordneten Instanz. Diese "Supervisor-BSM" (SBSM) enthält als Komponenten eine Menge von ADTs, eine Menge von gemeinsamen ADT-Objekten, für

jedes dieser Objekte eine OV-Maschine, und eine Menge von Prozessen mit privaten ADT-Objekten. Das dynamische Verhalten der SBSM wird durch das Starten und Beenden von Prozeßoperationen gegeben, und, falls ein Zugriff auf gemeinsame Objekte stattfindet, zusätzlich durch die Kommunikation mit den entsprechenden OV-Maschinen.

Die Schutzbetrachtungen in der SBSM lassen sich auf Betrachtungen in OV-Maschinen zurückführen, da diese die einzige Möglichkeit zur Interprozeßkommunikation sind. Es wird deshalb ausführlich auf die Definitionsmöglichkeiten von policies im OV-Modell und auf den entsprechenden Sicherheitsbegriff eingegangen.

Abschließend sei in diesem Abschnitt noch eine grobe Skizze zur Erläuterung der Strukturunterschiede zwischen BSM und deren Interpretation SBSM mit OV-Maschinen gegeben:

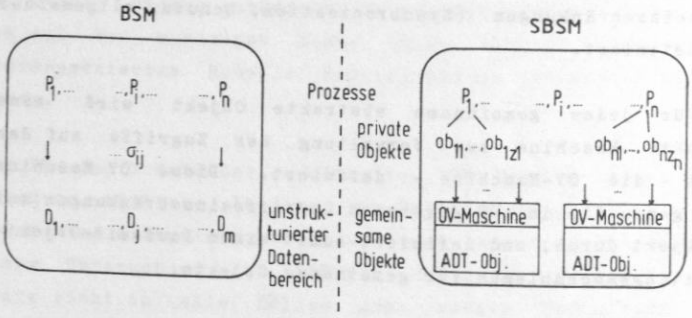


Abb. 5-1: Vergleich zwischen BSM und SBSM

5.3 Die erweiterte Betriebssystemmaschine

5.3.1 Syntax der erweiterten Betriebssystemmaschine

Grundlegend für das Modell der BSM ist der Begriff des sequentiellen Prozesses. Sequentielle Prozesse sind die aktiven Einheiten (in Schutzmodellen "Subjekte") des Modells.

Ein sequentieller Prozeß ist ein Quintupel $P = (A,D,R,ao,Do)$ mit

A	endliche, nichtleere Adreßmenge	
D	nichtleere Datenmenge	
R	Prozeßrelation	$R \subseteq (A \times D) \times (A \times D)$
ao	Anfangsadresse	$ao \in A$
Do	initiale Datenmenge	$Do \in D$

Die Prozeßrelation R beschreibt die möglichen Zustandsübergänge des Prozesses. $Q := (A \times D)$ heißt Zustandsmenge des Prozesses und $q = (a,d) \in Q$ bezeichnet einen Zustand. Die Menge der Aufangszustände des Prozesses ist $\{ao\} \times Do$, $qo = (ao,do)$ mit $do \in Do$ bezeichnet dann einen Anfangszustand des Prozesses.

Für $(q,q') \in R$ bzw. $q \; R \; q'$ (mit $q,q' \in Q$) schreibt man auch $q \dashrightarrow q'$.

Die Semantik des sequentiellen Prozesses P wird durch den Begriff der Berechnung definiert. Eine Folge $\langle qi \mid 0 \leq i \leq m \rangle$ heißt Berechnung des Prozesses P, wenn gilt

$$m \in N \cup \{\infty\} \wedge (\forall i \; (0 \leq i < m)) \; (qi \dashrightarrow qi+1)$$

Auf der Menge der Zustandsübergänge wird durch den Begriff der Operation eine Äquivalenzrelation bezüglich der Adressen eingeführt. Seien $ai,aj \in A$

$$\sigma ij := \{(d,d') \mid (ai,d) \; R \; (aj,d)\}$$

heißt Operation des Prozesses P. Von einem Zustand $q = (ai,d)$ ist ein Übergang zum Zustand $q` = (aj,d`)$ mit Hilfe von σij möglich, wenn $(d,d`) \in \sigma ij$. Man schreibt dafür $q \xrightarrow{\sigma ij} q`$.

Dem Prozeßbegriff liegt die Vorstellung zugrunde, daß der Datenbereich des Prozesses aus einer Menge $Var = \{v1,\ldots,vm\}$ von Datenvariablen besteht. Jede Variable vi besitzt einen Wertebereich Di, so daß gilt $D = D1 \times \ldots \times Dm$. Jedes $d \in D$ ordnet - als Abbildung betrachtet - jeder Variablen $vi \in Var$ einen Wert $d(vi)$ aus dem entsprechenden Wertebereich Di zu.

Für die Integration von Schutz in das Modell der BSM wird von folgenden zusätzlichen Voraussetzungen ausgegangen, die aber keine Beschränkung der Allgemeinheit bedeuten:
Die Variablenmenge des Prozesses sei eingeteilt in "normale" Datenvariable und "Fehlercodevariable", d.h.

$$Var := Var_{\hat{D}} \cup Var_F \quad \text{mit } D := \hat{D} \times F .$$

Auch die Prozeßrelation sei aufteilbar in eine Zustandsübergangsrelation "für den Normalfall" und eine "für den Fehlerfall", d.h. $R := R` \cup R``$. $R`$ führe dabei nur Änderungen in D und $R``$ nur in F durch. Betrachtet man eine Operation $\sigma ij = \{(d,d`) \mid (ai,d)\ R\ (aj,d`)\}$, so läßt sich σij aufgrund der Trennung von R in $R`$ und $R``$ ebenfalls in zwei Komponenten aufteilen, d.h. $\sigma ij = \sigma`ij \cup \sigma``ij$, wobei gelte

(i) $\quad Dom(\sigma`ij) \cap Dom(\sigma``ij) = 0$

(ii) $\quad (\forall (d,d`) \in \sigma`ij)\ (\forall x \in Var_F)\ (d`(x) = d(x))$

(iii) $\quad (\forall (d,d`) \in \sigma``ij)\ (\forall x \in Var_{\hat{D}})\ (d`(x) = d(x))$

Bedingung (i) sagt dabei aus, daß sich die Definitionsbereiche für den Normalfall und den Fehlerfall nicht überschneiden, Bedingung (ii) und (iii), daß durch die "Fehler-

fall-Operation" nur Veränderungen in Var_F, durch die "Normalfall-Operation" nur Veränderungen in $Var_{\hat{D}}$ vorgenommen werden.

Bemerkungen

Die o.g. Voraussetzungen sind keine Beschränkungen der Allgemeinheit, da für die Relation R wie in der BSM gilt

(i) R ist i.a. nicht funktional, da $\sigma`ij$ und $\sigma``ij$ Relationen sind

(ii) R ist i.a. nicht deterministisch, da für zwei von einer Adresse ai ausgehenden Operationen σik und σil nicht gelten muß, daß $Dom(\sigma ik) \cap Dom(\sigma il) = 0$.

Die Verwendung von allgemeinen relationalen Operationen, d.h. von Operationen, die einem Zustand einen aus mehreren möglichen Folgezuständen zuordnen können, ist gerade im Hinblick auf die spätere Integration von Abstraktionsmöglichkeiten besonders wichtig. Bei abstrakten Objekten möchte man nämlich auf einer betrachteten Abstraktionsebene von gewissen Entscheidungen, die erst in tieferen Ebenen fallen, abstrahieren. Für die Modellierung dieses Abstraktionsvorganges ist die Betrachtung nichtfunktionaler Operationen Voraussetzung.

Man kann für jedes σij die Datenmenge D in drei disjunkte Teilmengen D1, D2, D3 einteilen, für die gilt:

$D1 = Dom(\sigma`ij)$: Normalfall, d.h. die Operation σij kann ausgeführt werden

$D2 = Dom(\sigma``ij)$: Fehlerfall für Datenschutz, d.h. die normale Ausführung der Operation σij wird zurückgewiesen und eine Fehlermeldung gesetzt

$D3 = D - [D1 \cup D2]$: Fehlerfall für die Synchronisation, d.h. die Operation σij wird blockiert

Durch die Aufteilung der Operation σij in $\sigma`ij$ und $\sigma``ij$ wurde im vorliegenden Modell gegenüber der BSM die zusätzliche Möglichkeit geschaffen, daß die normale Ausführung der Operation zurückgewiesen wird, was den bisherigen Mechanismus der Blockade als Zugriffseinschränkung ergänzt.

Die Aufteilung von σij in $\sigma`ij$ und $\sigma``ij$ mit disjunkten Definitionsbereichen und disjunkten Ausgabevariablenmengen entspricht durchaus den Gegebenheiten bei "normalen" Operationen in Form von Prozeduren, wo im Normalfall die Ausgabeparameter mit Werten belegt werden, im Fehlerfall ein Fehlercode gesetzt wird (vgl. auch 5.4). Da auf der Ebene der BSM keine prozedurale Abstraktion modellierbar ist, mußte die Einteilung der Prozeßvariablen Var in $Var_{\hat{D}}$ und Var_F, und die Beschränkung von $\sigma`ij$ auf Änderungen in $\hat{D}$ bzw. $\sigma``ij$ auf Änderungen in F erfolgen.

Mit dem gewählten Modell läßt sich auch darstellen, daß eine Operation σjk auf einen Fehlerfall bei der vorausgegangenen Operation σij anders reagiert, als auf einen normalen Abschluß. Dies wird möglich durch eine Abfrage der Fehlercodevariablen Var_F in der Operation σjk.

Wie bei Keramidis soll hier noch den Begriff der Ausführbarkeit für Operationen eines Prozesses P definiert werden. Sei Σi die Menge der Operationen des Prozesses, die von der Adresse ai ausgehen, d.h. $\Sigma i := \{\sigma ij \mid |\sigma ij| \neq 0\}$. Eine Operation $\sigma \in \Sigma i$ heißt <u>im Zustand q ausführbar,</u> in Zeichen.

$ausf(\sigma, q)$ gdw. $q = (a,d) \wedge a = ai \wedge d \in Dom(\sigma)$

Man beachte, daß eine Operation als ausführbar definiert wird, auch wenn der momentane Datenzustand d nicht die operationsinternen Datenschutzbedingungen erfüllt. Der Grund dafür ist, daß die Operation in diesem Fall ja den

Fehlerzweig ausführt. Die Ausführbarkeit einer Operation hängt davon ab, ob die Operation für den momentanen Datenzustand überhaupt definiert ist, sie ist also im Modell der BSM eher eine Synchronisations- als eine Schutzeigenschaft.

Mit den definierten Begriffen des Prozesses und der Operation kann nun die erweiterte Betriebssystemmaschine als Modell für asynchrone Prozesse eingeführt werden.

Man betrachte eine endliche, nichtleere Menge $\overline{P} = \{P1,\ldots,Pn\}$ von sequentiellen Prozessen. Die einzelnen Variablenmengen der Prozesse seien zu einer globalen Variablenmenge Var mit zugeordneter globaler Datenmenge D zusammengefaßt. Die Mengen Var und D lassen sich dann angeben als

$$Var := \bigcup_{k=1}^{n} Var\ k\ ,\ \text{wobei gelte}$$

$$\forall\ k,l\ (1 \leq k,l \leq n \wedge k \neq l \quad \text{---}> Var_{Fk} \cap Var_{Fl} = 0)$$

$$D := \hat{D} \times F1 \times \ldots \times Fn$$

Var k sei hier die dem Prozeß Pk zugeordnete Variablenmenge. Die globale Variablenmenge Var wird durch die Vereinigung der Prozeßvariablenmengen Var k gewonnen, es wird also von globalen Variablenbezeichnungen ausgegangen. Eine Kommunikation zwischen zwei Prozessen Pk und Pl kann dabei über die Variablen aus Var k ∩ Var l stattfinden, wobei durch $Var_{Fk} \cap Var_{Fl} = 0$ ausgedrückt wird, daß die Fehlercodevariablen der einzelnen Prozesse private Prozeßvariablen sind. Die globale Datenmenge D kann somit als $\hat{D} \times F1 \times \ldots \times Fn$ dargestellt werden.

Bezeichne $\Sigma^{(k)}$ die Menge aller Operationen des Prozesses Pk. $\Sigma := \bigcup_{k=1}^{n} \Sigma^{(k)}$ ist dann die Menge der Operationen in der BSM.

Zur Beschreibung der Semantik der BSM werden Tätigkeitszustände für die Prozesse eingeführt. Die Menge aller Tätigkeitszustände wird mit H := H blo x H la x H be x H wa bezeichnet, wobei die H.. Teilmengen der Menge $\mathcal{P}(\overline{P} \times \Sigma)$ sind.

Die Zustandsmenge der BSM ist dann definiert als $QE := \prod_{k=1}^{n} Ak \times D \times H$. Ein Zustand $qe \in QE$ wird mit $qe = (\alpha, d, h)$ bezeichnet, wobei α der Vektor der Adreßvariablen aller Prozesse ist.

Die Betriebssystemmaschine (BSM) ist definiert durch das Tupel BSM = (P,D,Var,R,PRIO,VTLG,Do), wobei gilt

- $\overline{P} = \{P1,...Pn\}$ ist eine endliche, nichtleere Menge von sequentiellen Prozessen, deren Adreßmengen paarweise disjunkt sind, d.h. es gilt

  $\forall k,l\ (1 \leq k,l \leq n \wedge k \neq l \dashrightarrow Ak \cap Al = O)$

- D ist globale Datenmenge, Var globale Variablenmenge, wobei die o.g. Ausführungen gelten

- Die Relation R beschreibt die möglichen Zustandsübergänge der BSM, d.h. $R \subseteq QE \times QE$. Alle $h,h' \in H$ mit $(\alpha,d,h)\ R\ (\alpha',d',h')$ stellen, wenn man sie auf P projeziert, Zerlegungen der Prozeßmenge P dar

- $PRIO \subseteq \overline{P} \times \Sigma \times \overline{P} \times \Sigma \times D$ ist transitive, antisymmetrische und irreflexive Relation, Prioritätsrelation genannt

- $VTGL \subseteq \overline{P} \times \Sigma \times \overline{P} \times \Sigma \times D$ ist symmetrische und reflexive Relation, die Verträglichkeitsrelation

- Sei $dk = d|_{Var\,k}$ die Beschränkung von d auf die Variablenmenge des Prozesses Pk und sei $\pi(qe,k) = (\alpha k, dk)$ die Projektion des Zustands $qe = (\alpha, d, h)$ auf den Prozeß Pk, d.h. αk ist die Adreßvariable des Prozesses Pk.

Es gilt $Do \subseteq D$ und für jedes $do \in Do$ gibt es mindestens einen Prozeß Pk mit der initialen Datenmenge Dko und der Anfangsadresse ako, so daß für $\pi(qo,k) = (\alpha k, dk)$ gilt

$$\alpha k = ako \wedge dk \in Dko$$

Dies bedeutet, daß in jedem möglichen Anfangszustand der BSM mindestens eine Operation eines Prozesses ausführbar ist.

Bemerkungen

Die Tätigkeitszustände beinhalten im gegebenen Modell zusätzlich zum Prozeßnamen noch die jeweilige aktuelle Prozeßoperation des Prozesses. Dies ist deshalb notwendig, weil sowohl die Prioritäts- als auch die Verträglichkeitsrelation auf Prozessen und deren momentanen Operationen definiert ist. Man kann durch diese Ergänzung die in der ursprünglichen BSM notwendigen "worst-case"-Annahmen bei der Definition des Tätigkeitszustands "bereit" (vgl. Ker82 S48/49) und des Startens einer Operation entfallen lassen.

Die Prioritätsrelation definiert Prioritäten zwischen verschiedenen Prozessen, wobei sowohl deren momentan auszuführende Operationen, als auch der aktuelle Datenzustand berücksichtigt werden. Es handelt sich also um eine dynamische Prioritätsrelation, die sehr feine Scheduling-Strategien ermöglicht. Analoges gilt für die Verträglichkeitsrelation, die angibt, wann zwei Operationen unterschiedlicher Prozesse miteinander verträglich sind, d.h. deren gleichzeitige Ausführung keine inkonsistenten Datenzustände

zur Folge haben kann.

5.3.2 Semantik der erweiterten BSM

Die Semantik der BSM wird, analog zur Definition bei sequentiellen Prozessen, durch die Menge aller möglichen Berechnungen und somit durch die Zustandsübergangsrelation R definiert. Um die mit der Relation R möglichen Zustandsübergänge angeben zu können, müssen hier zuerst die in der BSM möglichen Tätigkeitszustände für Prozesse erläutert werden.

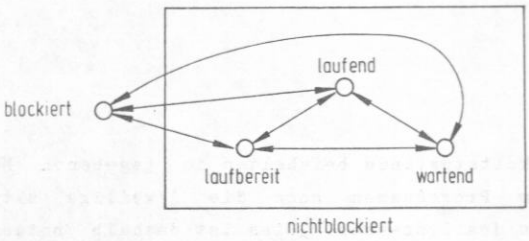


Abb. 5-2: Tätigkeitszustände für Prozesse

Die Interpretation der Zustände erfolgt dabei im Prinzip wie bei der ursprünglichen BSM, wobei hier allerdings Prozeß und aktuelle Operation berücksichtigt werden.

Für die weiteren Ausführungen werden folgende Notationen benutzt:

qe = (α,d,h) bezeichne einen Zustand der BSM.

π(qe,k) = (αk,dk) sei die Projektion von qe auf Pk.

$Op(qe,k) := \{\sigma \mid \sigma = \sigma ij \in \Sigma^{(k)} \wedge \pi(qe,k) =$

$(\alpha k, dk) \wedge \alpha k = ai\}$

sei die Menge der Operationen, die im Zustand qe von der aktuellen Adresse des Prozesses Pk ausgehen.

Eine Operation σ des Prozesses Pk heißt im Zustand qe zulässig, wenn gilt $\sigma \in Op(qe,k)$.

Definitionen

Sei $qe = (\alpha, d, h)$, $Pk \in \overline{P}$, $\pi(qe,k) = (\alpha\, k, dk)$ und $\sigma \in Op(qe,k)$ zulässige Operation von Pk im Zustand qe.

- Eine Operation σ des Prozesses Pk heißt im Zustand qe ausführbar, in Zeichen ausf(Pk, σ, qe), wenn gilt $dk \in Dom(\sigma)$.

- Eine Operation σ des Prozesses Pk heißt im Zustand qe blockiert, in Zeichen $(Pk,\sigma) \in h\ blo(qe)$, wenn gilt $\neg$ ausf(Pk, σ, qe).

- Eine Operation σ des Prozesses Pk heißt im Zustand qe bereit, in Zeichen $(Pk,\sigma) \in h\ be(qe)$, wenn gilt

$$(Pk,\sigma) \notin h\ blo(qe) \wedge (Pk,\sigma) \notin h\ la(qe) \wedge$$

$$\forall (Pl,\sigma`)\,[k \neq l \wedge (Pl,\sigma`) \in h\ la(qe) \rightarrow$$

$$(Pk,\sigma,Pl,\sigma`,d) \in VTGL]$$

- Eine Operation σ des Prozesses Pk heißt im Zustand qe wartend, in Zeichen $(Pk,\sigma) \in h\ wa(qe)$, wenn gilt

$$(Pk,\sigma) \notin h\ blo(qe) \wedge (Pk,\sigma) \notin h\ la(qe) \wedge$$

$$(Pk,\sigma) \notin h\ be(qe)$$

Nach der Initialisierung (d.h. im Zustand qeo = (a1o,...,ano,do,h)) sind die Mengen h wa(qeo) und h la(qeo) leer. Für jeden Prozeß Pk existiert eine zulässige Operation σ . Die eben erfolgten Definitionen legen fest, ob das Paar (Pk, σ) in h be(qeo) oder in h blo(qeo) eingereiht wird.

Durch die folgenden zwei Aktionen "Starten" und "Beenden" einer Operation eines Prozesses wird der Zustand "laufend" verändert und die Zustandsübergänge qe --> qe` der BSM, d.h. die Dynamik der BSM definiert.

Starten einer Operation

Sei qe = (α 1,...,α n,d,h) und qe` = ( $\alpha$ `1,..., α `n,d`,h`).

Ein Zustandsübergang qe ----Start(Pk, σ)----> qe` ist genau dann möglich, d.h. (qe,qe`) ε R wenn gilt

(Pk, σ) ε h be(qe) $\wedge$ ($\forall$ (Pl, σ `) ε h be(qe)) (k $\neq$ l ----->

(Pl, σ `,Pk, σ ,d) $\notin$ PRIO)

Für den Folgezustand qe` gilt dann

(i) $\forall l(1 \leq l \leq n)\ \alpha`l = \alpha\, l$

(ii) $d` = d$

(iii) $h\ la(qe`) = h\ la(qe) \cup \{(Pk, \sigma)\}$

(iv) $h\ blo(qe`) = h\ blo(qe)$

(v) $h\ be(qe`) = h\ be(qe) - (\{(Pk, \sigma)\} \cup NVTGL(Pk, \sigma))$

(vi) $h\ wa(qe`) = h\ wa(qe) \cup NVTGL(Pk, \sigma)$

mit

$$NVTGL(Pk, \sigma) := \{(Pl, \sigma`) \mid l \neq k \wedge (Pl, \sigma`) \in h\ be(qe) \wedge (Pk, \sigma, Pl, \sigma`, d) \notin VTGL\}$$

Eine bereite Prozeßoperation kann gestartet werden, und wird in den Zustand "laufend" versetzt, wenn keine höherpriore Prozeßoperation im Zustand "bereit" ist. Dabei werden alle bereiten Prozeßoperationen, die mit der gestarteten Operation nicht verträglich sind ($NVTGL(Pk, \sigma)$), in den Zustand "wartend" versetzt.

Beenden einer Operation

Sei $qe = (\alpha 1, \ldots, \alpha n, d, h)$ und $qe` = (\alpha`1, \ldots, \alpha`n, d`, h`)$.

Ein Zustandsübergang $qe \xrightarrow{End(Pk, \sigma)} qe`$ ist genau dann möglich, d.h. $(qe, qe`) \in R$, wenn gilt

$(Pk, \sigma) \in h\ la(qe)$.

Für den Folgezustand $qe`$ gilt dann

(i) Sei $\pi(qe,k) = (\alpha k, dk)$ mit $\alpha k = ai$ und $\sigma = \sigma ij \in \Sigma^{(k)}$

Für $\pi(qe\grave{},k) = (\alpha\grave{}k, d\grave{}k)$ gilt $\alpha\grave{}k = aj \wedge$
$(dk, d\grave{}k) \in \sigma ij$

(ii) $\forall l(1 \leq l \leq n \wedge l \neq k \rightarrow \alpha\grave{}k = \alpha k)$
$\forall x(x \in Var - Var\, k \longrightarrow d'(x) = d(x))$

(iii) $h\,la(qe\grave{}) = h\,la(qe) - \{(Pk, \sigma)\}$

(iv) Es wird eine in $qe\grave{}$ zulässige Prozeßoperation $\sigma\grave{}$ ($\sigma\grave{} \in Op(qe,k)$) des Prozesses Pk ausgewählt, und das Paar $(Pk, \sigma\grave{})$ gemäß der angegebenen Regeln in einen der Zustände $h\,blo(qe\grave{})$, $h\,be(qe\grave{})$ oder $h\,wa(qe\grave{})$ eingereiht. Auch die Einstufung der im Zustand qe in den Mengen h blo, h be und h wa befindlichen Prozeßoperationen kann sich - wegen des Beendens von (Pk, σ) und des Datenübergangs auf $d\grave{}$ - ändern. Dies ergibt sich gemäß der früheren Definitionen.

Die Definition des Beendens einer Operation ergibt sich direkt aus den bisherigen Definitionen in <Ker82>. Die Modellierung des Verhaltens im Fehlerfall, d.h. die Nichtausführung der normalen Operation und das Setzen von Fehlercodes muß hier nicht explizit unterschieden werden, da dies bereits innerhalb der Operation σ berücksichtigt wird.

Berechnung der BSM

Der Berechnungsbegriff wird hier völlig analog zum sequentiellen Prozeß eingeführt: Eine Berechnung der BSM ist eine Folge $\langle qei \mid 0 \leq i \leq m \rangle$ mit $m \in N \cup \{\infty\}$, wenn gilt

(i) Für qeo gelten die angegebenen Initialisierungsbedingungen

(ii) $\forall i\ (0 \leq i < m)$ gilt entweder

$$qe\ i \xrightarrow{Start(Pk, \sigma)} qe\ i+1 \text{ oder}$$

$$qe\ i \xrightarrow{End(Pk, \sigma)} qe\ i+1$$

wobei $Pk \in \overline{P} \wedge \sigma \in Op(qe\ i,k)$

Mit den gegebenen Definitionen wurde die um elementare Schutzmechanismen erweiterte BSM eingeführt. In den folgenden Abschnitten sollen nun - basierend auf Untersuchungen zur Abfrage und Veränderung von Variablen in der BSM - Synchronisationsprobleme und Schutzprobleme auf der Ebene der BSM betrachtet werden.

5.3.3 Zur Definition der Abfrage und Veränderung

Grundlegende Voraussetzung für Fragen der Synchronisation und des Schutzes - insbesondere für die Einführung des Begriffs "Informationsfluß" und die Untersuchung der Konsistenz von Verträglichkeitsrelationen - ist die Definition, wann eine Variable durch die Ausführung einer Operation abgefragt oder verändert wird.

Auf den ersten Blick mag dies trivial erscheinen - in der Tat ist dies bei der Veränderung auch so - jedoch wirft die Frage nach der Abfrage einer Variablen durch eine Operation nicht unerhebliche Probleme auf.

Prinzipiell läßt sich zwischen syntaktischer und semantischer Abfrage bzw. Veränderung unterscheiden. Die syntaktische Abfrage bzw. Veränderung ergibt sich dabei aus dem syntaktischen Vorkommen einer Variable auf der rechten bzw linken Seite einer Anweisung. Daraus ergibt sich unmittelbar, daß die Frage, ob eine Variable durch eine Operation syntaktisch abgefragt oder verändert wird, einfach durch eine syntaktische Betrachtung des zugeordneten Programms beantwortet werden kann.

Die semantische Abfrage bzw. Veränderung ist hingegen durch die Werte von Variablen vor und nach der Ausführung der Operation definiert. Insbesondere die Frage, ob eine Variable durch eine Operation semantisch abgefragt wird, oder nicht, ist nicht trivial zu beantworten.

Der Zusammenhang zwischen einer syntaktischen und semantischen Betrachtungsweise besteht nun darin, daß die syntaktische Abfrage bzw. Veränderung eine notwendige, aber nicht hinreichende Voraussetzung für die semantische Abfrage bzw. Veränderung ist, d.h. die syntaktische Betrachtungsweise ist gröber als eine semantische. Hierfür gibt es zwei Gründe:

- Eine Variable wird in dem einer Operation zugeordneten Programm zwar syntaktisch abgefragt/verändert, jedoch wird die entsprechende Programmstelle bei einem konkreten Operationsaufruf übersprungen.

- Eine syntaktische Abfrage/Veränderung einer Variablen durch eine Operation kann semantisch irrelevant sein (Bsp.: "x=x+y-y" oder "y=0; if x=0 then y=x else y=0")

Nur wenige Autoren haben sich bisher mit semantischer Abfrage und Veränderung beschäftigt. Im folgenden werden hier die existierenden Ansätze von Keramidis <Ker82>, Popek und Farber <Pop78> und Cohen <Coh78> näher betrachtet, und gezeigt, daß sie für eine exakte Definition der Begriffe semantische Abfrage bzw. Veränderung aus unterschiedlichen Gründen nicht geeignet sind. Anschließend wird dann ein neuer Ansatz präsentiert, der auch die Basis für die Betrachtungen in Abschnitt 5.3.4 darstellt.

5.3.3.1 Die Definitionen von Keramidis

Im formalen Modell der BSM gibt Keramidis folgende Definitionen für die semantische Abfrage und Veränderung (vgl. <Ker82> S 53ff):

Eine Operation σ kann den Wert einer Variablen u beim Vorliegen eines Datenzustands d ändern, in Zeichen $änd(\sigma,u,d)$ wenn gilt

$$\exists d` ((d,d`) \varepsilon \sigma \wedge d(u) \neq d`(u))$$

Eine Operation σ kann den Wert einer Variablen u beim Vorliegen eines Datenzustands d abfragen, in Zeichen $abf(\sigma,u,d)$ wenn gilt

$$\exists d` \; (d \underset{u}{=} d` \wedge [RSÄ(\sigma,d,u) \neq RSÄ(\sigma,d`,u) \vee RSNÄ(\sigma,d`,u) \neq RSNÄ(\sigma,d,u) \begin{bmatrix} u \\ d`(u) \end{bmatrix}])$$

Dabei sind die Mengen RSÄ, RSNÄ, M und die Relation $\underset{u}{=}$ wie folgt definiert:

$$d \underset{u}{=} d` \quad \text{gdw.} \quad (\forall v \in Var)\ (v \neq u \longrightarrow d(v) = d`(v))$$

$$RSÄ(\sigma,d,u) := \{d` \mid (d,d`) \in \sigma \wedge d(u) \neq d`(u)\}$$

$$RSNÄ(\sigma,d,u) := \{d` \mid (d,d`) \in \sigma \wedge d(u) = d`(u)\}$$

$$M\begin{bmatrix} u \\ x \end{bmatrix} := \{d` \mid \exists d\ (d \in M \wedge d \underset{u}{=} d` \wedge d`(u) = x)\}$$

Wegen der Komplexität der Definitionen soll hier eine kurze Erklärung der zugrungeliegenden Überlegungen gegeben werden.

Die Definition der Änderung änd(σ,u,d) bedeutet, daß es mindestens einen Folgezustand d` ((d,d`) $\in \sigma$) gibt, der sich vom Zustand d im Wert von u unterscheidet.

Wesentlich schwieriger ist die Definition der Abfrage zu erläutern:

$d \underset{u}{=} d`$ bedeutet, daß die Datenzustände d und d` bis auf die Variable u gleich sind

RSÄ(σ,d,u) ist die Menge aller möglichen Folgezustände d` von d (bei Anwendung von $\sigma$ ), bei denen sich der Wert der Variablen u gegenüber dem Ursprungszustand d` verändert hat

RSNÄ(σ,d,u) ergibt sich analog zu RSÄ(...)

$M\left[\begin{smallmatrix} u \\ x \end{smallmatrix}\right]$ ist die Menge, die sich aus M ergibt, wenn man in jedem Element von M den Wert der Variablen u durch x ersetzt.

Der Ausdruck RSÄ(σ,d,u) $\neq$ RSÄ(σ,d`,u) soll dann darstellen, daß sich der Wert der Variablen u bei d bzw. d` unterschiedlich ändert, während RSNÄ(...) $\neq$ RSNÄ(...) ... einen Einfluß von u auf andere Variable darstellen soll.

Die prinzipiellen Überlegungen dieses Ansatzes - nämlich die Unterscheidung des Einflusses einer abgefragten Variablen auf sich selbst bzw. auf andere Variable - sind durchaus richtig gewählt, jedoch ist die formale Darstellung falsch, wie das folgende Beispiel zeigt:

Man betrachte eine beliebige Wertzuweisung "σ : y:=f(x)". Man sieht leicht, daß hier die Variable y syntaktisch verändert, aber nicht abgefragt wird. Legt man die o.g. Definitionen der semantischen Abfrage zugrunde, so ergibt sich eine Abfrage von y.

Seien d = (x,f(x)) und d` = (x,y$\neq$f(x)) zwei Datenzustände, die sich nur im Wert von y unterscheiden, d.h. d $\overset{=}{y}$ d`.

RSÄ(σ,d,y) = 0 da y schon den Wert f(x) besitzt

RSÄ(σ,d`,y) = {(x,f(x))} d.h. der einzig mögliche Folgezustand von d` bei Anwendung von σ ist (x,f(x)); er unterscheidet sich aber von (x,y$\neq$f(x))

Analog ergibt sich RSNÄ(σ,d,y) = {(x,f(x))} und RSNÄ(σ,d`,y) = 0. Daraus ergibt sich dann abf(σ,d,y).

5.3.3.2 Die Definitionen von Popek und Farber

In <Pop78> definieren Popek und Farber mit der Einführung der "Actual Access Relations", wann eine Instruktion ein Objekt lesen oder verändern kann (vgl. 4.1.2). Diese Definitionen werden hier nicht in der Originaldarstellung, sondern in einer an die hier verwendete Notation angepaßten Darstellung präsentiert, um einen Vergleich der Methoden durchführen zu können.

Die Definition der Veränderung einer Variablen durch eine Operation ist völlig identisch zur Definition von Keramidis, so daß hier auf eine nochmalige Darstellung verzichtet werden kann.

Für die Definition der Abfrage wird eine Relation $E[u]$ zwischen zwei Zuständen eingeführt, die exakt der Relation $\underset{u}{=}$ entspricht. Im Gegensatz zur BSM werden hier nur funktionale Operationen betrachtet, wobei mit $\sigma(d)$ derjenige Zustand bezeichnet wird, der sich aus d durch Anwendung der Operation σ ergibt. Dann läßt sich der Abfragebegriff von Popek folgendermaßen definieren:

Eine Operation σ kann den Wert einer Variablen u beim Vorliegen eines Datenzustands d abfragen, in Zeichen abf(σ,u,d) wenn gilt

$$\exists d' \; (d \underset{u}{=} d' \wedge \neg(\sigma(d) \underset{u}{=} \sigma(d')))$$

Es handelt sich hierbei zwar um eine sehr einfache Definition, die jedoch wegen zweier Mängel nur eingeschränkt anwendbar ist:

- Die Definition beschränkt sich auf funktionale Operationen und ist somit für das Modell der BSM ungeeignet
- Der Ausdruck " $\neg(\sigma(d) \underset{u}{=} \sigma(d\grave{}))$" deckt nur den Fall ab, daß die Variable u andere Variable beeinflußt, nicht aber sich selbst, wie dies z.B. in "u:=f(u)" der Fall ist.

5.3.3.3 Die Definitionen von Cohen

Cohen präsentiert in <Coh78> verschiedene Ansätze zur Klärung der Frage, ob bei Ausführung einer Operation Information von einer Variablenmenge A zu einer Variablen b fließen kann. Zwar ist diese Fragestellung etwas anders als bei der Abfrage einer Variablen, trotzdem können die Untersuchungen von Cohen wichtige Aspekte zu einer Abfragedefinition beitragen, insbesondere, weil Cohen in seinem Ansatz als einziger den Einfluß von Zusicherungen über Zustände auf den mit einer Operation möglichen Informationsfluß betrachtet.

An dieser Stelle sei jedoch bereits erwähnt, daß Cohen wie Popek und Farber weder Abhängigkeiten der Form "u:=f(u)" noch relationale Operationen berücksichtigt.

Das Konzept der strengen Abhängigkeit wurde bereits in 4.3.3 detailliert vorgestellt, so daß hier nur noch einmal die Definition präsentiert werden soll, wobei wie bei der Vorstellung des Ansatzes von Popek und Farber eine Umsetzung der Darstellung in die hier verwendete Notation vorgenommen wird.

Die allgemeine Form des Ansatzes für strenge Abhängigkeit lautet: Eine Variable b ist bei der Ausführung der Operation von einer Variablenmenge A streng abhängig (d.h. Information fließt von A nach b) unter der Zusicherung Φ, in Zeichen

$$A \,||\!>^{\sigma}_{\Phi} b \quad \text{gdw.} \quad (\exists\, d,d`)\ (d \underset{A}{=} d` \wedge \Phi(d) \wedge \Phi(d`) \wedge \sigma(d)(b) \neq \sigma(d`)(b))$$

Wie in 4.3.3 bereits angedeutet, erkennt dieser Ansatz nur Informationsflüsse bei sogenannten autonomen Zusicherungen, d.h. Zusicherungen, die nicht Variablen aus A mit solchen nicht aus A verknüpfen. Ein Beispiel für das Versagen des Ansatzes ist die Operation "σ : b:=a1" mit der Zusicherung "Φ =[a1=a2]", wo der Informationsfluß von a1 nach b nicht erkannt wird.

Zur Erweiterung seiner Methode auf nicht-autonome Zusicherungen führt Cohen die beiden Alternativkonzepte "definitive Abhängigkeit" und "verbundene Abhängigkeit" ein. Diesen beiden Betrachtungsweisen liegt ein deduktiver Standpunkt zugrunde, d.h. Information kann bei der Ausführung der Operation σ von A nach b übertragen werden, wenn ein Wert von b nach der Ausführung von σ dazu benutzt werden kann, Eigenschaften der ursprünglichn Werte von A abzuleiten.

Für die formale Einführung der beiden Konzepte sind noch einige Hilfsdefinitionen erforderlich:

- Sei Φ eine Zusicherung, die nach der Ausführung einer Operation σ gelte, dann wird mit $\Phi \circ \sigma$ die schwächste Precondition für Φ bezeichnet, d.h. $\Phi \circ \sigma$ muß vor der Ausführung von σ gelten.

- Sei Φ ein Prädikat, das Variable innerhalb und außerhalb einer bestimmten Variablenmenge A betrifft. Mit Φ_A wird die strengste Schlußfolgerung aus Φ, die nur Variable aus A betrifft, bezeichnet. $\Phi|_A$ bezeichnet die strengste Schlußfolgerung aus Φ über Variable nicht aus A

- Seien Φ und Φ' zwei Prädikate. $\Phi \subseteq \Phi'$ bedeutet, daß die Menge, die durch das Prädikat Φ charakterisiert ist, in der Menge, die durch Φ' charakterisiert ist, enthalten ist.

Eine definitive Abhängigkeit der Variablen b von den Variablen aus A bei der Ausführung von σ liegt dann vor, wenn aus einem Wert von b nach der Ausführung von σ etwas definitives über die ursprünglichen Werte von A geschlossen werden kann. Die formale Definiton lautet:

- b hängt definitiv von A bei der Ausführung von σ ab, wenn gilt

$$(\exists\, \overline{\Phi})\ (\text{false} \subset (\overline{\Phi}_b \circ \sigma)_A \subset \text{true})$$

- Unter der Zusicherung Φ hängt b definitiv von A bei der Ausführung von σ ab, wenn gilt

$$(\exists\, \overline{\Phi})\ (\text{false} \subset (\overline{\Phi}_b \circ \sigma \wedge \Phi)_A \subset \Phi_A)$$

Der Ansatz der definitiven Abhängigkeit funktioniert nun zwar bei nicht-autonomen Zusicherungen, andererseits gibt es aber Fälle, in denen ein definitiver Schluß auf Ausgangswerte von Variablen nicht möglich ist, obwohl ein entsprechender Informationsfluß stattfindet. Die beiden folgenden Beispiele sollen dies erläutern.

Man betrachte die Operation "σ : b:=a1" mit der Zusicherung "Φ = [a1=a2]". Man wähle als $\overline{\Phi}_b$ = [b=k]; daraus ergibt sich $\overline{\Phi}_b \circ \sigma$ = [a1=k], und es gilt

$$(\overline{\Phi}_b \circ \sigma \wedge \Phi)_{a1} = ([a1=k] \wedge [a1=a2])_{a1} = [a1=k] \text{ und}$$

$$\Phi_{a1} = ([a1=a2])_{a1} = \text{true}$$

Da gilt "false $\subset$ [a1=k] $\subset$ true", hängt b unter der Zusicherung "[a1=a2]" definitiv von a1 bei der Ausführung von σ ab, d.h. beim Ansatz der definitiven Abhängigkeit wird trotz der nicht-autonomen Zusicherung "[a1=a2]" und im Gegensatz zum Ansatz der strengen Abhängigkeit für das gegebene Beispiel Informationsfluß von a1 nach b festgestellt.

Anders verhält sich das Konzept der definitiven Abhängigkeit im Beispiel "σ ; if a1≠a2 then b:=0 else b:=1 fi". Mit dem Ansatz der strengen Abhängigkeit ist leicht zu zeigen, daß Information von a1 nach b fließt, für die definitive Abhängigkeit gilt jedoch folgendes:

Wird b nach der Ausführung von σ als 1 beobachtet, so ergibt sich als $[b=1] \circ \sigma = [a1=a2]$, und $([a1=a2])_{\sigma 1} = true$, d.h. b ist nicht definitiv abhängig von a1. Analoges gilt für [b=0].

Bei näherer Betrachtung dieses Beispiels ergibt sich, daß zwar [a1=a2] nichts definitives über a1 aussagt, aber dennoch Information über a1 beinhaltet - aber nur in Verbindung mit einer zusätzlichen Information über a2. Allgemein läßt sich dann das Konzept der verbundenen Abhängigkeit so definieren:

Eine verbundene Abhängigkeit der Variablen b von der Variablenmenge A bei der Ausführung von σ liegt dann vor, wenn aus einem Wert von b nach der Ausführung von σ und einer zusätzlichen Information, die nicht A betrifft, etwas definitives über die ursprünglichen Werte von A geschlossen werden kann, oder formal:

$A \mid\!>^{\sigma} b$ gdw $(\exists \bar{\Phi}, \hat{\Phi})$ [false $\subset (\bar{\Phi}_b \circ \sigma \wedge \hat{\Phi}|_A)_A \subset$ true]

$A \mid\!>^{\sigma}_{\Phi} b$ gdw $(\exists \bar{\Phi}, \hat{\Phi})$

[false $\subset (\bar{\Phi}_b \circ \sigma \wedge \hat{\Phi}|_A \wedge \Phi)_A \subset (\hat{\Phi}|_A \wedge \Phi)_A$]

Auch diese Definition sei an einem Beispiel erläutert: Sei "σ : b:=a1+a2" und Φ = [a1=a3].

Aus $\bar{\Phi}_b$ = [b=k] ergibt sich $\bar{\Phi}_b \circ \sigma$ = [a1+a2=k]. Wählt man $\hat{\Phi}|_{a1}$ = [a2=1] als zusätzliche Information, die nicht a1 betrifft, dann gilt

$(\bar{\Phi}_b \circ \sigma \wedge \hat{\Phi}|_{a1} \wedge \Phi)_{a1}$ = ([a1+a2=k] $\wedge$ [a2=1] $\wedge$ [a1=a3])$_{a1}$ =

([a1+1=k] $\wedge$ [a1=a3])$_{a1}$ = [a1=k-1]

$(\hat{\Phi}|_{a1} \wedge \Phi)_{a1}$ = ([a2=1] $\wedge$ [a1=a3])$_{a1}$ = true

Da [a1=k-1] $\subset$ true, gilt a1 $\mid\!>^{\sigma}_{\Phi}$ b, was weder von der strengen, noch von der definitiven Abhängigkeit erkannt wird.

Das Konzept der verbundenen Abhängigkeit besitzt noch zwei interessante Eigenschaften, die an dieser Stelle erwähnt werden müssen:

Zum einen sind für autonome Zusicherungen Φ die Ansätze der strengen und verbundenen Abhängigkeit gleich, d.h. es gilt:

Falls Φ A-autonom ist, gilt $A \mid\!>^{\sigma}_{\Phi} b$ gdw $A \mid\mid\!>^{\sigma}_{\Phi} b$.

Die zweite Eigenschaft der verbundenen Abhängigkeit ist, daß es bei einer Informationsübertragung von einer Menge A zu einer Variablen b immer mindestens eine Variable $a \in A$ gibt, für die eine "Einzelübertragung" von a nach b

stattfindet, oder formal

$$A \mid>^{\sigma}_{\Phi} b \longrightarrow (\exists a \in A)\ (a \mid>^{\sigma}_{\Phi} b)$$

5.3.3.4 Ein neuer Ansatz für Abfrage und Veränderung

Die vorgestellten Konzepte von Popek und Farber bzw. Cohen haben beide den Mangel, daß sie weder für allgemeine relationale Operationen geeignet sind, noch den Einfluß einer Variablen auf sich selbst bei der Abfragedefinition berücksichtigen. Schon aus diesem Grund sind sie für das Modell der BSM nicht geeignet. Keramidis berücksichtigt zwar diese beiden Fälle, jedoch ist seine Definition rein formal nicht korrekt.

Aus diesem Grund muß hier ein neuer Ansatz präsentiert werden, der die o.g. Mängel nicht besitzt. Im folgenden wird hierfür eine Definition präsentiert, die wie bei Cohen auch den Einfluß von Zusicherungen über Zustände auf die Abfrage und Veränderung berücksichtigt. Dieser Ansatz wird in mehreren Schritten entwickelt, um die gewählte Definition plausibler zu machen.

Die Definition der Veränderung ist bei diesem Ansatz grundsätzlich gleich zu der von Keramidis gewählten. Zusätzlich wird hier allerdings die Definition noch um die Möglichkeit der Integration von Zusicherungen, und anschließend auf Variablenmengen erweitert. Zunächst soll jedoch eine geeignete Definition für die Abfrage hergeleitet werden, wobei diese zuerst für funktionale Operationen angegeben wird, und anschließend auf den relationalen Fall erweitert wird.

Definition der Abfrage

Grundsätzlich lassen sich bei der Abfrage einer Variablen u durch eine Operation σ zwei Fälle unterscheiden, nämlich ein Einfluß von u auf andere Variable v, d.h. v:=f(u), und ein Einfluß auf sich selbst, d.h. u:=f(u).

Der Einfluß der Variablen u auf andere Variable bei der Ausführung von σ läßt sich wie bei Popek und Farber so formulieren:

$$abf(\sigma, u, d) \quad \text{gdw.} \quad \exists d' \; (d \underset{\bar{u}}{=} d' \wedge \neg(\sigma(d) \underset{\bar{u}}{=} \sigma(d')))$$

Den Fall, daß sich u bei der Ausführung von σ im Zustand d selbst beeinflußt, stellt man formal am einfachsten über das Gegenteil dar:

Eine Variable u hat bei der Ausführung von σ im Zustand d keinen Einfluß auf sich selbst, wenn gilt:

- u wird bei der Ausführung von σ im Zustand d überhaupt nicht beeinflußt, d.h. für alle Zustände d, d' ist der Wert von u im Folgezustand $\sigma(d)$, $\sigma(d')$ unverändert, oder formal:

$$d(u) = \sigma(d)(u) \wedge d'(u) = \sigma(d')(u) \qquad (a)$$

- u wird bei der Ausführung von σ im Zustand d zwar beeinflußt, aber nur abhängig von anderen Variablen. Da von zwei Zuständen d, d' ausgegangen wird, die sich höchstens in u unterscheiden, muß in diesem Fall der Wert von u nach Ausführung von σ bei beiden Ausgangszuständen gleich sein, oder formal:

$$\sigma(d)(u) = \sigma(d')(u) \qquad (b)$$

- Eine Abfrage der Variablen u mit Einfluß auf sich selbst ergibt sich dann aus der Negation der beiden Fälle (a) und (b), d.h. aus $\neg((a) \vee (b))$.

Somit lautet die vollständige Definition der Abfrage im funktionalen Fall wie folgt:

Eine Operation σ kann den Wert einer Variablen u beim Vorliegen eines Datenzustands d abfragen, in Zeichen abf(σ,u,d) wenn gilt

$$\exists\, d\hat{}\ (d \underset{u}{=} d\hat{} \wedge [\neg(\sigma(d) \underset{u}{=} \sigma(d\hat{})) \vee \neg(\,(d(u) = \sigma(d)(u) \wedge d\hat{}(u) = \sigma(d\hat{})(u)) \vee \sigma(d)(u) = \sigma(d\hat{})(u)\,)\,]\,)$$

Für die Erweiterung auf den nichtfunktionalen Fall, d.h. wo gilt $|\sigma(d)| \geq 1$, benötigt man noch einige Hilfsdefinitionen:

- $RS(\sigma,d) := \{d\hat{} \mid (d,d\hat{}) \in \sigma\}$ sei die Menge der möglichen Folgezustände von d unter σ.

- Seien B, C zwei Teilmengen der Datenmenge D und $u \in Var$

  $\overline{pr}(B,u) := \{\hat{d} \mid \hat{d} = d|_{Var-\{u\}} \wedge d \in B\}$ sei die Projektion der Menge B auf den Variablenvektor ohne u, d.h. ein Ausblenden von u aus allen Elementen in B findet hier statt.

  $B \underset{u}{=} C$ gdw. $\overline{pr}(B,u) = \overline{pr}(C,u)$, d.h. die Zustandsmengen B und C stimmen überein bis auf die Werte von u

$B(u) := \{\hat{d} \mid \hat{d} = d|_{\{u\}} \wedge d \in B\}$ ist die Projektion der Menge B auf die Variable u.

Im einzelnen lassen sich dann die funktionalen Formulierungen folgendermaßen umsetzen:

1) Einfluß auf andere Variable
$\neg(\sigma(d) \underset{\underline{u}}{=} \sigma(d`))$ geht über in
$\neg(RS(\sigma,d) \underset{\underline{u}}{=} RS(\sigma,d`))$

2) Keine Beeinflussung von u
$d(u) = \sigma(d)(u) \wedge d`(u) = \sigma(d`)(u)$ geht über in
$(\{d(u)\} = RS(\sigma,d)(u) \wedge \{d`(u)\} = RS(\sigma,d`)(u))$

3) Beeinflussung nur durch andere Variable bzw. Konstante
$\sigma(d)(u) = \sigma(d`)(u)$ geht über in
$(RS(\sigma,d)(u) = RS(\sigma,d`)(u))$

Die Definition der Abfrage im relationalen Fall lautet dann:

Eine Operation σ kann den Wert einer Variablen u beim Vorliegen eines Datenzustands d abfragen, in Zeichen abf(σ,u,d), wenn gilt

$$\exists d` \, (d \underset{\underline{u}}{=} d` \wedge [\neg(RS(\sigma,d) \underset{\underline{u}}{=} RS(\sigma,d`)) \vee$$

$$\neg(\,(\{d(u)\} = RS(\sigma,d)(u) \wedge \{d`(u)\} = RS(\sigma,d`)(u)\,) \vee$$

$$RS(\sigma,d)(u) = RS(\sigma,d`)(u)\,)\,]\,)$$

Diese Definition läßt sich nun dahingehend erweitern, daß nicht die Abfrage in einem spezielle Datenzustand d betrachtet wird, sondern gefragt wird, ob eine Variable u durch eine Operation σ überhaupt abgefragt oder verändert werden kann. Die entsprechenden Definitionen lauten dann:

$abf(\sigma,u)$ gdw $\exists d\ (abf(\sigma,u,d))$
$änd(\sigma,u)$ gdw $\exists d\ (änd(\sigma,u,d))$

Bevor nun die gegebenen Definitionen um die Integration von Zusicherungen und auf Variablenmengen erweitert werden, seien hier einige Bemerkungen zur Entscheidbarkeit der Prädikate "abf" und "änd" gemacht. Im allgemeinen, d.h. wenn σ beliebige Operation ist, sind diese Prädikate nicht entscheidbar . Dies läßt sich unmittelbar auf das Halteproblem für beliebige rekursive Funktionen zurückführen, wie die folgende Operation zeigt: σ = "if f(x) hält then y:=0". Die Frage, ob x gelesen bzw. y verändert wird, ist auf das Halteproblem von f(x) zurückzuführen.

Auch für rekursive Funktionen σ sind "abf" und "änd" i.a. nicht entscheidbar, da dann zwar die Prädikate "$(d \underset{u}{=} d` \wedge \ldots)$" bzw. "$(d\,\sigma\,d` \wedge d(u) \neq d`(u))$" entscheidbar sind, aber wegen der Nicht-Beschränktheit der Existenzquantoren nicht die Prädikate "abf" und "änd".

Erweiterung um Zusicherungen

Im folgenden werden hier in die Definitionen der Abfrage und Veränderung Zusicherungen bzw. Preconditions eingeführt, die den jeweiligen Zustand vor Ausführung der Operation charakterisieren:

Sei Φ ein Prädikat über die Datenmenge D. Eine Operation σ kann den Wert einer Variablen u unter der Zusicherung abfragen , in Zeichen $abf(\sigma,u,\Phi)$, wenn gilt

$$\exists d\ (\Phi(d) \wedge abf(\sigma,u,d))$$

Analog dazu kann eine Operation σ den Wert einer Variablen u unter der Zusicherung Φ ändern , in Zeichen änd(σ ,u, Φ) , wenn gilt

$$\exists d \; (\Phi(d) \wedge \text{änd}(\sigma, u, d))$$

Allgemein kann die Angabe einer Zusicherung die Möglichkeit der Abfrage einer Variablen gegenüber dem allgemeinen Fall abf(σ ,u) nur einschränken, aber nie erhöhen. Für zwei Prädikate $\Phi 1$, $\Phi 2$ über D läßt sich deshalb zeigen

$$(\Phi 1 \dashrightarrow \Phi 2) \longrightarrow (\text{abf}(\sigma, u, \Phi 1) \dashrightarrow \text{abf}(\sigma, u, \Phi 2))$$

Nachdem der ursprüngliche Ansatz für die Abfrage und Veränderung um die Möglichkeit der Integration von Eingangszusicherungen erweitert wurde, ist es interessant, einen direkten Vergleich zwischen dem hier präsentierten Ansatz und den Konzepten von Cohen anzustellen.

Der offensichtlichste Unterschied zwischen beiden Ansätzen ist, daß Cohen auf dem Begriff des Informationsflusses aufbaut, d.h. in seinen Definitionen eine Übertragung von Information zwischen einem Sender und Empfänger betrachtet, während hier mit dem Begriff der Abfrage einer Variablen nur der potentielle Sender von Information betrachtet wird; (der Begriff Informationsfluß wird hier dann in Abschnitt 5.3.4.2 als Kombination von Abfrage und Veränderung definiert!).

Cohen definiert Informationsfluß auf zwei Arten. Im Konzept der strengen Abhängigkeit findet Informationsfluß statt, wenn eine Mannigfaltigkeit des Senders zum Empfänger übertragen werden kann. Der deduktive Standpunkt der verbundenen Abhängigkeit erkennt Informationsfluß, wenn aus einem Wert der Empfängervariablen nach der Übertragung auf Werte der Sendervariablen vor der Übertragung geschlossen

werden kann.

Bei beiden Arten kann durch Angabe einer Zusicherung Φ ein Informationsfluß völlig unterbunden werden. Im Konzept der strengen Abhängigkeit ist dies der Fall, wenn durch Φ die mögliche Mannigfaltigkeit des Senders zu stark eingeschränkt wird, was z.B. auch bei nicht autonomen Zusicherungen der Fall ist. Exakter ist hier die Definition der verbundenen Abhängigkeit: Hier findet kein Informationsfluß mehr statt, wenn Φ die Sendevariablen so stark einschränkt, daß aus den Werten der Empfängervariablen nur mehr Φ geschlossen werden kann. Als Beispiel hierfür wäre "σ : b:=a" mit " Φ : [a=k]", wo kein Informationsfluß von a nach b diagnostiziert wird (auch nicht bei strenger Abhängigkeit!).

Betrachtet man dieses Beispiel mit dem Ansatz dieser Arbeit, so stellt man fest, daß hier - obwohl unter der Zusicherung Φ i.e.S keine (neue , d.h. über Φ hinausgehende) Information von a nach b fließt - die Variable abgefragt wird.

Dieser Unterschied läßt sich auch rein formal durch Gegenüberstellung der Definitionen der strengen Abhängigkeit in der hier präsentierten Notation mit der Abfragedefinition erklären. Während die strenge Abhängigkeit definiert ist als

$$a \overset{\sigma}{\underset{\Phi}{|\!|\!>}} b \quad \text{gdw} \quad \exists d,d` \; (d \overset{=}{a} d` \wedge \Phi(d) \wedge \Phi(d`) \wedge \sigma(d)(b) \neq \sigma(d`)(b))$$

lautet die Definition der Abfrage

$$abf(\sigma, a, \Phi) \quad \text{gdw} \quad \exists d \; (\Phi(d) \wedge abf(\sigma, a, d)).$$

Der Unterschied zwischen den beiden Konzepten besteht darin, daß hier alle Systemzustände d` als Vergleich betrachtet werden, die sich vom durch Φ charakterisierten

Zustand nur durch den Wert von u unterscheiden, während beim Konzept der strengen Abhängigkeit nur die Systemzustände d` betrachtet werden, für die ebenfalls Φ gilt. Analoges gilt für die verbundene Anbhängigkeit, wo die in einer Zusicherung Φ enthaltene Information über eine Sendevariable - auch wenn diese zu einer Empfängervariablen übertragen wird - nicht als Information betrachtet wird, was einer Beschränkung der Betrachtung auf Zustände entspricht, die die Zusicherung Φ erfüllen.

Zusammenfassend lassen sich die Unterschiede wie folgt darstellen:

- Falls keine Zusicherungen vorliegen, ist der Ansatz dieser Arbeit identisch mit den Konzepten von Cohen. (Natürlich erkennt die hier vorgestellte Definition zusätzlich Abfragen der Form u:=f(u) und ist auf relationale Operationen anwendbar!)

- Beim Vorliegen nicht-autonomer Preconditions erkennt der hier definierte Ansatz - wie die verbundene Abhängigkeit von Cohen - Abfragen einer Variablen richtig.

- Der dieser Arbeit zugrundeliegende Ansatz erkennt eine Abfrage einer Variablen, auch wenn eine Zusicherung Φ über diese Variable den Informationsfluß unterbindet. Schon hier sei erwähnt, daß diese Eigenschaft sehr wesentlich für die Untersuchung von Verträglichkeitsbedingungen ist (vgl. 5.3.4.1).

- Allgemein läßt sich sagen, daß beim vorliegenden Ansatz außer bei Zusicherungen, die den Informationsfluß von einer Variablen unterbinden, das Konzept der Abfrage gleich dem der verbundenen Abhängigkeit von Cohen ist, und in den o.g. Fällen der Ansatz dieser Arbeit strenger ist.

Allerdings geht dadurch die bei Cohen geltende Eigenschaft

"$A \mid\!>^{\sigma}_{\Phi} b \longrightarrow (\exists a \in A)\ (a \mid\!>^{\sigma}_{\Phi} b)$" verloren, so daß die Definition der Abfrage und Veränderung auf Variablenmengen erweitert werden muß, weil ja eine kombinierte Abfrage nicht durch Untersuchung aller Einzelabfragen ermittelt werden kann. (vgl. auch 5.3.4 und 5.4.4)

Zum Abschluß dieses Abschnitts soll hier noch die notwendige Erweiterung der Definitionen auf Variablenmengen präsentiert werden, bevor dann im Abschnitt 5.3.4 auf Synchronisations- und Schutzprobleme in der BSM eingegangen wird.

Erweiterung auf Variablenmengen

Für die Erweiterung der Definitionen auf Variablenmengen gehe man von einer Menge $U \subseteq Var$ aus. Mit den erweiterten Hilfsdefinitionen

$d \bar{=}_U d'$ gdw. $\forall v\ (v \in Var \wedge v \notin U \longrightarrow d(v) = d'(v))$

$\overline{pr}(B,U) := \{\hat{d} \mid \hat{d} = d\big|_{Var-U} \wedge d \in B\}$

$B \bar{=}_U C$ gdw. $\overline{pr}(B,U) = \overline{pr}(C,U)$

$B(U) := \{\hat{d} \mid \hat{d} = d\big|_{U} \wedge d \in B\}$

läßt sich die Abfrage einer Variablenmenge wie folgt definieren.

Eine Operation σ kann den Wert einer Menge von Variablen U beim Vorliegen eines Datenzustands d abfragen, in Zeichen $abf(\sigma,U,d)$ wenn gilt

$$\exists d' \; (d \underset{U}{=} d' \wedge [\, \neg(RS(\sigma,d) \underset{U}{=} RS(\sigma,d') \vee$$

$$\neg(\, (\{d(U)\} = RS(\sigma,d)(U) \wedge \{d'(U)\} = RS(\sigma,d')(U)) \vee$$

$$RS(\sigma,d)(U) = RS(\sigma,d')(U)\,)\,]\,)$$

Analog dazu läßt sich definieren

$abf(\sigma,U)$ gdw $\exists d\ (abf(\sigma,U,d))$ und

$abf(\sigma,U,\Phi)$ gdw $\exists d\ (\Phi(d) \wedge abf(\sigma,U,d))$

Die Erweiterung der Definition der Veränderung auf Variablenmengen läßt sich direkt auf die Veränderung von Einzelvariablen zurückführen; man kann hierfür definieren

$änd(\sigma,U,d)$ gdw $\exists u\ (u \in U \wedge änd(\sigma,u,d))$

$änd(\sigma,U)$ gdw $\exists u\ (u \in U \wedge änd(\sigma,u))$

$änd(\sigma,U,\Phi)$ gdw $\exists u\ (u \in U \wedge änd(\sigma,u,\Phi))$

5.3.4 Synchronisations- und Schutzbetrachtungen

5.3.4.1 Die Konsistenz von Verträglichkeitsrelationen

Aufbauend auf den gegebenen Definitionen für die Abfrage und Veränderung in der BSM sollen hier zunächst Fragen der Synchronisation in der BSM behandelt und anschließend die Formulierung von Schutzproblemen diskutiert werden.

Die Verträglichkeitsrelation der BSM definiert den Parallelitätsgrad des Systems, indem sie angibt, wann zwei Prozeßoperationen miteinander verträglich sind, d.h. wann sie gleichzeitig ausgeführt werden können, ohne daß inkonsistente Datenzustände entstehen. Die Frage, ob zwei laut Verträglichkeitsrelation verträgliche Operationen wirklich in diesem Sinn verträglich sind, wird als Konsistenz der Verträglichkeitsrelation bezeichnet.

Eine Verträglichkeitsrelation heißt somit konsistent, wenn bei der gleichzeitigen Ausführung mehrerer laut VTGL-Relation verträglichen Operationen jede Operation dieselben Effekte besitzt, als wäre sie alleine ausgeführt worden.

Die Konsistenz von Verträglichkeitsrelationen ist eine ganz entscheidende Voraussetzung für die Sicherheit eines Systems. Keramidis stellt in <Ker82> einen Ansatz zum Nachweis der Konsistenz von VTGL-Relationen vor, der auf der Betrachtung gemeinsam abgefragter und veränderter Variablen basiert.

Demnach ist eine Verträglichkeitsrelation konsistent, wenn sie nur die gleichzeitige Ausführung von Operationen gestattet, falls gilt:

- Die Operationen greifen auf verschiedene Variable zu
- Die Operationen fragen gemeinsame Variable ab, ohne sie zu verändern.

Dieser Ansatz gibt ein hinreichendes Kriterium für die Konsistenz von VTGL-Relationen, allerdings hängt das Ergebnis bei diesem Ansatz und somit der erreichbare Parallelitätsgrad sehr stark von der Definition der Abfrage und Veränderung ab. Legt man eine syntaktische Betrachtungsweise zugrunde, so ist die Einhaltung der obigen Bedingungen zu streng, d.h. man muß auf jeden Fall auf semantische Methoden zurückgreifen. Hier wird im folgenden gezeigt, daß für Konsistenzbetrachtungen an Verträglichkeitsrelationen die verbundene Abhängigkeit von Cohen nicht geeignet ist, und anschließend die Anwendbarkeit des vorgestellten Ansatzes untersucht.

Man betrachte die beiden Operationen
$\sigma 1$: if u then b:=0 else b:=1 fi und
$\sigma 2$: u:= false
unter der Zusicherung Φ : [u=true]

Hier wird von $\sigma 2$ bei Φ die Variable u verändert, während der Ansatz von Cohen bei $\sigma 1$ keine Abfrage von u erkennt, d.h. bei Zugrundelegung des Ansatzes von Cohen wären $\sigma 1$ und $\sigma 2$ unter Φ als verträglich definierbar. Laut Definition der Konsistenz von Verträglichkeitsrelationen wäre eine solche Relation jedoch nicht konsistent, was zur Folge hat, daß der Einsatz der Konzepte von Cohen für Verträglichkeitsfragen nicht geeignet ist.

Bereits im vorigen Abschnitt wurde die Notwendigkeit der Erweiterung der bisherigen Definitionen auf Variablenmengen für die Untersuchung von Verträglichkeitsfragen erwähnt. Dies soll im folgenden an einem Beispiel erläutert werden:

Beispiel

Gegeben seien die drei Operationen

$\sigma 1$: if u or v then x:=1 else x:=2 fi
$\sigma 2$: u:= false
$\sigma 3$: v:= false

Man betrachte die Verträglichkeit der drei Operationen unter verschiedenen Zusicherungen Φ . Die Zustände d und d` werden dabei durch Tripel (u,v,x) repräsentiert.

(i) Φ : [true]: u, v und {u,v} werden von $\sigma 1$ abgefragt
Wähle d=(true,false,x) und d`=(false,false,x)

$RS(\sigma 1,d) = \{(true,false,1)\}$ und
$RS(\sigma 1,d`) = \{(false,false,2)\}$

Da gilt $d \overset{=}{u} d`$, aber $\neg RS(\sigma 1,d) \overset{=}{u} RS(\sigma 1,d`)$, wird u abgefragt. (analog v und {u,v})

Da u und v von $\sigma 2$ bzw. $\sigma 3$ bei Φ : [true] verändert werden, ist $\sigma 1$ mit $\sigma 2$ und $\sigma 3$ nicht verträglich.

(ii) Φ : [u=true]: u und {u,v} werden von $\sigma 1$ abgefragt
Die Abfrage von u ergibt sich wie unter (i), da $\Phi(d)$ gilt. Zur Ermittlung der Abfrage von v muß man einen neuen Zustand d wählen mit $\Phi(d)$ und die Zustände d` mit $d \overset{=}{v} d`$ betrachten.

Wähle d=(true,true,x) mit $\Phi(d)$. Es gibt nur einen Zustand d` mit $d \overset{=}{v} d`$, nämlich d`=(true,false,x).

Da aber gilt

$RS(\sigma 1,d) = \{(true,true,1)\}$ und
$RS(\sigma 1,d`) = \{(true,false,1)\}$,

und somit $RS(\sigma 1,d) \overset{=}{v} RS(\sigma 1,d`)$, und da zusätzlich

gilt $d(v) = \sigma(d)(v)$ und $d'(v) = \sigma(d')(v)$, wird v von $\sigma 1$ bei ϕ : [u=true] nicht abgefragt.

Dies hat zur Konsequenz, daß bei ϕ : [u=true] $\sigma 1$ mit $\sigma 3$ verträglich ist, aber nicht mit $\sigma 2$.

(iii) ϕ : [u=true $\wedge$ v=true]
Hier kann man analog zeigen, daß von $\sigma 1$ weder u noch v einzeln abgefragt werden, jedoch die Menge {u,v}
Somit ist bei ϕ : [u=true $\wedge$ v=true] $\sigma 1$ mit $\sigma 2$ und mit $\sigma 3$ verträglich.

Betrachtet man im Teil (iii) des obigen Beispiels die drei Operationen $\sigma 1$, $\sigma 2$, $\sigma 3$, so liefert der vorgestellte Ansatz eine paarweise Verträglichkeit aller drei Operationen, und somit würde die BSM die gleichzeitige Ausführung aller drei Operationen gestatten.

Tatsächlich sind jedoch zwar die Operationen paarweise miteinander verträglich, jedoch nicht alle drei gleichzeitig.

Zur Lösung dieses Problems gibt es in der BSM grundsätzlich zwei Möglichkeiten:

- Die erste Möglichkeit besteht in einer Erweiterung der Konsistenzbedingung für VTGL-Relationen dahingehend, daß im obigen Beispiel Teil (iii) - um die gleichzeitige Ausführung von $\sigma 1$, $\sigma 2$ und $\sigma 3$ nicht zuzulassen - auch $\sigma 1$ mit $\sigma 2$ und $\sigma 1$ mit $\sigma 3$ nicht als verträglich definiert werden dürfen. Dies ließe sich prinzipiell durch eine Betrachtung von Variablenmengen bei den Konsistenzbedingungen erreichen. Allerdings ist dieser Ansatz zu streng, da bei gleichzeitiger Ausführung von $\sigma 1$ und $\sigma 2$ bzw. $\sigma 1$ und $\sigma 3$ jede Operation dieselben Effekte besitzt, als wenn sie allein ausgeführt worden wäre, also eine gleichzeitige Ausführung eigentlich als konsistent

definiert werden müßte.

- Als Alternative bietet sich eine Erweiterung der Verträglichkeitsdefinition auf die Verträglichkeit einer Prozeßoperation mit einer Menge von Prozeßoperationen an. Dies läßt die Formulierung zu, daß zwar $\sigma 1$ mit $\sigma 2$ und $\sigma 1$ mit $\sigma 3$, aber nicht gleichzeitig $\sigma 1$, $\sigma 2$ und $\sigma 3$ verträglich sind. Allerdings erfordert dieser Ansatz bei der Spezifikation eine Angabe aller verträglichen Kombinationen, da ja hier aus der paarweisen Verträglichkeit einer Menge von Operationen nicht auf deren gleichzeitige Verträglichkeit geschlossen werden kann. So könnte evtl. in vielen Fällen eine Spezifikation über Nichtverträglichkeitsrelationen günstiger für die Systementwicklung sein. Bei der Formulierung der Konsistenzbedingungen für erweiterte VTGL-Relationen muß für diesen Ansatz wie bei der ersten Möglichkeit eine Betrachtung der Abfrage und Veränderung von Variablenmengen erfolgen.

Die als zweite Möglichkeit vorgestellte Erweiterung der Verträglichkeitsdefinition findet sich auch in <Mac83>, einer Arbeit, die vor allem Synchronisationsfragen in der BSM untersucht. Allerdings betrachtet Mackert nicht die Frage nach der Konsistenz der angegebenen Verträglichkeitsrelation, sondern führt die o.g. Erweiterung deshalb ein, weil sich verschiedene Synchronisationsprobleme nicht mit der bisherigen Definition spezifizieren lassen (vgl. <Mac83> S 79-92).

In dieser Arbeit wird nun ebenfalls diese Erweiterung gewählt, jedoch erst bei der Definition der OV-Maschine in das Konzept eingeführt. Die Definition der Konsistenbedingungen für die so erweiterte VTGL-Relation wird dann bei der Untersuchung des Sicherheitsbegriffs für die OV-Maschine im Abschnitt 5.4.4 präsentiert.

An dieser Stelle seien noch einige Bemerkungen zur Definition von Verträglichkeit in der BSM gemacht (vgl. <Ker82>):

Die einfachste Vorgehensweise wäre - wie im Monitor-Konzept von Hoare - alle Operationen als nichtverträglich zu definieren. Dies würde zwar komplexe Konsistenzuntersuchungen überflüssig machen, jedoch einen streng seriellen Ablauf zur Folge haben. Dem gegenüber bringt schon eine syntaktische Betrachtungsweise eine Erhöhung der möglichen Parallelität, die durch semantische Methoden weiter erhöht wird. Trotzdem ist der durch semantische Betrachtung erreichbare Parallelitätsgrad nicht der maximal mögliche, wie das folgende Beispiel zeigt:

Sei $\sigma 1$: if u=100 then ... fi und $\sigma 2$: u:=u+1. Bei semantischer Betrachtungsweise sind $\sigma 1$ und $\sigma 2$ nicht verträglich, da u in $\sigma 1$ abgefragt wird und in $\sigma 2$ verändert wird. Jedoch beeinflussen sich die beiden Operationen nur bei einem Zustand mit u=100, weil bei anderen Zuständen die Veränderung von u durch $\sigma 2$ keinen Einfluß auf die Abfrage in $\sigma 1$ hat. Eine Erweiterung der Konsistenzbetrachtungen auf solche Fälle würde allerdings eine völlig andere Konsistenzbedingung für Verträglichkeit erfordern: Die Abfrage und Veränderung von Variablen/-mengen dürfte nämlich nicht unabhängig voneinander, sondern müßte zusammenhängend betrachtet werden, was allerdings den Rahmen dieser Arbeit erheblich sprengen würde.

5.3.4.2 Zugriffsschutz und Informationsfluß in der BSM

In diesem Abschnitt wird die Darstellung von Schutzproblemen in der BSM anhand von Fragen des Zugriffsschutzes und Informationsflusses erläutert. Hier werden allerdings nur Basisdefinitionen präsentiert, um zu zeigen, daß Schutzprobleme auch im uninterpretierten Modell der BSM formulierbar sind. Eine detaillierte Betrachtung von Schutzfragen wird dann im Modell der OV-Maschine im Abschnitt 5.4 erfolgen.

Bei der Definition der erweiterten Betriebssystemmaschine wurden in das ursprüngliche Konzept der BSM bereits Schutzmechanismen eingeführt, die eine Definition von unterschiedlichen Reaktionen im "Normalfall" und im "Fehlerfall" gestatteten. Allerdings waren diese Schutzmechanismen in den Operationen verborgen: Jede Operation wurde in eine Reaktion für den Normalfall σ' und eine Fehlerfallreaktion σ'' aufgeteilt mit $\sigma = \sigma' \cup \sigma''$. Der Datenbereich D der BSM wurde eingeteilt in einen gemeinsamen Bereich $\hat{D}$ und für jeden Prozeß Pk einen privaten Fehlercodebereich Fk. Alle Operationen $\sigma \in \Sigma^{(k)}$ des Prozesses Pk konnten dabei im Fehlerfall durch den Fehlerzweig σ'' nur Veränderungen in Fk bewirken, und die Operationen $\sigma \in \Sigma^{(k)}$ konnten nur den prozeßeigenen Fehlercodebereich Fk lesen.

Diese Eigenschaften schränken zwar nicht die Allgemeinheit des Modells ein, beschränken jedoch den möglichen Kommunikationsbereich für Prozesse auf die gemeinsame Datenmenge D. Dieser Einschränkung des Kommunikationsbereichs von Prozessen liegt die Forderung zugrunde, daß bereits im formalen Modell der BSM eine Beeinflussung eines Prozesses durch eine Fehlerreaktion eines anderen Prozesses - z.B. einem zurückgewiesenen Zugriffswunsch o.ä. - ausgeschlossen sein sollte. Die gewählte Formulierung ergab sich dabei aus der Darstellung der Operationen als Übergänge im Zustandsraum und dem Fehlen von prozeduralen Abstraktionen in der BSM.

Zugriffsschutz

Das allgemeine Zugriffsschutzproblem

"Kann ein Subjekt beim Vorliegen eines bestimmten Zustands mit einer bestimmten Operation auf den Datenbereich zugreifen?"

läßt sich im Modell der BSM formulieren als

"Kann ein Prozeß Pk im Datenzustand d mit einer Operation auf den Datenbereich zugreifen?"

oder formal

$\underline{zugr(Pk,\sigma,d)}$ gdw $\sigma \in \Sigma^{(k)} \wedge \sigma = \sigma` \cup \sigma`` \wedge d \in Dom(\sigma`)$

Bereits zur Formulierung von Zugriffen auf ein bestimmtes Objekt (=Variable) benötigt man die Definition der Abfrage und Veränderung der BSM.

Die allgemeine Frage

"Kann ein Subjekt (beim Vorliegen eines bestimmten Zustands) (mit einer bestimmten Operation) auf ein bestimmtes Objekt lesend oder schreibend zugreifen?"

stellt sich dar als

"Kann ein Prozeß Pk (im Datenzustand d) (mit einer Operation σ) die Variable u abfragen oder verändern?"

oder formal

$\underline{les(Pk,\sigma,u,d)}$ gdw $\sigma \in \Sigma^{(k)} \wedge abf(\sigma,u,d)$

$\underline{schreib(Pk,\sigma,u,d)}$ gdw $\sigma \in \Sigma^{(k)} \wedge änd(\sigma,u,d)$

Darauf aufbauend lassen sich folgende Variationen definieren:

les
$schreib(Pk,\sigma,u,\Phi)$ gdw $\sigma \in \Sigma^{(k)} \wedge \overset{abf}{änd}(\sigma,u,\Phi)$

les
$schreib(Pk,\sigma,u)$ gdw $\sigma \in \Sigma^{(k)} \wedge \overset{abf}{änd}(\sigma,u)$

$$\text{les schreib}(P_k,u,\Phi) \quad \text{gdw} \quad \exists\sigma\,(\sigma\in\Sigma^{(k)} \wedge \text{abf änd}(\sigma,u,\Phi))$$

$$\text{les schreib}(P_k,u) \quad \text{gdw} \quad \exists\sigma\,(\sigma\in\Sigma^{(k)} \wedge \text{abf änd}(\sigma,u))$$

Man kann diese Abfragen und Veränderungen von Variablen durch Prozesse auch als lokalen Informationsfluß zwischen Subjekt und Objekt bezeichnen.

Informationsfluß

Zur Definition des Begriffs "Informationsfluß" wird hier davon ausgegangen, daß Information in der BSM nur über Variable fließen kann. Es wird also vorausgesetzt, daß verdeckte Kanäle nicht existieren. Zum Problem der verdeckten Kanäle betrachte man z.B. <Lam73>, <Jon78b> oder in <Rei83> S 131ff und die Bemerkungen im Abschnitt 5.4.

Von Interesse ist der Informationsfluß zwischen zwei Prozessen (globaler Fluß) über Objekte. Voraussetzung dafür ist, unter der obigen Annahme, daß es ein gemeinsames Objekt (Variable) geben muß, das von einem Prozeß verändert wird und von einem anderen gelesen wird. Hier wird eine rein statische Informationsflußdefinition eingeführt (vgl <Rei83>), die eigentlich nicht "Information fließt", sondern "Information kann fließen" heißen muß, da in der Definition die Dynamik der BSM nicht berücksichtigt wird. Es wird also nur gefragt, ob eine gemeinsame Variable existiert, die von einem Prozeß verändert werden kann, und von einem anderen gelesen werden kann. Eine Definition, die auch die Dynamik berücksichtigt, wird dann in 5.4.4 für die OV-Maschine präsentiert.

Der Begriff des direkten Informationsflusses läßt sich dann wie folgt definieren:

Von einem Prozeß Pk kann über eine Variable u Information zu einem Prozeß Pl fließen , in Zeichen dflow(Pk,Pl,u) gdw

schreib(Pk,u) ∧ les(Pl,u)

Oft ist es sinnvoll, zusätzliche Bedingungen für den Informationsfluß zwischen zwei Prozessen einzuführen. Die Definition läßt sich dann wie folgt erweitern:

Unter den Bedingungen $\Phi 1$ und $\Phi 2$ kann zwischen den Prozessen Pk und Pl Information über die Variable u fließen, in Zeichen dflow(Pk,Pl,u, $\Phi 1$, $\Phi 2$) gdw

schreib(Pk,u, $\Phi 1$) ∧ les(Pl,u, $\Phi 2$)

Zur Definition des Informationsflusses über ein beliebiges Objekt werden die Mengen

Les(Pk) := {u | les(Pk,u)} und
Schreib(Pk):= {u | schreib(Pk,u)} eingeführt.

Man kann dann definieren

dflow(Pk,Pl) gdw Schreib(Pk) ∩ Les(Pl) ≠ 0 .

Bei der Definition des direkten Informationsflusses fließt die Information zwischen zwei Prozessen Pk und Pl über eine Variable. Jedoch lassen sich auch Fälle vorstellen, in denen die Information von Pk über mehrere andere Prozesse und verschiedene Variable nach Pl fließt. Dies führt zum Begriff des allgemeinen Informationsflusses , der definiert wird als:

Information kann von einem Prozeß Pk zu einem Prozeß Pl fließen, in Zeichen flow(Pk,Pl) gdw

$\exists$ endliche Folge $\langle Pi \mid 1 \leq i \leq m \rangle$

$(Pk=P1 \wedge Pl=Pm \wedge \forall Pi,Pi+1\ (1 \leq i \leq m-1 \longrightarrow dflow(Pi,Pi+1))$

Bemerkung

Diese Basisdefinitionen des Begriffs Informationsfluß entsprechen im wesentlichen den bei <Rei83> angegebenen, wobei dort allerdings auf eine nicht korrekte Abfragedefinition aufgesetzt wird. Nochmals sei erwähnt, daß der hier definierte Informationsflußbegriff rein statisch ist, d.h. die Dynamik der BSM - z.B. ob die entsprechenden Zustände überhaupt erreicht werden können - wird hier nicht berücksichtigt (vgl. hierzu 5.4.4). Aus diesem Grund ist die gegebene Definition auch nur eine notwendige aber nicht hinreichende Bedingung für Informationsfluß. Es sollte mit diesem Abschnitt nur aufgezeigt werden, daß auch im uninterpretierten Modell der BSM der Begriff "Informationsfluß" formulierbar ist, und somit die erweiterte BSM ein allgemeines Basismodell für asynchrone, geschützte Systeme darstellt.

Im folgenden Abschnitt 5.4 sollen nun in das präsentierte Konzept Abstraktionsmöglichkeiten eingeführt werden, und dafür formale Modelle für Abstrakte Datentypen und die abstrakte OV-Maschine entwickelt werden.

5.4 Abstrakte Datentypen und die abstrakte Objektverwaltungsmaschine

Ausgangsbasis für die Einführung der abstrakten OV-Maschine ist die erweiterte BSM des Abschnitts 5.3 mit einem unstrukturierten Zustandsraum als Datenbereich und uninterpretierten Übergängen in diesem Zustandsraum als Operationen. Dieses Basismodell kann nun durch die Einführung des Strukturierungskonzepts "Abstrakter Datentyp" (vgl. Kapitel 3) interpretiert werden, wobei vom ADT-Konzept gleichzeitig prozedurale und Datenabstraktion unterstützt werden: Der unstrukturierte Datenbereich der BSM wird durch eine Menge abstrakter Objekte, d.h. Objekte eines Abstrakten Datentyps, ersetzt, und für jedes abstrakte Objekt werden durch den entsprechenden ADT abstrakte Operationen mit Parametern definiert, die an die Stelle der uninterpretierten Zustandsübergänge treten.

Im formalen Modell für ADTs werden neben der Festlegung der abstrakten Repräsentation (d.h. dem Wertebereich) von Objekten und der auf die Objekte definierten Operationen noch Zugriffseinschränkungen für die Operationen definiert. Auf diese Weise ist vollständig festgelegt, wann und von wem mit welcher Operation auf das abstrakte Objekt zugegriffen werden darf. Diese gemeinsame Definition von Objekt, Operationen und Zugriffsbeschränkungen orientiert sich an den Grundsätzen des Modulkonzepts und der objektorientierten Programmierung. Hierdurch wird dem Lokalitätsprinzip Rechnung getragen, und außerdem die Zuverlässigkeit erhöht, da die Verantwortung für Zugriffe auf Objekte nicht mehr den aktiven Einheiten übertragen wird.

Zur Verwaltung der Zugriffe auf die durch einen ADT definierten abstrakten Objekte wird das formale Modell der OV-Maschine eingeführt. Die OV-Maschine als Laufzeit-Objektverwaltungsmechanismus setzt die im entsprechenden ADT definierten Zugriffseinschränkungen durch.

Die abstrakten Objekte mit ihren OV-Maschinen als passive Einheiten können nun zusammen mit Prozessen als aktive Einheiten zu einem Modell für ein Gesamtsystem vereinigt werden. Auch hier regelt wieder ein Laufzeitmechanismus (in der Praxis das Betriebssystem) das Zusammenspiel von Prozessen und OV-Maschinen. Diese übergeordnete Instanz wird Supervisor-Betriebssystemmaschine (SBSM) genannt.

Man beachte, daß bei der formalen Beschreibung des Modells in dieser Arbeit strikt zwischen den Begriffen "ADT" und "OV-Maschine" unterschieden wird. Ein ADT entspricht der Angabe eines Musters für Objekte des Typs, d.h. er beinhaltet die Abstrakte Repräsentation der Objekte, deren initiale Belegung, die Zugriffsoperationen mit E/A-Parametern und die Zugriffsbeschränkungen. Die OV-Maschine definiert dagegen den Laufzeit-Verwaltungsmechanismus für Objekte eines ADT`s. Ihre Aufgaben bestehen im Entgegennehmen der Operationsaufrufe, der Durchsetzung der Einschränkungen, der Ausführung, Verzögerung oder Zurückweisung von Operationsaufrufen und schließlich in der Ausgabe der Resultate an den aufrufenden Prozeß bzw. an die übergeordnete Supervisor-BSM. Ein Analogon zur Beziehung zwischen ADT und OV-Maschine wäre z.B. der Zusammenhang zwischen Turingprogramm und Turingmaschine.

Im folgenden sollen nun formale Modelle für Abstrakte Datentypen und die OV-Maschine angegeben werden. Hierbei ist besonders zu beachten, daß die formalen Modelle problemadäquat sind und so definiert werden, daß sich ein einheitlicher Ansatz vom formalen Modell über die Spezifikation bis hin zur Implementierung ergibt. Aus diesem Grund findet bei der Erstellung der Modelle eine Orientierung an sprachlichen Ansätzen, wie sie z.B. in <Wul76>, <Ker82> oder für Schutzkonstrukte in <Rei83> gegeben sind, statt.

5.4.1 Einführung des Strukturierungskonzepts ADT

Abstrakte Datentypen geben ein Muster für abstrakte Objekte des entsprechenden Typs an. Die bei <Ker82> gegebene sprachliche Definition von ADTs beinhaltet dabei

- den Wertebereich für Objekte des Typs (Abstrakte Repräsentation)
- die initiale Belegung, d.h. den Anfangszustand für Objekte des Typs
- eine Menge von Operationen auf den Wertebereich der Objekte mit Ein- und Ausgabeparametern und
- die Angabe der Synchronisationseinschränkungen ähnlich wie bei der BSM, d.h. in Form von Verträglichkeits- und Prioritätsrelationen. Voraussetzung hierfür ist eine Definition von sogenannten Aktivitätenmengen, d.h. Operationen mit einer entsprechenden Belegung der Eingabeparameter.

Aufbauend auf dieser sprachlichen Basisdefinition wird das dieser Arbeit zugrundeliegende formale Modell für ADTs eingeführt, wobei zusätzlich noch Schutzeinschränkungen berücksichtigt werden. Die hierbei zu beachtenden Randbedingungen seien im folgenden erläutert.

5.4.1.1 Zur Definition der Schutzeinschränkungen für ADTs

Entscheidende Voraussetzung für eine sinnvolle Definition von Schutzeinschränkungen im formalen Modell ist eine strikte Trennung zwischen policy, d.h. der Schutzstrategie, und dem Schutzmechanismus, der diese policy durchsetzt. Natürlich muß hierbei die policy in einer solchen Form angegeben werden, daß sie vom Mechanismus - d.h. der OV-Maschine - durchgesetzt werden kann. Die hierfür möglichen Vorgehensweisen lassen sich einführend sehr gut anhand von Beispielen aus der Synchronisation erläutern. Man betrachte hierzu die Formulierungsmöglichkeiten von Synchronisationspolicies und die entsprechend notwendigen Synchronisationsmechanismen (vgl. 5.3.3 und 5.3.4):

- Die erste Möglichkeit besteht in einer expliziten Angabe der miteinander verträglichen Aktivitäten in der VTGL-Relation. Die so angegebenen Synchronisationspolicies sind von einem Synchronisationsmechanismus durch einen einfachen Vergleich der laufenden Aktivitäten mit der auszuführenden Aktivität durchsetzbar (vgl. BSM). Sowohl einfache syntaktische als auch feinere semantische Betrachtungsweisen, die den Objektzustand und Parameterwerte mit einbeziehen, sind realisierbar, d.h. vom gegenseitigen Ausschluß aller Aktivitäten bis zur maximal möglichen Parallelität sind alle Abstufungen möglich. Für den Beweis der Sicherheit muß hier die Konsistenz der angegebenen Verträglichkeitsrelation nachgewiesen werden, d.h. ob die laut VTGL-Relation verträglichen Aktivitäten wirklich miteinander verträglich sind (vgl. 5.3.4).

  Hierzu ist es notwendig, die von einer Aktivität abgefragten und veränderten Objektkomponenten zu analysieren. Legt man nur die einfache syntaktische Betrachtungsweise zugrunde, so ist die Frage nach der Konsistenz einer VTGL-Relation entscheidbar. Dies trift bei semantischer Betrachtung im allgemeinen nicht zu. Trotzdem wird man wegen des höheren erreichbaren Parallelitäts-

grads auf semantische Methoden nicht verzichten können, zumal sicherlich bei den meisten praktischen Problemen die Konsistenzfrage auch hier entscheidbar ist.

- Eine zweite Möglichkeit ist, nicht explizit anzugeben, welche Aktivitäten miteinander verträglich sein sollen, und stattdessen für jede Aktivität zu spezifizieren, welche Objektkomponenten von ihr abgefragt und verändert werden. Diese Spezifikationen der Aktivitäten entsprechen den bei Popek und Farber eingeführten Intended Access Relations (IAR). Aufgrund der spezifizierten Zugriffe einer Aktivität kann der entsprechende Objektverwaltungsmechanismus dann selbständig entscheiden, welche Aktivitäten miteinander verträglich sind. Durch eine entsprechende Angabe sind in den IARs sowohl syntaktische als auch semantische Betrachtungen möglich. Für rein syntaktische Betrachtungen könnten diese IARs z.B. zur Compilierungszeit ermittelt werden. Dies hat allerdings den Nachteil, daß von worst-case-Annahmen ausgegangen werden muß. Ein Beispiel hierfür wäre eine "if..then..else" Anweisung, wo der then- und else-Teil betrachtet werden muß, und nicht der von Objektwert und Parametern abhängig aktuell eingeschlagene Zweig. Dieser Nachteil fällt bei einer semantischen Betrachtungsweise weg.

Für den Beweis der Sicherheit müssen hier die IARs verifiziert werden. Dies ist ein völlig äquivalentes Problem zu den bei der ersten Möglichkeit notwendigen Verifikationen, so daß es prinzipiell keine Rolle spielt, ob die BSM aufgrund von VTGL-Relationen entscheidet, oder aufgrund von IARs die Verträglichkeiten ermittelt und dann entscheidet. Der Konsistenzbeweis ist in beiden Fällen von gleichem Komplexitätsgrad. Die erste Methode hat allerdings den Vorteil, daß mit ihr beliebige Synchronisationsstrategien realisiert werden können, während im zweiten Fall immer eine maximale Parallelität erreicht wird, auch wenn dies gar nicht notwendig oder erwünscht

ist, weil z.B. die Performance darunter leidet. Ein weiterer Nachteil dieser Methode ist, daß eine genaue Spezifikation der IARs hier in jedem Fall notwendig ist, auch wenn die zu realisierende Verträglichkeitsstrategie so einfach ist, daß sie z.B. nach der ersten Methode ohne Probleme auf Konsistenz überprüft werden könnte. Dies wirkt sich vor allem auf die Komplexität und den Zeitaufwand für die Spezifikation aus, so daß ein Vorgehen nach der ersten Methode vor allem für den praktischen Einsatz sicherlich vorzuziehen ist.

- Eine weitere denkbare Möglichkeit wäre, weder explizit die verträglichen Aktivitäten anzugeben, noch eine Spezifikation der Zugriffe von Aktivitäten in der Form von IARs. Der Objektverwaltungsmechanismus müßte in diesem Fall selbständig entscheiden, ob zwei Aktivitäten miteinander verträglich sind oder nicht. Dies würde eine Ermittlung der Zugriffe zur Laufzeit erfordern, die in der Praxis vom Betriebssystem mit Unterstützung durch die HW durchgeführt werden müßte. Dies ist allerdings von heutigen Betriebssystem- und Hardwarestrukturen nicht lösbar, so daß diese Möglichkeit zumindest zur Zeit nicht realisierbar ist.

Die hier für die Synchronisation aufgezeigten Möglichkeiten lassen sich nun prinzipiell auch auf den Bereich Schutz übertragen. Auch hier muß die Angabe der policies so erfolgen, daß sie vom Mechanismus (der OV-Maschine) durchgesetzt werden können. Der Unterschied zur Synchronisation besteht nur darin, daß bei nicht erfüllter policy die auszuführende Aktivität nicht verzögert, sondern zurückgewiesen wird.

Für den Bereich Schutz lassen sich verschiedene Arten von policies differenzieren. Die entsprechenden Begriffe wurden schon bei der BSM definiert, allerdings nicht in Form

von policies, die von der BSM durchzusetzen sind. Man unterscheidet zwischen

- Zugriffsschutz, d.h. ob eine Aktivität bei einem bestimmten Objektzustand ausgeführt werden darf oder nicht,

- Informationsfluß innerhalb einer Aktivität, d.h. ob eine Aktivität bei einem bestimmten Zustand gewisse Objektkomponenten abfragen oder verändern darf, und

- Informationsfluß zwischen Aktivitäten, d.h. ob Information von einer Aktivität zu einer anderen fließen darf.

Eine Zugriffsschutzpolicy kann definiert werden durch eine Menge von Paaren, bestehend aus einer Aktivität und einem Objektzustand. Für jedes Paar aus der Menge gilt, daß die angegebene Aktivität beim entsprechenden Objektzustand zurückgewiesen werden soll. Diese policy ist von der OV-Maschine als Schutzmechanismus durchsetzbar, da ja nur ein Vergleich der aktuell auszuführenden Aktivität und des aktuellen Objektzustands mit der Menge der Paare in der policy durchgeführt werden muß. Natürlich werden in der Praxis bei der Spezifikation diese policies nicht durch Mengen von Paaren angegeben, sondern durch Prädikate über die Menge aller Aktivitäten und die Zustände des Objekts charakterisiert (vgl. Kapitel 6). Für das formale Modell ist jedoch die Darstellung als Menge von Paaren, d.h. als Relation adäquater.

Informationsflußpolicies für eine Aktivität legen fest, ob eine Aktivität bei einem bestimmten Zustand eine Objektkomponente lesen oder verändern darf. Sie bestehen also aus einer Menge von Tupeln (Aktivität, Objektzustand, Teilmenge von Objektkomponenten, Teilmenge von Objektkomponenten), wobei die zwei Teilmengen von Objektkomponenten diejenigen Komponenten bezeichnen, die von der Aktivität im entspre-

chenden Objektzustand gelesen oder verändert werden dürfen. Aufgabe des OV-Mechanismus, der eine solche policy durchsetzen soll, ist eine Zurückweisung der Aktivität, falls diese auf andere Komponenten als angegeben lesend oder verändernd zugreifen kann.

Informationsflußpolicies zwischen Aktivitäten legen fest, ob Information zwischen zwei Aktivitäten fließen darf oder nicht. Der zugrundeliegende Informationsflußbegriff ist dabei so definiert, daß ein Informationsfluß von einer Aktivität x zu einer Aktivität y stattfinden kann, wenn von x eine Objektkomponente verändert werden kann, die von y gelesen werden kann. Letztendlich basiert also auch dieser erweiterte Informationsflußbegriff auf der Abfrage- und Veränderungsdefinition. Eine policy besteht hier einfach aus einer Menge von Tripeln (Aktivität, Aktivität, Prädikat), wobei das Prädikat angibt, unter welcher Voraussetzung Information von der ersten zur zweiten Aktivität fließen darf. Aufgabe des Laufzeit-OV-Mechanismus ist dann, nicht erlaubte Informationsflüsse zu verhindern, indem die zweite, lesende Aktivität, wenn sie auf eine Objektkomponente zugreifen will, die von der ersten Aktivität zuvor verändert wurde, zurückgewiesen wird.

Der große Unterschied zu den bisherigen policies und ihrer Durchsetzung durch den Laufzeit-OV-Mechanismus ist hier, daß der Faktor "Zeit", d.h. die Historie der Berechnung der Maschine eine Rolle spielt, was sich im Wort "zuvor" ausdrückt. Im Laufe dieses Abschnitts 5.4 wird gezeigt, daß der bisher für die BSM definierte statische Informationsflußbegriff, der auch in <Rei83> verwendet wird, aus diesem Grund nicht ausreicht und durch einen dynamischen Informationsflußbegriff ersetzt werden muß (vgl. 5.4.4).

Wie die Informationsflußpolicies für eine Aktivität sind Informationsflußpolicies zwischen Aktivitäten von der OV-Maschine nicht direkt durchsetzbar, da dies eine Zugriffsermittlung zur Laufzeit erfordern würde. Zur Durchsetzung dieser policies bieten sich aber zwei Möglichkeiten

an, die in Abbildung 5-3 überblicksartig dargestellt sind.

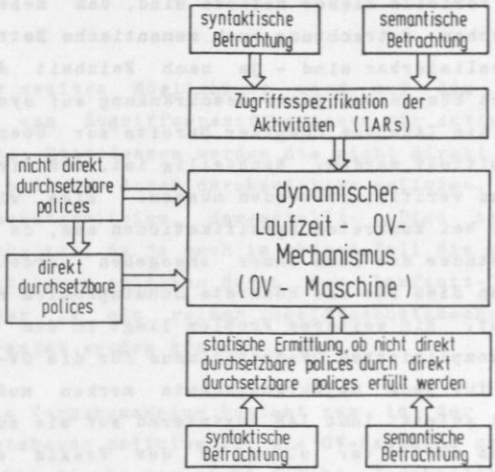


Abb. 5-3: Möglichkeiten zur Integration von Informationsflußpolicies

Die Methoden basieren auf einer Kombination von statischer Zugriffsermittlung und dynamischer Zugriffsüberwachung, und entsprechen im wesentlichen den bei der Synchronisation genannten:

- Die erste Möglichkeit besteht in der zusätzlichen Angabe von Zugriffsspezifikationen oder Intended Access Relations (IARs) für die Aktivitäten. Hier kann die OV-Maschine durch einen Vergleich zwischen policy und IAR zur Laufzeit entscheiden, ob die aktuelle Aktivität zurückgewiesen wird, oder nicht. Diese Methode wird auch bei <Rei83> verwendet; man beachte jedoch, daß dort die Mengen CHANGESET und DEPENDSET rein syntaktisch ermittelt

werden.

Die Vorteile dieser Methode sind, daß neben dieser syntaktischen Betrachtung auch semantische Betrachtungsweisen realisierbar sind - je nach Feinheit der IARs. Zusätzlich können bei einer Beschränkung auf syntaktische Analysen die IARs vom Compiler bereits zur Übersetzungszeit ermittelt werden. Nachteilig ist, daß die IARs erstellt und verifiziert werden müssen. Dies wirkt sich vor allem bei konkreten Spezifikationen aus, da hier nach dieser Methode die IARs immer angegeben werden müssen, auch wenn dies für das konkrete Schutzproblem nicht notwendig ist. Ein weiteres Problem liegt in dem erforderlichen komplizierten OV-Mechanismus für die OV-Maschine, der sich für jede Objektkomponente merken muß, welche Aktivität zuletzt laut IAR verändernd auf sie zugegriffen hat. Dies dürfte vor allem in der Praxis erhebliche Speicherplatz- und auch Laufzeitprobleme mit sich bringen.

Man beachte noch, daß bei dieser Methode - wenn nur eine syntaktische Betrachtung zugrundegelegt wird (wie in <Rei83>) - unter Umständen nicht erlaubte Informationsflüsse nicht verhindert werden können. Ein Beispiel soll dies erläutern:

Gegeben seien drei Aktivitäten x, y und z. Informationsflüsse seien erlaubt von y nach z, aber nicht von x nach z. Es existiere eine gemeinsame Objektkomponente, die von x und y syntaktisch verändert wird und von z syntaktisch und semantisch gelesen wird. Zuerst werde die Aktivität x ausgeführt, wobei die entsprechende Objektkomponente semantisch verändert werde. Bei der anschließenden Ausführung von y finde aber aufgrund des Objektzustands keine semantische Veränderung der Komponente statt. Eine anschließend auszuführende Aktivität z wird nun aufgrund der syntaktischen Betrachtungsweise nicht zurückgewiesen, da ja ein Informationsfluß von y nach z stattfinden darf. Die Komponente enthält aber zu

diesem Zeitpunkt noch die von der Aktivität x eingetragene Information, so daß ein unerlaubter Informationsfluß stattfindet. Der auf syntaktischer Analyse aufbauende OV-Mechanismus kann diesen nicht verhindern.

- Bei der zweiten Möglichkeit wird auf die zusätzliche Angabe von Zugriffsspezifikationen für Aktivitäten verzichtet. Stattdessen werden die nicht direkt durchsetzbaren policies durch durchsetzbare policies, nämlich Zugriffsschutzpolicies, dargestellt. Dies ist deshalb immer möglich, da ja auch im obigen Fall die nicht direkt durchsetzbaren policies durch den Laufzeit-OV-Mechanismus, der ja ein reiner Zugriffsschutzmechanismus ist, durchgesetzt werden können.

Die Vorgehensweise besteht nun in der Angabe von durchsetzbaren policies für die OV-Maschine und einer zusätzlichen Angabe der nicht direkt durchsetzbaren policies als zu beweisende Zusicherungen, die durch die Einhaltung der durchsetzbaren policies erfüllt werden. Der Beweis, daß die angegebenen durchsetzbaren policies auch die als Zusicherungen gegebenen nicht durchsetzbaren policies realisieren, muß dann statisch durchgeführt werden. Hierzu ist zu bemerken, daß dieser Beweis auch nicht schwieriger als die Ermittlung und der Konsistenzbeweis der IARs, die ja im ersten Fall notwendig werden, ist.

Der große Vorteil gegenüber der ersten Methode liegt nun darin, daß eine Ermittlung von IARs für die Aktivitäten nicht mehr nötig ist. Dies erleichtert vor allem die Spezifikation einfacher policies, da ja der Ballast der Ermittlung der IARs wegfällt.

Wie bei der Bestimmung von IARs sind natürlich auch hier beim Beweis der Erfüllung nicht direkt durchsetzbarer policies durch die angegebenen durchsetzbaren policies sowohl syntaktische als auch semantische Betrach-

tungsweisen möglich. Ein weiterer Vorteil dieser Methode liegt noch darin, daß der in der OV-Maschine zu realisierende Mechanismus sehr einfach gestaltet werden kann und trotzdem beliebige policies realisierbar sind.

Ein gewisser Nachteil der Methode liegt darin, daß bei komplexen Informationsflußpolicies die zu findenden Zugriffsschutzpolicies, die diese erfüllen, natürlich ebenfalls sehr komplex werden, da in ihnen ja die gesamte Berechnungsvorgeschichte berücksichtigt werden muß. Für die praktische Anwendung der Methode erscheint es jedoch günstiger, in diesen seltenen Fällen eine Erhöhung der Komplexität der Spezifikation in Kauf zu nehmen, dafür aber bei Standardsituationen eine einfach handhabbare Methode zur Verfügung zu haben.

Zusammenfassend soll nun die hier verwendete Methode zur Formulierung von Schutzeinschränkungen noch einmal überblicksartig dargestellt werden:

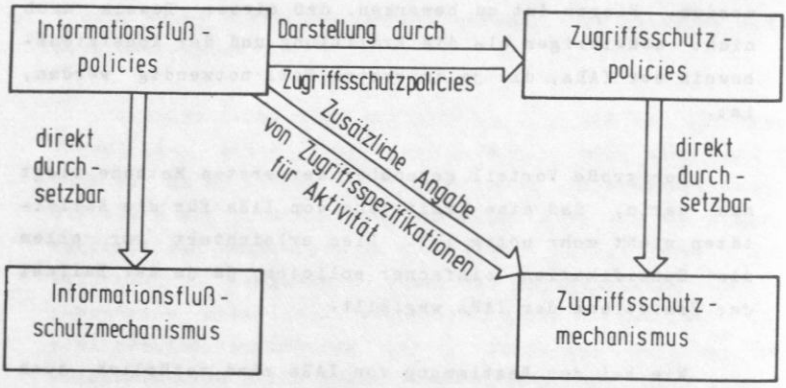


Abb. 5-4: Zur Darstellung von policies

Die obenstehende Abbildung 5-4 zeigt die zwei prinzipiell verschiedenen Arten von policies, nämlich Zugriffsschutzpolicies und Informationsflußpolicies und ihre Durchsetzung durch Schutzmechanismen. Da Informationsfluß-Schutzmechanismen nicht zur Verfügung stehen, muß die OV-Maschine als Zugriffsschutzmechanismus modelliert werden. Für die Durchsetzung von Informationsflußpolicies durch einen Zugriffsschutzmechanismus gibt es dann die zwei dargestellten Möglichkeiten. In dieser Arbeit werden aus den oben genannten Gründen Informationsflußpolicies durch Zugriffsschutzpolicies dargestellt, die von dem Zugriffsschutzmechanismus OV-Maschine direkt durchgesetzt werden können. Die durch die angegebenen Zugriffsschutzpolicies realisierten Informationsflußpolicies werden dann als Zusicherungen formuliert, deren Erfüllung bewiesen werden muß.

An dieser Stelle soll noch kurz auf das Problem der verdeckten Kanäle bei Informationsflußpolicies eingegangen werden (vgl. hierzu <Lam73>, <Jon78b> oder <Rei83> S131ff.). Bei der Zurückführung der Informationsflußpolicies auf Zugriffsschutzpolicies wird hier davon ausgegangen, daß verdeckte Kanäle nicht existieren. Diese Annahme ist insofern gerechtfertigt, da nur die Ebene des formalen Modells bzw. die Spezifikationsebene untersucht wird. Betrachtet man die Implementierung des OV-Mechanismus (vgl. <Rei83>), so müssen verdeckte Kanäle natürlich beachtet werden. Die Implementierung von OV-Mechanismen ist aber nicht Gegenstand dieser Arbeit, so daß hier auf die Betrachtung verdeckter Kanäle verzichtet werden kann.

Hier sei jedoch erwähnt, daß das angegebene formale Modell so flexibel ist, daß die Betrachtung verdeckter Kanäle integrierbar ist. Man muß nämlich beim Übergang von den Informationsflußpolicies zu den Zugriffsschutzpolicies nur zusätzlich die in den Nichtblockierungsbedingungen und Zurückweisungsbedingungen abgefragten Objektkomponenten und den darüber möglichen Informationsfluß berücksichtigen (vgl. <Rei83>).

Nach dieser sehr ausführlichen informalen Beschreibung der Integration von Schutzaspekten in das hier zu entwickelnde Konzept für ADTs und die OV-Maschine sollen nun die entsprechenden formalen Modelle präsentiert werden.

5.4.1.2 Ein formales Modell für ADTs

Ein synchronisierter, geschützter Abstrakter Datentyp (ADT) ist ein Quadrupel $\tau = (\Omega, T, B, \omega_0)$ mit

Ω ist eine Menge, der Wertebereich oder die abstrakte Repräsentation für Objekte des Typs τ.

Man kann Ω noch in $\Omega \subseteq \Omega_V \times \Omega_D$ einteilen, wobei Ω_V die Menge der Verwaltungszustände, Ω_D die Menge der Datenzustände eines Objekts des Typs τ ist.

Eine andere Möglichkeit der Einteilung von Ω, die auch benutzt werden wird, ist $\Omega \subseteq \Omega_{d1} \times \ldots \times \Omega_{dm}$, wobei die di als Unterobjekte bezeichnet werden und Ω_{di} den entsprechenden Wertebereich angibt.

T ist eine endliche Menge von Operationen (Transaktionen) auf Ω mit Ein- und Ausgabeparametern, d.h.

$T := \{t1, \ldots tn\}$ mit $tj \subseteq (Ej \times \Omega) \times (\Omega \times Aj)$ und

$Ej \subseteq \Omega_1^{ej} \times \ldots \times \Omega_{m_{ej}}^{ej}$

$Aj \subseteq \Omega_1^{aj} \times \ldots \times \Omega_{m_{aj}}^{aj}$

Die tj sind partielle und nichtfunktionale Operationen, und die Ej bzw. Aj sind Mengen von Ein- und Ausgabetupeln für die Operation tj, wobei der Wertebereich der einzelnen E/A-Parameter durch $\Omega_{..}^{ej}$ und $\Omega_{..}^{aj}$ gegeben ist.

Eine Aktivität auf ein Objekt des Typs τ ist ein Paar $(t,\mathcal{n})$, wobei gilt $\exists j\ (t = tj \in T \wedge \mathcal{n} \in Ej)$.

Die Menge aller Aktivitäten $\mathcal{A}$ auf ein Objekt des Typs τ ist definiert als

$$\mathcal{A} := \{(t,\mathcal{n}) \mid \exists j\ (t=tj \in T \wedge \mathcal{n} \in Ej)\}$$

B ist ein Tripel von Zugriffs-Beschränkungen
B = (REJ,VTGL,PRIO) mit

$$REJ \subseteq (\mathcal{A} \times \Omega) \times \mathcal{F}$$
$$VTGL \subseteq \mathcal{A} \times \mathcal{P}(\mathcal{A}) \times \Omega$$
$$PRIO \subseteq \mathcal{A} \times \mathcal{A} \times \Omega$$

REJ ist partielle und nichtfunktionale Abbildung von $\mathcal{A} \times \Omega$ nach $\mathcal{F}$, wobei für $\mathcal{F}$ gilt

$$\mathcal{F} = \bigcup_{j=1}^{n} Fj \quad \text{und für alle } j$$

$$Fj \subseteq \Omega_1^{aj} \times \ldots \times \Omega_{maj}^{aj} \quad \wedge \quad Fj \cap Aj = 0$$

d.h. die Fj sind genauso strukturiert wie die Aj, jedoch sind Fj und Aj disjunkt. Fj gibt die Zurückweisungs-Fehlercodes für die Operation tj an.

Die Abbildung REJ (rejection) gibt an, bei welchen Zuständen eine Aktivität zurückgewiesen wird, und welcher Fehlercode in diesem Fall ausgegeben wird. Da die Fehlercodes Fj für jede Operation tj genauso strukturiert sind wie die normalen Ausgabetupel Aj, muß für REJ gelten

$$\forall (tj,\mathcal{n}j)\ \forall \omega\ \forall f\ (tj \in T \wedge \mathcal{n}j \in Ej \wedge \omega \in \Omega \wedge$$
$$f \in \mathcal{F} \wedge (tj,\mathcal{n}j,\omega,f) \in REJ \longrightarrow f \in Fj)$$

VTGL ist die Verträglichkeitsrelation, die angibt, bei welchem Zustand eine Aktivität mit einer Menge von Aktivitäten verträglich ist. Für VTGL mit $x \in \mathcal{A}$, $y \in \mathcal{A}$ und $Y, Y` \subseteq \mathcal{A}$ gilt

(i) $(x,\{y\},\omega) \in VTGL \longleftrightarrow (y,\{x\},\omega) \in VTGL$

(ii) $(x,Y,\omega) \in VTGL \longrightarrow$

$$\forall Y` (Y` \subseteq Y \dashrightarrow (x,Y`,\omega) \in VTGL)$$

PRIO ist die Prioritätsrelation, die die Prioritäten zwischen den Aktivitäten definiert. Für PRIO mit $x, y, z \in \mathcal{A}$ gilt

(i) $(x,x,\omega) \notin PRIO$

(ii) $(x,y,\omega) \in PRIO \longrightarrow (y,x,\omega) \notin PRIO$

(iii) $(x,y,\omega) \in PRIO \wedge (y,z,\omega) \in PRIO \longrightarrow$

$(x,z,\omega) \in PRIO$

ω_o ist die initiale Belegung der Objekte des Typs τ mit $\omega_o \in \Omega$.

Bemerkungen

Bei der Definition des Modells für ADTs wurden die Operationen und die Zugriffs-Beschränkungen für diese Operationen getrennt angegeben, d.h. sowohl Schutz- als auch Synchronisationsaspekte wurden aus den eigentlichen Operationen herausgezogen. Dies trägt vor allem zur leichteren Modifizierbarkeit von ADTs bei, wenn z.B. sich die Zugriffsstrategien bei gleichbleibenden Operationen ändern sollen.

Die Operationen auf Objekte eines ADT`s sind hier als allgemeine relationale Operationen definiert. Diese Eigenschaft ist wichtig, um auf einer Abstraktionsebene von konkreten implementierungstechnischen Details niedrigerer Ebenen und den dazugehörigen Entscheidungen abstrahieren zu können. Der prozeduralen Abstraktion wurde dadurch Rechnung getragen, daß für die Operationen Ein- und Ausgabeparameter definiert wurden. Jeder E/A-Parameter ist dabei wiederum ein abstraktes Objekt (privates Objekt des aufrufenden Prozesses), so daß bei der späteren Einbettung der OV-Maschinen in eine Supervisor-BSM die $\Omega^{ej}_{..}$ und $\Omega^{aj}_{..}$ Wertebereiche von im System definierten ADTs sind. (vgl. 5.4.3)

Der Begriff "Aktivität" wird hier eingeführt als ein Paar, bestehend aus einer Operation und einem zugehörigen Tupel von Eingabewerten. Dies entspricht genau den für die Ausführung der Operation relevanten Größen des Operationsaufrufs. Auf eine explizite Angabe des Namens des aufrufenden Prozesses wurde bewußt bei der Definition des Begriffs verzichtet, um die Formulierung von Synchronisations- und Schutzeinschränkungen unabhängig von den Namen der im System befindlichen Prozesse zu machen. Ein abstraktes Objekt wird also als ein offenes System angesehen, in dessen Umwelt Prozesse existieren, die auf das Objekt zugreifen können. Sind für die Definition von Zugriffseinschränkungen Prozeßnamen erforderlich, müssen diese als Eingabeparameter übergeben werden.

Die Zugriffsbeschränkungen für die möglichen Aktivitäten des ADT`s werden im Modell durch das Tripel (REJ,VTGL,PRIO) beschrieben, wobei REJ - die Zurückweisungsrelation - sich auf die Durchführbarkeit einer Aktivität, VTGL und PRIO auf die gleichzeitige Ausführbarkeit bzw. die Reihenfolge der Ausführung mehrerer Aktivitäten beziehen.

Man beachte, daß in der Zurückweisungsrelation REJ für die einzelnen Fälle der Zurückweisung noch die Ausgabe von Fehlercodes aus einer Menge $\mathcal{F}$ vorgesehen ist. Dies dient einfach zur Differenzierung zwischen verschiedenen Zurück-

weisungsgründen.

Für die Verträglichkeitsrelationen werden hier wie auch in <Mac83> auf Aktivitätenmengen erweiterte Definitionen angegeben. Die Notwendigkeit hierfür wurde bereits in 5.3.4 begründet. Für die Prioritätsrelation wurde aus Gründen der Einfachheit und Übersichtlichkeit die ursprüngliche Definition beibehalten.

Die durch die Zurückweisungsrelation REJ, die die Zugriffsschutzpolicy des ADT`s realisiert, dargestellten Informationsflußpolicies werden an dieser Stelle noch nicht formal eingeführt. Dies geschieht aus verschiedenen Gründen. Zum einen sind sie zur Einführung des formalen Modells der OV-Maschine nicht notwendig, da diese ja nur die Zurückweisungsrelation REJ durchsetzen kann. Ein zweiter Grund ist, daß es sinnvoll ist, diese policies im Rahmen der Definition des Begriffs "Sicherheit" einzuführen. Hierzu ist aber erst die Einführung des formalen Modells der OV-Maschine erforderlich. Im Abschnitt 5.4.4 wird dann ausführlich auf die formale Definition von policies als Zusicherungen und den Sicherheitsbegriff für ADTs eingegangen werden.

Im folgenden sollen nun noch einige zusätzliche Vereinbarungen für die Operationen auf abstrakte Objekte getroffen werden, die jedoch die Allgemeinheit nicht beschränken.

Sei $t = tj \in T$, $\eta \in Ej$ und $\omega \in \Omega$. Man schreibt $\underline{<t(\eta, \omega)>}$, wenn t für (η, ω) definiert ist, d.h. wenn $(\eta, \omega) \in Dom(t)$.

Es bedeutet keine Einschränkung, wenn sowohl der Definitionsbereich der Operationen als auch die Zugriffsbeschränkungen B nur vom Verwaltungszustand des Objekts abhängen, da ja eine Übernahme von Teilen des Datenzustands in den Verwaltungszustand möglich ist. Dies bedeutet auch eine Erleichterung für eine Implementierung eines Schutzmechanis-

mus.

Formal wird definiert:

Eine Operation $tj \in T$ heißt $\underline{\Omega_V\text{-bestimmt}}$, gdw. gilt

$$\forall \mathcal{n} \in Ej \; \forall \omega, \omega` \in \Omega \quad (\omega_V = \omega_{\hat{V}} \longrightarrow$$

$$(<tj(\mathcal{n}, \omega)> \quad <--> \quad <tj(\mathcal{n}, \omega`)>))$$

Das Zugriffsbeschränkungs-Tripel B=(REJ,VTGL,PRIO) heißt

$\underline{\Omega_V\text{-bestimmt}}$, gdw. gilt

$$\forall tj \in T \; \forall \mathcal{n} \in Ej \; \forall \omega, \omega` \in \Omega \quad (\omega_V = \omega_{\hat{V}} \longrightarrow$$

$$((tj, \mathcal{n}, \omega) \in REJ \quad <---> \quad (tj, \mathcal{n}, \omega`) \in REJ))$$

$$\forall x \in \mathcal{O} \; \forall Y \subseteq \mathcal{O} \; \forall \omega, \omega` \in \Omega \quad (\omega_V = \omega_{\hat{V}} \longrightarrow$$

$$((x, Y, \omega) \in^{PRIO} VTGL \quad <---> \quad (x, Y, \omega`) \in^{PRIO} VTGL))$$

Im folgenden seien alle $tj \in T$ und B Ω_V-bestimmt.

Im nächsten Abschnitt soll nun das formale Modell der OV-Maschine als Laufzeit-Objektverwaltungsmechanismus eingeführt werden.

5.4.2 Syntax und Semantik der OV-Maschine

Bevor hier eine formale Definition der OV-Maschine angegeben wird, soll an dieser Stelle eine kurze Übersicht über das Konzept gegeben werden, um die Wirkungsweise zu verdeutlichen:

Schnittstelle zur Außenwelt sind die Operationsaufrufe von Prozessen, die als Eingabe für die OV-Maschine dienen, und die Resultate der Operation, die die Ausgabe darstellen. Es handelt sich hierbei gewissermaßen um "Aufträge" oder "Nachrichten", die die OV-Maschine von außen entgegennimmt und nach der Bearbeitung mit den Ergebnissen wieder nach außen weitergibt. Die OV-Maschine bildet sozusagen eine Schutzhülle um das abstrakte Objekt, d.h. es kann nur über einen Auftrag an die OV-Maschine auf das Objekt zugegriffen werden. Man beachte die direkte Entsprechung zum Konzept der objektorientierten Programmierung (vgl. <Sto83>) mit seinem Nachrichten-Sendemechanismus. Die folgende Skizze soll die Funktionsweise der OV-Maschine verdeutlichen:

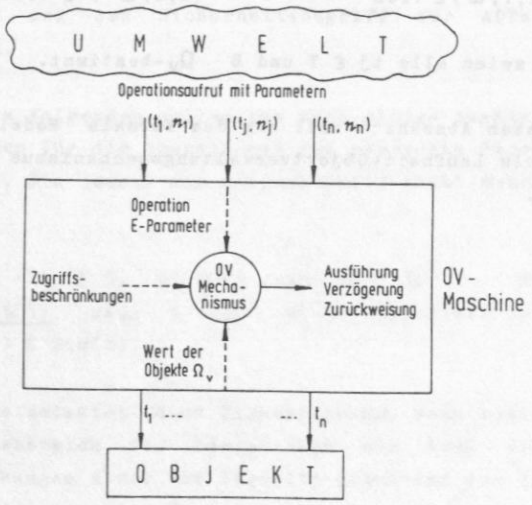


Abb. 5-5: Funktionsweise der OV-Maschine

Beim Aufruf einer Operation auf das Objekt entscheidet der OV-Mechanismus der OV-Maschine aufgrund der Operation, des Werts der Eingabeparameter, der definierten Zugriffsbeschränkungen und des momentanen Objektwerts, ob der Zugriff ausgeführt, verzögert oder zurückgewiesen wird.

Die OV-Maschine wird hier als automatentheoretisches Modell modelliert, jedoch mit unendlichen Zustands-, Input- und Outputmengen. Auf die Kommunikation zwischen der OV-Maschine und der Umwelt, d.h. der übergeordneten Supervisor-BSM, soll jedoch nicht an dieser Stelle, sondern im folgenden Abschnitt 5.4.3 eingegangen werden. Hier interessiert nur die Schnittstelle zwischen den beiden Maschinen, d.h. zum einen der Operationsaufruf und zum anderen die Antwort darauf. Neben der Übergabe der gewünschten Aktivität, d.h. von Operation mit Eingabeparametern, wird von der Supervisor-BSM noch eine eindeutige Auftragsidentifikation mit übergeben, die, ohne in der OV-Maschine interpretiert zu werden, wieder mit an die Supervisor-BSM zurückgegeben wird.

Die OV-Maschine kennt drei verschiedene Aktionen, den Aufruf, das Starten und das Beenden einer Aktivität. Analog zur BSM gibt es verschiedene Tätigkeitszustände für laufende, bereite, wartende und blockierte Aktivitäten. Die Syntax der OV-Maschine wird dann wie folgt definiert:

Gegeben sei ein ADT $\tau = (\Omega, T, B, \omega_o)$. Eine OV-Maschine für Objekte des Typs τ ist ein 6-Tupel
$\mathcal{M} := (Q, Ein, Aus, do, out, q_o)$ mit

Q ist eine i.a. nicht endliche Menge, Zustandsmenge genannt.

$$Q \subseteq \Omega \times H \subseteq \Omega \times H_{blo} \times H_{la} \times H_{be} \times H_{wa}$$

Wie in der BSM bezeichnet H die Menge der Tätigkeitszustände der OV-Maschine. Die $H_{..}$ sind Teilmengen der Menge $\mathcal{P}(I \times \mathcal{A})$, wobei I eine i.a. unendliche Menge

von Identifikatoren ist.

Ein Zustand $q \in Q$ wird mit
$q = (\omega, h\,blo, h\,la, h\,be, h\,wa)$ angegeben, und für alle $q \in Q$ sind die h.. paarweise disjunkt.

Ein ist die i.a. nicht endliche Eingabemenge

$$Ein := \{call, start, end\} \times I \times \mathcal{A}$$

wobei call, start, end die drei verschiedenen Aktionen der OV-Maschine darstellen.

Aus ist die i.a. nicht endliche Ausgabemenge

$$Aus := (I \times \mathcal{A} \times (\bigcup_{j=1}^{n} Aj \cup \mathcal{F}))^*$$

Die Vereinigung der Aj ist hier die normale Ausgabe der Operationen aus T, $\mathcal{F}$ die Menge der Zurückweisungs-Fehlercodes und der "*" steht für die Möglichkeit der leeren Ausgabe bzw. der Ausgabe von mehreren Antworten gleichzeitig (vgl. Semantik)

do ist die Zustandsübergangsrelation $do : Q \times Ein \dashrightarrow Q$

out ist die Ausgaberelation $out : Q \times Ein \dashrightarrow Aus$

qo ist der Anfangszustand $qo \in Q$ mit $qo = (\omega_o, 0, 0, 0, 0)$

Obwohl do und out i.a. Relationen sind, wird hier eine funktionale Schreibweise verwendet, um die Darstellung übersichtlicher zu gestalten.

Analog zur BSM muß nun definiert werden, wann eine Aktivität aus $\mathcal{A}$ bzw. "erweiterte" Aktivität aus $I \times \mathcal{A}$ ausführbar, blockiert, wartend, bereit oder laufend ist.

Definitionen

Sei $q = (\omega, h\,blo, h\,la, h\,be, h\,wa)$ und $(id,t,\mathcal{N}) \in I \times \mathcal{OL}$.

- $(id,t,\mathcal{N})$ heißt im Zustand q ausführbar, in Zeichen $ausf(id,t,\mathcal{N},q)$, wenn gilt $<t(\mathcal{N},\omega)>$.

- $(id,t,\mathcal{N})$ heißt im Zustand q blockiert, in Zeichen $(id,t,\mathcal{N}) \in h\,blo(q)$, wenn gilt $\neg(ausf(id,t,\mathcal{N},q))$.

- $(id,t,\mathcal{N})$ heißt im Zustand q bereit, in Zeichen $(id,t,\mathcal{N}) \in h\,be(q)$, wenn gilt

$$(id,t,\mathcal{N}) \notin h\,blo(q) \wedge (id,t,\mathcal{N}) \notin h\,la(q) \wedge ((t,\mathcal{N}), \mathcal{OL}_{hla}(q), \omega) \in VTGL$$

wobei $\mathcal{OL}_{hla}(q) := \{(t`,\mathcal{N}`) \mid \exists\, id\ (id,t`,\mathcal{N}`) \in h\,la(q)\}$

- $(id,t,\mathcal{N})$ heißt im Zustand q wartend, in Zeichen $(id,t,\mathcal{N}) \in h\,wa(q)$, wenn gilt

$$(id,t,\mathcal{N}) \notin h\,blo(q) \wedge (id,t,\mathcal{N}) \notin h\,la(q) \wedge (id,t,\mathcal{N}) \notin h\,be(q)$$

Die Menge der laufenden Aktivitäten der OV-Maschine im Zustand q, h la(q) wird durch die Dynamik der Maschine, d.h. durch die ausführbaren Aktionen der OV-Maschine bestimmt. Dies bedeutet eine Festlegung der Zustandsübergangsrelation do und der Ausgaberelation out bei den verschiedenen möglichen Eingaben "call", "start" und "end".

Aufruf einer Aktivität

Sei $q = (\omega, h\,blo, h\,la, h\,be, h\,wa)$ und $(id,t,\pi) \in I \times \mathcal{A}$. Die Eingabe in die OV-Maschine sei $(call,id,t,\pi)$, wobei $(id,t,\pi) \notin h\,blo \cup h\,la \cup h\,be \cup h\,wa$.

(i) falls $\exists f\,(f \in \mathcal{F} \wedge (t,\pi,\omega,f) \in REJ)$ gilt

$do(q,(call,id,t,\pi)) = q$ und
$out(q,(call,id,t,\pi)) = (id,t,\pi,f)$

(ii) sonst gilt

$do(q,(call,id,t,\pi)) = (\omega, h\,blo', h\,la, h\,be', h\,wa')$,

wobei durch die obigen Definitionen bestimmt wird, ob (id,t,π) in $h\,blo'$, $h\,be'$ oder $h\,wa'$ eingereiht wird.

$out(q,(call,id,t,\pi)) = \lambda$

Starten einer Aktivität

Sei $q = (\omega, h\,blo, h\,la, h\,be, h\,wa)$ und $(id,t,\pi) \in I \times \mathcal{A}$. Die Eingabe in die OV-Maschine sei $(start,id,t,\pi)$, wobei gilt

$$(id,t,\pi) \in h\,be \wedge \neg\exists (id',t',\pi')\,((id',t',\pi') \in h\,be \wedge ((t',\pi'),(t,\pi),\omega) \in PRIO)$$

$do(q,(start,id,t,\pi)) = (\omega, h\,blo, h\,la', h\,be', h\,wa')$, wobei $h\,la' = h\,la \cup \{(id,t,\pi)\}$ und die Mengen $h\,be'$ und $h\,wa'$ sich aus $h\,be - \{(id,t,\pi)\}$ und $h\,wa$ gemäß obigen Definitionen ergeben.

$out(q,(start,id,t,\pi)) = \lambda$

Beenden einer Aktivität

Sei q = (ω,h blo,h la,h be,h wa) und (id,t,π) ε I x $\mathcal{A}$.
Die Eingabe in die OV-Maschine sei (end,id,t,π), wobei (id,t,π) ε h la.

Sei (ω`,$\mathcal{A}$) ε t(π,ω)

do(q,(end,id,t,π)) = (ω`,h blo`,h la`,h be`,h wa`), wobei
h la` = h la - {(id,t,π)}.

Sei $\hat{h}$:= h blo $\cup$ h be $\cup$ h wa und
h rej := {(id`,t`,π`) | (id`,t`,$\pi$`) ε $\hat{h}$ $\wedge$ $\exists$f (f ε $\mathcal{F}$ $\wedge$ ((t`,$\pi$`,ω`,f) ε REJ)}

Die Mengen h blo`, h be` und h wa` ergeben sich aus $\hat{h}$ - h rej durch Anwendung der obigen Definitionen

out(q,(end,id,t,π)) = (id,t,π,$\mathcal{A}$) o (id1,t1,π1,f1) o ...
... o (idz,tz,πz,fz), wobei

$\bigcup_{i=1}^{z}$ (idi,ti,πi,fi) = h rej, "o" die Konkatenation ist, und

$\forall$i (1 $\leq$ i $\leq$ z ---> (ti,πi,ω`,fi) ε REJ)

Bemerkung

Auf der Ebene der OV-Maschine spielen Prozesse keine Rolle, da von der Supervisor-BSM nur Aktivitäten zur Ausführung übergeben werden. Falls benötigt, wird die Prozeßidentifikation als Eingabeparameter mit übergeben.

Beim Aufruf einer Aktivität wird von der Supervisor-BSM neben der Aktivität noch eine eindeutige Identifikation übergeben. Diese eindeutige Identifikation bewirkt, daß in der OV-Maschine gleiche Aktivitäten durch diese Erweiterung unterschieden werden können. Falls die aufgerufene Akti-

vität beim momentanen Zustand zurückgewiesen werden muß, bleibt der Zustand der OV-Maschine unverändert, und eine entsprechende Fehlermeldung wird ausgegeben. Ansonsten wird die erweiterte Aktivität in einen der Zustände h blo, h be oder h wa eingereiht, je nachdem, ob die Aktivität zum momentanen Zustand ausführbar ist, bzw. mit der Menge der laufenden Aktivitäten verträglich ist. In diesem Fall erfolgt natürlich keine Ausgabe.

Eine bereite Aktivität kann gestartet werden, wenn keine höherpriore Aktivität bereit ist. Die erweiterte Aktivität wird "laufend" gesetzt, und alle bereiten Aktivitäten, die mit der neuen Menge dar laufenden Aktivitäten nicht verträglich sind, in den Zustand "wartend" versetzt. Eine Ausgabe erfolgt nicht.

Eine laufende Aktivität kann beendet werden, wobei sich der Datenzustand gemäß der ausgeführten Operation und den Eingabeparametern ändert. Die Aktivität wird aus dem Zustand "laufend" genommen und alle nichtlaufenden Aktivitäten, die aufgrund des veränderten Datenzustands zurückgewiesen werden müssen, werden aus den entsprechenden Tätigkeitszuständen genommen. Als Ausgabe ergibt sich dann die Antwort auf die beendete Aktivität und die Fehlermeldungen aller zurückgewiesenen Aktivitäten.

Die wesentlichsten Unterschiede zur BSM bestehen in folgenden Punkten:

- Die OV-Maschine ist ein offenes System in dem Sinn, daß sie mit der Umwelt kommuniziert, d.h. Aufträge entgegennimmt und die bearbeiteten Aufträge wieder ausgibt, während die BSM keine Schnittstellen nach außen besitzt und nur interne Zustandsübergänge durchführen kann.

- Die OV-Maschine kennt nur Aktivitäten, die ihr zur Bearbeitung übergeben wurden, und keine aktiven Einheiten wie die Prozesse in der BSM. Sie ist also - als Modell für die Verwaltung der Zugriffe auf abstrakte Objekte - ein passives System.

- Im Gegensatz zur BSM, wo ja die Schutzmaßnahmen implizit in den Operationen verborgen waren, realisiert die OV-Maschine den Schutz durch die Zurückweisung nicht erlaubter Aktivitäten und die Ausgabe eines entsprechenden Fehlercodes.

- Während die BSM auf einem unstrukturiertem Datenbereich aufbaut und Operationen als Übergänge in diesem Zustandsraum betrachtet, sind in der OV-Maschine bereits im formalen Modell Abstraktionsmöglichkeiten berücksichtigt. So sind die Operationen als Prozeduren mit Ein- und Ausgabeparametern definiert, und die Datenabstraktion ergibt sich dadurch, daß der Datenbereich des Gesamtsystems aus abstrakten ADT-Objekten besteht, wobei für jedes dieser Objekte eine OV-Maschine zur Verwaltung der Zugriffe definiert ist.

Im folgenden soll nun der Berechnungsbegriff analog zur BSM definiert werden.

Berechnung der OV-Maschine

Eine Berechnung der OV-Maschine ist eine Folge $\langle (qi,ei,ai) \mid 0 \leq i \leq m \rangle$ von Tripeln mit $m \in N \cup \{\infty\}$, wobei gilt

(i) $\forall i \, (0 \leq i \leq m \quad \longrightarrow \quad qi \in Q \wedge ei \in Ein \wedge ai \in Aus)$

(ii) $\forall i\ (0 \leq i \leq m-1 \quad \rightarrow$

$q_{i+1} = do(q_i, e_i) \wedge a_i = out(q_i, e_i))$

In den beiden folgenden Abschnitten soll nun zunächst eine Einbettung von OV-Maschinen in eine übergeordnete Supervisor-BSM diskutiert werden und dann policies und der Sicherheitsbegriff für OV-Maschinen eingeführt werden.

5.4.3 Einbettung von OV-Maschinen in eine Supervisor-BSM

Als Datenabstraktionsmechanismus wurde in die ursprünglich unstrukturierte BSM das ADT-Konzept und für die Verwaltung der Objekte eines ADT`s die OV-Maschine eingeführt. Die OV-Maschine modelliert das Systemverhalten beim Zugriff auf ein abstraktes Objekt, ist also als lokaler Verwaltungsmechanismus zu sehen.

Ein formales Modell für ein Gesamtsystem muß neben den Objekten mit den zugehörigen OV-Maschinen als passive Einheiten noch Prozesse oder Subjekte als aktive Einheiten beinhalten. Die Prozesse greifen über die definierten Operationen auf die abstrakten Objekte des Systems zu. Damit diese Zugriffe kontrolliert ablaufen, muß eine übergeordnete Instanz - Supervisor-BSM (SBSM) - für die Kommunikation mit den jeweiligen OV-Maschinen sorgen. Die Schnittstellen hierfür wurden bereits im letzten Abschnitt eingeführt, es sind von der SBSM zur OV-Maschine der Aufruf einer Aktivität (call,id,t,π) und in der umgekehrten Richtung die Antwort der OV-Maschine (id,t,π,α) bzw. (id,t,π,f).

Die Aktionen der SBSM und der einzelnen OV-Maschinen laufen dabei völlig asynchron ab. Eine mögliche Realisierung könnte über einen Supervisor-Prozeß und mehrere OV-Prozesse innerhalb des Betriebssystems erfolgen, die asynchron arbeiten. Ebenso könnte es sich auch um mehrere Prozessoren handeln, so daß auch verteilte Systeme direkt modelliert werden können. Der Übergabebereich zwischen SBSM und OV-Maschinen könnte dann aus Warteschlangen bestehen oder durch ein Kommunikationssystem in einem Netzwerk ersetzt werden. Man vergleiche hierzu das vorliegende Modell mit dem Konzept der objektorientierten Programmierung (<Sto83>). Auf die Realisierung der notwendigen Mechanismen soll in dieser Arbeit jedoch nicht näher eingegangen werden, da hier mehr die methodischen Aspekte im Vordergrund stehen.

Zusätzlich zu den gemeinsamen ADT-Objekten mit den entsprechenden OV-Maschinen besitzen die einzelnen Prozesse noch private Objekte. Diese seien aus Gründen der Datenabstraktion ebenfalls ADT-Objekte, jedoch ist hier eine Überwachung der Zugriffe durch OV-Maschinen nicht notwendig.

Im folgenden wird nun die SBSM und die Einbettung von OV-Maschinen formal präsentiert, wobei aber an einigen Stellen auf eine streng formale Darstellung zugunsten der Übersichtlichkeit verzichtet wurde.

Prozeßdefinition

Ausgehend von der Prozeßdefinition P = (A,D,R,ao,Do) der BSM wird hier auch für Prozesse prozedurale und Datenabstraktion eingeführt, d.h.

- Die unstrukturierte Variablenmenge Var wird ersetzt durch eine Menge von privaten ADT-Objekten O = {O1, ... ,Om}.

- Die unstrukturierte Datenmenge D wird ersetzt durch den strukturierten Wertebereich der privaten ADT-Objekte des Prozesses $\Omega = \Omega 1 \times \ldots \times \Omega m$.

- Die Prozeßrelation R bzw. die Übergänge σij im Zustandsraum werden ersetzt durch eine Menge Σ von Operationsaufruftripeln (ak,t(o,φ,ψ),al) mit ak,al $\in$ A

  o ist der Name eines privaten oder globalen Objekts vom Typ τ

  t ist eine für Objekte des Typs τ definierte Operation

  φ,ψ sind Tupel von Namen privater Objekte des Prozesses, wobei die Anzahl und Typen der Objekte abhängig von t sind.

Ein Prozeß in der SBSM läßt sich somit als Tupel
$P = (A, O, \Omega, \Sigma, ao, \Omega o)$ darstellen.

Syntax der SBSM

Eine Supervisor-Betriebssystemmaschine (SBSM) ist definiert als Tupel $SBSM = (\overline{P}, \mathcal{T}, \overline{O}, OV, QE, R)$, wobei gilt

$\overline{P}$ ist eine endliche, nichtleere Menge $\overline{P} = \{P1, \ldots, Pn\}$ von sequentiellen Prozessen, deren Adreßmengen paarweise disjunkt sind, d.h.

$$\forall k,l\ (1 \leq k,l \leq n \wedge k \neq l \longrightarrow Ak \cap Al = O).$$

$\mathcal{T}$ ist eine endliche, nichtleere Menge $\mathcal{T} = \{\tau 1, \ldots, \tau p\}$ von Abstrakten Datentypen

$\overline{O}$ ist eine endliche, nichtleere Menge $\overline{O} = \{O1, \ldots, Or\}$ von globalen Objekten. τi bezeichne den Typ des Objekts Oi, ωi dessen Wert. Es gilt $\tau i \in \mathcal{T}$.

OV ist eine endliche, nichtleere Menge $OV = \{\mathfrak{M}1, \ldots, \mathfrak{M}r\}$ von OV-Maschinen, wobei $\mathfrak{M}i$ die dem globalen Objekt Oi zugeordnete Maschine ist.

QE ist die Zustandsmenge der SBSM mit

$$QE = \prod_{k=1}^{n} Ak \times \Omega k \times H\,blo \times H\,la \times H\,be \times Q1 \times \ldots \times Qr$$

Ein Zustand qe der SBSM ist somit definiert durch die momentanen Adressen der Prozesse aus $\overline{P}$, die momentanen Werte der privaten Objekte der Prozesse aus $\overline{P}$ (lokale Prozeßzustände), die Tätigkeitszustände derjenigen Prozesse, die momentan auf private Objekte zugreifen und die aktuellen Zustände der OV-Maschinen.

Bezeichne $\mathcal{A}$ jetzt die Menge aller Aktivitäten im Gesamtsystem, d.h. $\mathcal{A} := \bigcup_{j=1}^{p} \mathcal{A}_j$, dann sind die H.. Teilmengen der Menge $\mathcal{P}(\bar{P} \times (\bar{O} \cup \bigcup_{k=1}^{n} O_k) \times \mathcal{A})$

R mit $R \subseteq QE \times QE$ ist die Zustandsübergangsrelation der SBSM

Bemerkungen

Es wird davon ausgegangen, daß innerhalb der SBSM sämtliche Objektnamen verschieden sind, so daß eine eindeutige Objektidentifizierung möglich ist. Dies entspricht der Voraussetzung bei der BSM, daß die Variablenbezeichnungen global sind. Ohne diese Voraussetzung, die die Allgemeinheit des Ansatzes jedoch nicht einschränkt, müßte eine Variablenidentifikationsfunktion eingeführt werden, die dem Modell viel an Übersichtlichkeit nehmen würde.

Die Tätigkeitszustände der Prozesse beinhalten bei der SBSM den Prozeßnamen, die auszuführende Aktivität und das private oder globale Objekt, auf das die Aktivität ausgeübt wird, bzw. ausgeübt werden soll.

Der Anfangsdatenzustand braucht bei der SBSM nicht explizit angegeben zu werden, da er für jedes Objekt bereits durch die entsprechende ADT-Definition gegeben ist.

Semantik der SBSM

Die Interpretation der globalen Tätigkeitszustände erfolgt hier etwas anders als bei der BSM oder den OV-Maschinen, da in der SBSM das konkurrierende Verhalten der Prozesse um gemeinsame Objekte nicht modelliert werden muß. Hierfür wurde ja das Konzept der OV-Maschine eingeführt.

So gibt es in der SBSM nur die drei Tätigkeitszustände "laufend", "blockiert" und "bereit". Der Zustand "wartend" existiert nicht, weil Operationen auf private Objekte eines Prozesses mit allen anderen laufenden Operationen (der anderen Prozesse!) verträglich sind.

Wird ein Prozeß in den Zustand "blockiert" der SBSM versetzt, d.h. blockiert er sich beim Aufruf einer Operation auf ein privates Objekt, so handelt es sich um einen Fehler, da der Prozeß in diesem Zustand nie mehr fortgesetzt werden kann.

Im Zustand "laufend" befinden sich alle Prozesse, die momentan eine Operation auf private Objekte ausführen.

Der Zustand "bereit" beinhaltet alle Prozesse,

- deren nächste Operation eine Operation auf ein globales Objekt ist, wobei diese Operation noch nicht aufgerufen wurde, oder
- deren nächste Operation eine Operation auf ein privates Objekt ist, wobei diese Operation zum momentanen Zustand ausführbar ist.

Definitionen

Sei $qe = (\alpha 1, \omega 1, \ldots, \alpha n, \omega n, h\,blo, h\,la, h\,be, q1, \ldots, qr)$ und $\pi(qe,k) = (\alpha k, \omega k)$ die Projektion von qe auf den Prozeß Pk.

$$Op(qe,k) := \{(ai, t(o, \varphi, \psi), aj) \mid (\ldots) \in \Sigma^{(k)} \wedge \pi(qe,k) = (\alpha k, \omega k) \wedge \alpha k = ai\}$$

definiert die im Zustand qe von der aktuellen Adresse des Prozesses Pk ausgehenden Operationsaufrufe.

Ein Operationsaufruf (ai,t(o,φ,ψ),aj) des Prozesses Pk heißt im Zustand qe zulässig, wenn gilt

(ai,t(o,φ,ψ),aj) ε Op(qe,k)

Sei Pk ein Prozeß, o privates Objekt vom Typ τ des Prozesses Pk und (t,π) eine Aktivität auf Objekte des Typs τ. Der Wert des Objekts im Zustand qe sei ω. Die Aktivität (t,π) heißt im Zustand qe ausführbar, in Zeichen ausf(Pk,o,t,π,qe), wenn gilt <t(π,ω)>.

Das dynamische Verhalten der SBSM wird dann durch die folgenden Aktionen festgelegt:

Initialisierung

Nach der Initialisierung, d.h. im Zustand qeo gilt:

- Jeder Prozeß Pk befindet sich im lokalen Prozeßzustand (ako, Ωko).

- h la(qeo) = 0

- Alle Anfangszustände der OV-Maschinen $\mathfrak{M}$ i seien qio.

- Für jeden Prozeß Pk existiere ein zulässiger Operationsaufruf (ako,t(o,φ,ψ),ai). Sei π der Vektor der aktuellen Werte der in φ aufgeführten privaten Objekte des Prozesses Pk.

(i) falls o privates Objekt von Pk und
ausf(Pk,o,t,π,qeo)
so gilt (Pk,o,t,π) ε h be(qeo)
sonst gilt (Pk,o,t,π) ε h blo(qeo)

(ii) falls o globales Objekt ist, gilt
(Pk,o,t,π) ε h be(qeo)

Starten einer Aktivität

Sei qe = (α1, ω1, ... , αn, ωn, h blo, h la, h be, q1, ... , qr)
und qe` = ( $\alpha$1`, ω1`, ... , $\alpha$n`, ωn`, h blo`, h la`, h be`, q1`, ... , qr`)

Start(Pk,o,t,π)
Ein Übergang qe ----------> qe` ist genau dann möglich, d.h. (qe,qe`) ε R, wenn gilt (Pk,o,t,π) ε h be.

Für den Folgezustand qe` lassen sich zwei Fälle unterscheiden:

(i) o ist privates Objekt des Prozesses Pk

h be` = h be - {(Pk,o,t,π)}
h la` = h la $\cup$ {(Pk,o,t,π)}

Alle anderen Komponenten bleiben unverändert.

(ii) o ist globales Objekt

Es wird eine Eingabe (start,id,t,π) für die entsprechende OV-Maschine erzeugt und

h be` = h be - {(Pk,o,t,π)}

Alle anderen Komponenten bleiben unverändert.

Die Identifikation id des Auftrags an die OV-Maschine ergibt

sich durch eine eineindeutige Identifikationsfunktion
Ident ; $\overline{P} \times \overline{O}$ ---> I aus dem Prozeß Pk und dem Objekt o.

Aufgrund der Eingabe (start,id,t,π) verändert anschließend die OV-Maschine ihren internen Zustand bzw. gibt eine entsprechende Fehlerausgabe.

Beenden einer Aktivität auf ein privates Objekt

Sei qe = ($\alpha 1, \omega 1, \ldots, \alpha n, \omega n, h\,blo, h\,la, h\,be, q1, \ldots, qr$)
und qe` = ( $\alpha 1`, \omega 1`, \ldots, \alpha n`, \omega n`, h\,blo`, h\,la`, h\,be`, q1`, \ldots, qr`$)

End(Pk,o,t,π)
Ein Übergang qe ----------> qe` ist genau dann möglich, d.h. (qe,qe`) ε R, wenn gilt (Pk,o,t,π) ε h la.

Für den Folgezustand qe` gilt dann:

(i) $\forall$ l (l $\neq$ k ---> α`l = $\alpha$l $\wedge$ $\omega$`l = ωl)

(ii) $\forall$ i (1 $\leq$ i $\leq$ r ---> qi` = qi)

(iii) Sei (ai,t(o,φ,ψ),aj) der beendete Operationsaufruf und (ω`,α) ε t(π,ω) das Ergebnis der Aktivität.

Den in ψ aufgeführten privaten Objekten werden die Werte des Ergebnisvektors α zugeordnet und dem Objekt o der Wert ω`. Alle anderen Objekte des Prozesses verändern sich in $\omega$`k gegenüber ωk nicht.

Die Adresse des Prozesses verändert sich nach aj, d.h. α`k = aj.

(iv) h la` = h la - {(Pk,o,t,π)}

(v) Es wird ein zulässiger Operationsaufruf (aj,t`(o`,φ`,$\psi$`),al) des Prozesses Pk ausgewählt, und wie bei der Initialisierung der SBSM erläutert (Pk,o`,t`,π`) in h be` oder h blo` eingereiht.

Empfang einer Ausgabe von $\mathfrak{M}$i

Sei qe = (α1,ω1, ... ,αn,ωn,h blo,h la,h be,q1, ... ,qr)
und qe`= ($\alpha$1`,ω1`, ... ,$\alpha$n`,ωn`,h blo`,h la`,h be`,q1`, ... ,qr`)

Ausg(Pk,o,t,π)
Ein Übergang qe ----------> qe` ist genau dann möglich, d.h. (qe,qe`) $\in$ R, wenn gilt

(id,t,π,α) oder (id,t,π,f) ist Ausgabe der OV-Maschine $\mathfrak{M}$i an die SBSM und ident^{-1}(id) = (Pk,o).

Für den Folgezustand qe` gilt dann das unter "Beenden einer Aktivität auf private Objekte" gesagte, mit der Ausnahme, daß der Wert des globalen Objekts o schon innerhalb $\mathfrak{M}$i verändert wurde und daß h la` = h la.

Berechnung der SBSM

Eine Berechnung der SBSM ist eine Folge $\langle qei \mid 0 \leq i \leq m \rangle$ mit $m \in N \cup \{\infty\}$, wenn gilt:

(i) Für qeo gelten die angegebenen Initialisierungsbedingungen

(ii) $\forall i\ (0 \leq i \leq m)$ gilt

$$qei \xrightarrow{Start(Pk,o,t,\pi)} qei+1 \quad \text{oder}$$

$$qei \xrightarrow{End(Pk,o,t,\pi)} qei+1 \quad \text{oder}$$

$$qei \xrightarrow{Ausg(Pk,o,t,\pi)} qei+1.$$

5.4.4 Policies und Sicherheitsbegriff für ADTs

In diesem Abschnitt wird der für die Zuverlässigkeit eines ADT`s grundlegende Begriff der Sicherheit eingeführt. Für die Sicherheit eines ADT`s sind dabei zwei Gesichtspunkte maßgebend, die Konsistenz der angegebenen Verträglichkeitsrelationen und die Einhaltung gewisser Sicherheitszusicherungen, der sogenannten policies.

Zuerst sollen hier Möglichkeiten zur Definition von policies für ADTs bzw. OV-Maschinen angegeben werden und anschließend der Begriff der Konsistenz von Verträglichkeitsrelationen definiert werden. Aufbauend darauf kann dann der Sicherheitsbegriff für ADTs eingeführt werden. Zum Schluß dieses Abschnitts sollen schließlich noch kurz systemweite Zusicherungen, d.h. Eigenschaften, die sich auf eine SBSM mit ADTs als Systemkomponenten beziehen, präsentiert werden.

5.4.4.1 Die Definition von policies

Wie bereits ausführlich dargestellt, spielt der Begriff der policy und die Trennung zwischen policy und Schutzmechanismus eine wichtige Rolle für die Sicherheit und Zuverlässigkeit geschützter Systeme. Die meisten existierenden Schutzmodelle bieten allerdings nur sehr eingeschränkte Möglichkeiten zur Definition von policies, häufig wird sogar von einer festen policy - wie z.B. militärische Sicherheit - ausgegangen (vgl. 4.5). Eine Ausnahme bilden hier nur die Modelle von Jones und Lipton (4.3.2), Goguen und Meseguer (4.4.1) und Stoughton (4.4.2). Die Definition von policies in diesen Modellen sei deshalb hier nochmals kurz dargestellt.

Jones und Lipton unterscheiden in ihrem Modell klar zwischen policy und Mechanismus. Sie betrachten Informationsflußpolicies für Programme (bzw. Operationen),

die festlegen, auf welche Daten ein Programm lesend zugreifen darf, wobei eine wertabhängige Definition (content dependent policy) möglich ist. Die Aufgabe des Schutzmechanismus ist dann, die gegebenen policies durchzusetzen; dies wird als Zuverlässigkeit des Mechanismus bezeichnet. Jones und Lipton beschränken sich bei ihren Untersuchungen auf Informationsflußpolicies für jeweils eine Operation. Sie betrachten weder Fragen der Integrität von Daten noch den Informationsfluß, der aus Zugriffen mehrerer Operationen auf gemeinsame Daten resultieren kann.

Aufbauend auf einer abstrakten Maschine werden im Modell von Goguen und Meseguer policies als Zusicherungen über die gegenseitige Nichtbeeinflußbarkeit von Subjekten bzw. Operationen definiert. Das Modell ist nicht objektorientiert und ermöglicht auch nur globale, keine lokalen, objektorientierten Schutzentscheidungen. Die Formulierung von content dependent policies ist nicht möglich. Der Schutzmechanismus des Modells ist einfach durch die abstrakte Maschine mit den gegebenen Operationen definiert, d.h. der Schutzmechanismus ist hier in den einzelnen Operationen verborgen. Durch die Trennung zwischen policy und Mechanismus (abstrakte Maschine) ist ein klarer Sicherheitsbegriff definierbar, wobei jedoch auf die Verifikation von Sicherheit von den Autoren nicht eingegangen wird.

Grundlage des Gesamtsystems im Modell von Stoughton sind dessen Objekte. Zu jedem Objekt lassen sich zwei policies definieren. Die "current accesses" beschreiben, welche Operationen jedes Subjekt auf das Objekt anwenden darf, sie sind also die Zugriffskontrollkomponente des Objekts. Die "potential accesses" geben an, welche Operationen jedes Subjekt auf die im Objekt enthaltene Information anwenden darf, sie sind also die Informationsfluß-Kontrollkomponente des Objekts. Für die Operationen des Systems definiert der Autor dann die "Legalitätseigenschaft", d.h. die Operationen halten die durch die Objekte gegebenen policies ein. Auch hier ist also der Schutzmechanismus in den Operationen bzw. in der abstrakten Maschine verborgen, während die Zu-

griffseinschränkungen lokal am Objekt definiert sind. Eine Definition wertabhängiger policies ist nicht vorgesehen.

Im vorliegenden Modell werden policies in der Form von Zusicherungen bzw. Invarianzeigenschaften definiert, wobei man zwischen globalen (systemweiten) und lokalen (objektbezogenen) policies differenzieren kann. Da sich globale Zusicherungen auf lokale zurückführen lassen, und der Schwerpunkt der Arbeit auf der Definition und Verwaltung der Zugriffe auf synchronisierte, geschützte ADT-Objekte liegt, werden lokale policies detailliert betrachtet, während auf die Definition globaler policies mit Hilfe von lokalen policies nur kurz eingegangen werden soll.

Zur Darstellung der möglichen Zugriffe auf ADT-Objekte wurde der Begriff der Aktivität eingeführt, so daß hier policies als Prädikate über Aktivitäten oder über Beziehungen zwischen Aktivitäten definiert werden. Wie bei den Zugriffseinschränkungen lassen sich policies, die nur eine Aktivität betreffen, und solche, die die wechselseitigen Beziehungen zwischen zwei oder mehreren Aktivitäten betreffen, unterscheiden. Natürlich muß der zu präsentierende Ansatz zur Definition von policies so umfassend sein, daß er sowohl die Formulierung von Zugriffsschutzpolicies, als auch Informationsflußpolicies erlaubt, und auch eine wertabhängige (content dependent) Formulierung ermöglicht.

Lokale, objektorientierte policies sind alleine durch Prädikate über Aktivitäten - ohne Verwendung von Prozeßnamen - formulierbar. Sollen Prozeßnamen für die Schutzentscheidungen, und damit auch für die policies, trotzdem relevant sein, so müssen sie, wie bereits dargestellt, als Eingabeparameter definiert werden. Damit können dann natürlich policies auch abhängig von Prozeßnamen formuliert werden.

Wie bereits dargelegt, müssen im vorliegenden Modell policies als Zusicherungen über Aktivitäten formuliert werden, deren Einhaltung durch die möglichen Zugriffe auf das Objekt sowie die definierten Zugriffseinschränkungen

verifiziert werden muß. Es ist also nicht möglich, die gewünschten policies einfach der OV-Maschine vorzugeben, und einen Mechanismus in die OV-Maschine zu integrieren, der die gegebenen policies automatisch durchsetzt. Dies wurde in 5.4.1.1 ausführlich begründet.

Im folgenden werden nun zunächst die policies für eine Aktivität und anschließend die für Beziehungen zwischen Aktivitäten formal eingeführt. Da die Zusicherungen im allgemeinen die Nicht-Existenz von Zugriffsmöglichkeiten bzw. Informationsflüssen beinhalten werden, wird dies in der formalen Definition entsprechend vorgesehen.

Definition von policies für eine Aktivität

Policies, die nur eine Aktivität betreffen, lassen sich einteilen in reine Zugriffsschutzpolicies und einfache Informationsflußpolicies, die angeben, ob bestimmte Unterobjekte bei der Ausführung einer Aktivität abgefragt oder verändert werden dürfen.

Sei $\tau = (\Omega, T, B, \omega_0)$ ein ADT mit Aktivitätenmenge $\mathcal{A}$.

Eine Nichtzugriffspolicy für τ ist eine Menge NACC $\subseteq \mathcal{A} \times \Omega$, d.h. NACC besteht aus einer Menge von Tripeln (t, α, ω).

Die Nichtzugriffspolicy NACC gibt an, bei welchen Objektzuständen mit einer Aktivität nicht auf das ADT-Objekt zugegriffen werden darf, d.h. für alle $(t, \alpha, \omega) \in$ NACC darf mit der Aktivität (t, α) im Zustand ω nicht zugegriffen werden.

Für die Definition von Sicherheit ist es wichtig, festzulegen, wann eine policy vom durch den entsprechenden ADT definierten Mechanismus erfüllt wird. Für die Nichtzugriffspolicy NACC bedeutet dies:

Sei τ ein ADT und NACC $\subseteq \mathcal{A} \times \Omega$ eine Nichtzugriffspolicy für τ. Die policy NACC wird von τ erfüllt, wenn gilt

$$\forall (t,\mathcal{H},\omega)\ ((t,\mathcal{H},\omega) \in \text{NACC} \ \dashrightarrow\ \exists f\ (f \in \mathcal{F} \wedge (t,\mathcal{H},\omega,f) \in \text{REJ}) \vee \neg\langle t(\mathcal{H},\omega)\rangle)$$

Die Frage, ob eine Nichtzugriffspolicy von einem ADT erfüllt wird, ist entscheidbar, da dies unmittelbar durch die Zurückweisungsrelation REJ und die Definition der Operationen gegeben ist. Dies hat zur Konsequenz, daß Zugriffsschutz unmittelbar durch die Definition von REJ bzw. die Definitionsbereiche der Operationen beeinflußbar ist.

Zur Formulierung der Zusicherung, daß bei der Ausführung einer Aktivität auf bestimmte Unterobjekte nicht lesend oder nicht schreibend zugegriffen werden kann, werden dann "Nichtabfrage-" bzw. "Nichtveränderungspolicies" definiert:

Sei $\tau = (\Omega, T, B, \omega_0)$ ein ADT mit Aktivitätenmenge $\mathcal{A}$, und sei $\text{Ind} = \{1, \ldots, m\}$ die Indexmenge für die Unterobjekte $d_1, \ldots, d_m$ eines Objekts des ADT's τ.

- Eine Nichtabfragepolicy für τ ist eine Menge NREF $\subseteq \mathcal{A} \times \Omega \times \text{Ind}$.

- Eine Nichtveränderungspolicy für τ ist eine Menge NMOD $\subseteq \mathcal{A} \times \Omega \times \text{Ind}$.

Die policies NREF bzw. NMOD enthalten also Tripel $(t,\mathcal{H},\omega,i)$, wobei $(t,\mathcal{H},\omega,i) \in$ NREF bzw. $\in$ NMOD bedeutet, daß mit der Aktivität $(t,\mathcal{H})$ im Zustand ω das Unterobjekt d_i nicht abgefragt bzw. nicht verändert werden darf. Die Erfüllung der policies NREF bzw. NMOD durch den entsprechenden ADT wird dann wie folgt definiert:

Sei τ ein ADT und NREF eine Nichtabfragepolicy sowie NMOD eine Nichtveränderungspolicy für τ.

- Die policy NREF wird von τ erfüllt, wenn gilt

$$\forall(t,\pi,\omega,i)\ ((t,\pi,\omega,i) \in \text{NREF} \dashrightarrow \neg abf(t,\pi,\omega,di))$$

- Die policy NMOD wird von τ erfüllt, wenn gilt

$$\forall(t,\pi,\omega,i)\ ((t,\pi,\omega,i) \in \text{NMOD} \dashrightarrow \neg änd(t,\pi,\omega,di))$$

Hierbei ist die Definition der Abfrage und Veränderung für ADT-Objekte fast analog zur Definition bei der erweiterten BSM. Ersetzt man dort die Operation "σ" durch die Aktivität "(t,π)", die Variable "u" durch das Unterobjekt "di" und den Datenzustand "d" durch den Wert des Objekts "ω", so kann man definieren:

$abf(t,\pi,\omega,di)$ gdw.

$$\exists\omega`\ (\omega \overset{=}{di} \omega` \wedge (\neg(t(\pi,\omega) \overset{=}{di} t(\pi,\omega`)) \vee$$

$$\neg((\{\omega(di)\} = RS(t,\pi,\omega)(di) \wedge$$

$$\{\omega`(di)\} = RS(t,\pi,\omega`)(di)) \vee$$

$$(RS(t,\pi,\omega)(di) = RS(t,\pi,\omega`)(di)))))$$

und

$änd(t,\pi,\omega,di)$ gdw. $\exists\omega`\ (\omega` \in RS(t,\pi,\omega) \wedge$

$$\omega`(di) \neq \omega(di))$$

Der einzige Unterschied zu den Definitionen bei der erweiterten BSM besteht darin, daß bei der Abfragedefinition die abgefragten Unterobjekte neben dem Einfluß auf andere

Unterobjekte auch die Ausgabewerte beeinflussen können. Dies wird in der Definition durch "$\neg(t(\pi,\omega) \underset{di}{=} t(\pi,\omega'))$" berücksichtigt.

Wegen der Nichtentscheidbarkeit der Abfrage- und Änderungsprädikate abf und änd ist natürlich auch die Frage, ob eine policy NREF bzw. NMOD von einem ADT erfüllt wird, im allgemeinen nicht entscheidbar. Man beachte, daß die policies NREF und NMOD genau den "Intended Access Relations" von Popek und Farber (vgl. 4.1.2) entsprechen, während die Prädikate "abf" und "änd" die "Actual Access Relations" repräsentieren. Die Erfüllung der policies NREF und NMOD entspricht damit dem Sicherheitsbegriff von Popek und Farber, wobei hier aber allgemeine relationale Aktivitäten und bei der Abfrage auch Fälle wie "di := f(di)" berücksichtigt werden.

Die definierten policies für eine Aktivität sind statische policies in dem Sinn, daß zu ihrer Definition die Dynamik der OV-Maschine nicht berücksichtigt werden muß. Sie definieren statisch für einen möglichen Objektzustand ein bestimmtes Verhalten. Da sich dies immer nur auf eine Aktivität und den momentanen Zeitpunkt bezieht, braucht die Berechnungsvorgeschichte nicht betrachtet zu werden.

Definition von policies für mehrere Aktivitäten

In der Literatur werden die den policies für mehrere Aktivitäten entsprechenden policies entweder als Informationsflußpolicies oder als Nichtbeeinflussungspolicies definiert. Hier soll sich auf die Definition von Informationsflußpolicies beschränkt werden, da Nichtbeeinflussungspolicies für das angegebene Modell nicht adäquat sind. Dies soll an einem Beispiel verdeutlicht werden:

Man betrachte einen multiuserfähigen Editor als ADT. Zwischen den einzelnen Aktivitäten findet eine gegenseitige Beeinflussung statt, weil z.B. keine gleichzeitige Benut-

zung gemeinsamer Files oder kein Anlegen zweier Files unter dem gleichen Namen erlaubt wird, und so Aktivitäten von vorhergehenden Aktivitäten beeinflußt werden. Da diese Art von "Beeinflussung" für ein gemeinsames Objekt völlig selbstverständlich ist, soll sie natürlich erlaubt werden, wogegen Informationsfluß im eigentlichen Sinn zwischen den einzelnen Aktivitäten nicht stattfinden soll. Die Modellierung eines solchen - für gemeinsame Objekte typischen - Verhaltens läßt sich nur mit Hilfe von Informationsflußpolicies erreichen, da generelle Nichtbeeinflussungspolicies, wie sie z.B. Goguen und Meseguer definieren, hierfür zu streng sind.

Für die Definition von Informationsfluß gibt es im vorliegenden Modell prinzipiell mehrere mögliche Ansätze. Hier werden zunächst informal die verschiedenen Möglichkeiten vorgestellt, und schließlich der für die gestellten Anforderungen geeignete Ansatz formal präsentiert.

Die einfachste Definition baut - völlig analog zu den für die BSM gegebenen Definitionen - auf einer rein statischen Betrachtungsweise auf. Seien "Schreib(t,π)" und "Les(t,π)" die Menge der Unterobjekte, die von einer Aktivität (t,π) verändert bzw. abgefragt werden können. Man kann nun Informationsfluß zwischen zwei Aktivitäten (t,π) und (t`,$\pi$`) so definieren, daß gilt

$$flow(t,\pi,t`,\pi`) \quad gdw \quad Schreib(t,\pi) \cap Les(t`,\pi`) \neq 0.$$

Diese Definition ist viel zu grob, da in ihr die Dynamik der OV-Maschine nicht berücksichtigt wird. Dies betrifft vor allem die Frage, ob von einem Zustand q, in dem ein Unterobjekt durch eine Aktivität (t,π) verändert wird, überhaupt ein Zustand q` erreicht werden kann, in dem die Aktivität (t`,π`) dieses Unterobjekt abfragen kann.

Eine Berücksichtigung der Maschinendynamik erfordert allerdings im Gegensatz zu rein statischen Betrachtungen eine Formulierung der Informationsflußdefinition mit Hilfe der möglichen Berechnungen der OV-Maschine. Dies könnte in-

formal z.B. so dargestellt werden:

"Information kann von einer Aktivität (t,π) zu einer Aktivität $(t`,\pi`)$ fließen, wenn es eine Berechnung gibt, in der zuerst durch die Aktivität (t,π) eine Menge von Unterobjekten verändert werden kann und anschließend durch die Aktivität $(t`,\pi`)$ eine andere Menge von Unterobjekten abgefragt werden kann, und diese beiden Mengen nicht disjunkt sind."

Leider ist auch diese Definition noch zu grob, da es sein kann, daß zwischen den beiden Aktivitäten (t,π) und $(t`,\pi`)$ eine Anzahl von Aktivitäten stattfinden, die alle für die Informationsübertragung relevanten Objekte überschreiben, so daß kein Informationsfluß von (t,π) nach $(t`,\pi`)$ stattfindet. Als praktisches Beispiel könnte man sich hierfür denken, daß in einem ADT zuerst eine "Löschaktivität" stattgefunden haben muß, bevor nach einer Aktivität (t,π) eine Aktivität $(t`,\pi`)$ ausgeführt werden kann. In einem Pufferpool könnte dies beispielsweise eine Wiederbenutzung freigegebener Puffer durch andere Benutzer sein.

Eine entsprechende Ergänzung der Definition müßte dann folgendermaßen lauten:

".... und es gibt mindestens ein Unterobjekt aus dem Durchschnitt dieser beiden Mengen, das zwischen der Ausführung von (t,π) und $(t`,\pi`)$ durch keine stattfindende Aktivität verändert werden kann."

Im folgenden werden nun, aufbauend auf dieser sehr exakten Informationsflußdefinition, Informationsflußpolicies eingeführt sowie formal definiert, wann eine gegebene policy erfüllt wird.

Sei $\tau = (\Omega, T, B, \omega o)$ ein ADT mit Aktivitätenmenge $\mathcal{A}$ und sei PRÄD die Menge aller möglichen Prädikate über Berechnungen der zugeordneten OV-Maschine.

Eine Informationsflußpolicy für τ ist eine Menge NFLOW $\subseteq \mathcal{A} \times \mathcal{A} \times$ PRÄD.

Die policy NFLOW enthält also Tupel (t, π, t', π', Φ), wobei $(t, \pi, t', \pi', \Phi) \in$ NFLOW bedeutet, daß von der Aktivität (t, π) zur Aktivität (t', π') in keiner Berechnung $\langle(qi, ei, ai)\rangle$ der dem ADT zugeordneten OV-Maschine mit $\Phi(\langle(qi, ei, ai)\rangle)$ Information fließen darf.

Zur Definition, wann eine Informationsflußpolicy NFLOW von τ erfüllt wird, benötigt man zunächst eine formale Informationsflußdefinition:

Information kann von einer Aktivität (t, π) zu einer Aktivität (t', π') unter der Bedingung Φ direkt fließen, in Zeichen dflow(t, π, t', π', Φ), wenn gilt

$\exists$ endliche Berechnung $\langle(qi, ei, ai) \mid 0 \leq i \leq m\rangle$ mit

$\Phi(\langle(qi, ei, ai) \ldots\rangle) \wedge \exists k, l, dj\ (0 \leq k < l \leq m \wedge$

$ek = (end, id, t, \pi) \wedge el = (end, id', t', \pi') \wedge$

$dj \in Schreib(t, \pi, qk) \cap Les(t', \pi', ql) \wedge$

$\forall i\ (k < i < l \wedge ei = (end, id'', t'', \pi'') \rightarrow$

$dj \notin Schreib(t'', \pi'', qi)))$

Hier gilt mit $q = (\omega, \ldots.)$

$$Schreib(t,\pi,q) := \{dj \mid dj \in \{d1,\ldots,dm\} \wedge änd(t,\pi,\omega,dj)\}$$

$$Les\ (t,\pi,q) := \{dj \mid dj \in \{d1,\ldots,dm\} \wedge abf(t,\pi,\omega,dj)\}$$

Neben der Möglichkeit des direkten Flusses zwischen zwei Aktivitäten kann Information von einer Aktivität (t,π) zu einer Aktivität $(t`,\pi`)$ auch über eine Folge von "Zwischenaktivitäten" fließen. Obwohl die zugehörige Definition recht komplex ist, soll sie, um die Vollständigkeit des Ansatzes darzulegen, präsentiert werden:

Information kann von einer Aktivität (t,π) zu einer Aktivität $(t`,\pi`)$ unter der Bedingung Φ fließen, in Zeichen $flow(t,\pi,t`,\pi`,\Phi)$, wenn gilt

$\exists$ endliche Berechnung $\langle(qi,ei,ai) \mid 0 \leq i \leq m\rangle$ mit

$\Phi(\langle(qi,ei,ai)..\rangle) \wedge \exists$ endl. Folgen $\langle kl \mid 0 \leq l \leq z\rangle$ $\langle djl \mid 0 \leq l \leq z-1\rangle$

$(0 \leq ko < k1 < \ldots < kz \leq m \wedge eko = (end, id_{ko}, t, \pi) \wedge$

$ekz = (end, id_{kz}, t`, \pi`) \wedge$

$\forall l,l+1\ (0 \leq l \leq z-1 \ \text{---}> \ djl \in Schreib(t_{kl}, \pi_{kl}, q_{kl}) \cap$

$Les(t_{kl+1}, \pi_{kl+1}, q_{kl+1}) \wedge$

$\forall i\ (kl < i < kl+1 \wedge ei = (end, id``, t``, \pi``) \ \text{----}>$

$djl \notin Schreib(t``, \pi``, qi))))$

Informal ausgedrückt bedeutet dies, daß allgemeiner (indirekter) Informationsfluß zwischen zwei Aktivitäten (t,π) und $(t`,\pi`)$ unter der Bedingung Φ stattfinden

kann, wenn es eine Berechnung, die Φ genügt, gibt, und innerhalb dieser Berechnung eine Folge von Aktivitäten existiert, zwischen denen Information direkt fließt und deren erste (t,π) und letzte $(t`,\pi`)$ ist.

Mit dieser Definition ist man nun in der Lage, zu definieren, wann eine Informationsflußpolicy durch einen ADT erfüllt wird:

Sei τ ein ADT und NFLOW eine Informationsflußpolicy für τ. Die policy NFLOW wird von τ erfüllt, wenn gilt

$$\forall (t,\pi,t`,\pi`,\Phi)\ ((t,\pi,t`,\pi`,\Phi) \in \text{NFLOW} \dashrightarrow \neg\, \text{flow}(t,\pi,t`,\pi`,\Phi))$$

Natürlich ist diese Eigenschaft wegen der Unentscheidbarkeit von "abf" und "änd", aber auch wegen der Unendlichkeit der Anzahl möglicher Berechnungen i.a. nicht entscheidbar.

Im folgenden soll nun die Definition von policies direkt in das ADT-Konzept integriert werden, so daß sich ein ADT mit policies wie folgt definieren läßt:

Ein synchronisierter, geschützter ADT mit policies ist ein Quintupel $\tau = (\Omega, T, B, Pol, \omega_o)$, wobei gilt

- (Ω, T, B, ω_o) ist ADT mit Aktivitätenmenge $\mathcal{A}$, Indexmenge Ind für Unterobjekte und PRÄD als Menge aller möglichen Prädikate für die Berechnungen der zugehörigen OV-Maschine.

- Pol = (NACC,NREF,NMOD,NFLOW) ist ein Quadrupel von policies mit

  NACC $\subseteq \mathcal{A} \times \Omega$
  NREF $\subseteq \mathcal{A} \times \Omega \times$ Ind
  NMOD $\subseteq \mathcal{A} \times \Omega \times$ Ind
  NFLOW $\subseteq \mathcal{A} \times \mathcal{A} \times$ PRÄD

Mit der Definition von policies für ADTs und deren Integration in das ADT-Konzept wurde der wichtigste Schritt zur Einführung des Sicherheitsbegriffs für ADTs schon gemacht. Vor dessen endgültiger Definition muß nun nur noch der Konsistenzbegriff für Verträglichkeitsrelationen des ADT`s eingeführt werden.

5.4.4.2 Die Konsistenz von Verträglichkeitsrelationen

Bereits im Rahmen der BSM wurden Betrachtungen zum Thema Konsistenz von Verträglichkeitsrelationen angestellt. Es ergab sich die Notwendigkeit, Konsistenzbetrachtungen von Paaren auf Mengen von Aktivitäten zu erweitern, da sich aus der paarweisen Verträglichkeit von Aktivitäten nicht auf deren gemeinsame Verträglichkeit schließen läßt. Außerdem müssen in den zugrundeliegenden Abfrage- und Veränderungsdefinitionen statt einfachen Unterobjekten Mengen von Unterobjekten betrachtet werden.

Im folgenden soll nun zunächst ein Verträglichkeitsprädikat für Mengen von Aktivitäten eingeführt werden und darauf aufbauend die Konsistenz von Verträglichkeitsrelationen definiert werden. Bei der Einführung des Verträglichkeitsprädikats werden dabei teilweise analoge Definitionen von der BSM ohne gesonderte Herleitung in die Notation der OV-Maschine übernommen.

Basis der folgenden Definitionen ist die Abfrage und Veränderung von Unterobjekten durch eine Aktivität - "abf(t,π,ω,di)" und "änd(t,π,ω,di)" - wie sie im Abschnitt 5.4.4.1 für die OV-Maschine definiert wurden.

Diese Definition läßt sich völlig analog zur BSM auf Mengen von Unterobjekten erweitern. Man schreibt dafür mit UO:={d1,...dm} und x $\subseteq$ UO

"abf(t,π,ω,x)" und "änd(t,π,ω,x)".

Damit läßt sich dann die Menge aller von einer Aktivität abgefragten Teilmengen von Unterobjekten definieren als

$$\underline{Abf(t,\pi,\omega)} := \{x \mid x \neq 0 \wedge x \subseteq UO \wedge abf(t,\pi,\omega,x)\}.$$

Da für die Menge Abf(t,π,ω) gilt

$$x \in Abf(t,\pi,\omega)$$

$$\dashrightarrow \forall y\ (y \subseteq UO \dashrightarrow x \cup y \in Abf(t,\pi,\omega)),$$

interessieren an der Menge "Abf(t,π,ω)" nur die bezüglich "$\subseteq$" minimalen Elemente. Man definiert deshalb für eine Menge A $\subseteq \mathcal{P}(UO) - 0$

$$\underline{Min(A)} := \{x \mid x \in A \wedge \neg \exists y\ (y \in A - \{x\} \wedge y \subseteq x)\}.$$

Die Menge der bezüglich "$\subseteq$" minimalen Elemente von Abf(t,π,ω) bezeichnet man somit als $\underline{Min(Abf(t,\pi,\omega))}$.

Die Menge der von einer Aktivität veränderten Unterobjekte läßt sich einfacher definieren. Man kann dafür schreiben

$$\underline{\text{Änd}(t, \mathcal{H}, \omega)} := \{dj \mid dj \in UO \wedge \text{änd}(t, \mathcal{H}, \omega, dj)\}.$$

Da in der gegebenen Definition auch Mengen von Aktivitäten betrachtet werden müssen, ist es notwendig, die Mengen Abf und Änd entsprechend zu erweitern:

$$\overline{\underline{\text{Abf}(\{t1, \mathcal{H}1, \ldots, tm, \mathcal{H}m\}, \omega)}} :=$$

$$\text{Abf}(t1, \mathcal{H}1, \omega) \cup \ldots \cup \text{Abf}(tm, \mathcal{H}m, \omega)$$

$$\overline{\underline{\text{Änd}(\{t1, \mathcal{H}1, \ldots, tm, \mathcal{H}m\}, \omega)}} :=$$

$$\text{Änd}(t1, \mathcal{H}1, \omega) \cup \ldots \cup \text{Änd}(tm, \mathcal{H}m, \omega)$$

Das Verträglichkeitsprädikat vtgl wird dann durch folgende rekursive Definition eingeführt:

(i) $\underline{\text{vtgl}(\{(t, \mathcal{H}), (t', \mathcal{H}')\}, \omega)}$ gdw

(i.i) $\text{Änd}(t, \mathcal{H}, \omega) \cap \text{Änd}(t', \mathcal{H}', \omega) = 0$

(i.ii) $\text{Min}(\text{Abf}(t, \mathcal{H}, \omega)) \cap \mathcal{P}(\text{Änd}(t', \mathcal{H}', \omega)) = 0$

(i.iii) $\mathcal{P}(\text{Änd}(t, \mathcal{H}, \omega)) \cap \text{Min}(\text{Abf}(t', \mathcal{H}', \omega)) = 0$

(ii) Sei Y eine Menge von Aktivitäten

$\underline{\text{vtgl}(\{(t, \mathcal{H})\} \cup Y, \omega)}$ gdw

(ii.i) $\text{vtgl}(Y, \omega)$

(ii.ii) $\text{Änd}(t, \mathcal{H}, \omega) \cap \overline{\text{Änd}}(Y, \omega) = 0$

(ii.iii) $Min(Abf(t, \mathcal{H}, \omega)) \cap \mathcal{P}(\overline{Änd(Y, \omega)}) = 0$

(ii.iv) $\mathcal{P}(Änd(t, \mathcal{H}, \omega)) \cap Min(\overline{Abf(Y, \omega)}) = 0$

Zur formalen Definition von vtgl sei hier bemerkt, daß die Potenzmengenbildung auf Änd(...) notwendig ist, um den Durchschnitt mit Min(Abf(...)), das ja als Elemente ebenfalls Teilmengen von UO enthält, bilden zu können.

Das Verträglichkeitsprädikat vtgl entspricht vom Grundgedanken her immer noch dem Ansatz aus <Ker82>, wo zwei Operationen als miteinander verträglich definiert werden, wenn sie auf verschiedene Variable zugreifen oder gemeinsame Variable nur abfragen. Die hier vorgestellte Definition berücksichtigt nur zusätzlich Aktivitätenmengen und Mengen von Unterobjekten.

Mit Hilfe des eingeführten Verträglichkeitsprädikats läßt sich nun die Konsistenz einer Verträglichkeitsrelation definieren:

Sei $\tau = (\Omega, T, B, Pol, \omega o)$ ein ADT. Die Verträglichkeitsrelation VTGL des ADT's τ heißt konsistent, wenn gilt

$\forall (t, \mathcal{H}, x, \omega)\ ((t, \mathcal{H}, x, \omega) \in VTGL \ \dashrightarrow$

$vtgl(\{(t, \mathcal{H})\} \cup x, \omega))$

Die Konsistenz von Verträglichkeitsrelationen ist ebenso wie die Frage nach der Erfüllung von gegebenen policies im allgemeinen nicht entscheidbar, weil als Basis wieder die Abfrage und Veränderung in der OV-Maschine dient.

Man beachte, daß die gegebene Definition zwar ein hinreichendes, aber nicht notwendiges Kriterium für die Konsistenz von Verträglichkeitsrelationen gibt. In diesem Zusammenhang sei an das Beispiel aus Abschnitt 5.3.4.1 mit "if u=100 then ... fi" und "u:=u+1" in Zuständen mit u=100 erinnert. Wie dort bereits erwähnt, soll auf eine Erweiterung des Ansatzes in diese Richtung verzichtet werden, da der damit erreichbare höhere Parallelitätsgrad die erforderliche komplexere Definition nicht rechtfertigen würde.

An dieser Stelle soll noch kurz auf den Konsistenzbeweis von Verträglichkeitsrelationen in der Praxis eingegangen werden. Hier werden die in VTGL enthaltenen Tupel im Regelfall durch Prädikate Φ über Ω gegeben sein. Analog zur BSM lassen sich dann die Mengen Abf und Änd und damit auch das Verträglichkeitsprädikat vtgl für Prädikate Φ definieren:

Man schreibt

$$abf(t,\mathcal{H},\Phi,x) \quad gdw \quad \exists\,\omega\,(\,\Phi(\omega) \wedge abf(t,\mathcal{H},\omega,x))$$

$$Abf(t,\mathcal{H},\Phi) := \{x \mid x \neq 0 \wedge x \subseteq U0 \wedge abf(t,\mathcal{H},\Phi,x)\}$$

und analog dazu änd(...) und Änd. Mit den so erweiterten Definitionen kann dann vtgl(x, Φ) eingeführt werden.

Für die oftmals nicht triviale Frage, welche Unterobjektmengen von einer Aktivität bei verschiedenen Prädikaten Φ abgefragt werden, kann die folgende Beziehung wertvolle Hilfestellung geben. Es gilt nämlich für alle $x \in \mathcal{P}(\mathcal{A})$

$$\Phi = \Phi 1 \wedge \Phi 2 \longrightarrow$$

$$Abf(x,\Phi) \subseteq Abf(x,\Phi 1) \cap Abf(x,\Phi 2)$$

wobei für $\Phi 1 \wedge \Phi 2 \neq$ false sogar die Gleichheit gilt!

Im folgenden Abschnitt wird nun mit Hilfe der Definitionen für die Erfüllung von policies durch einen ADT und für die Konsistenz von Verträglichkeitsrelationen der Sicherheitsbegriff für ADTs eingeführt.

5.4.4.3 Der Sicherheitsbegriff für ADTs

Sicherheit bzw. Zuverlässigkeit von SW-Systemen bedeutet, daß sich die Systeme "wie gewünscht" verhalten. Unter der Voraussetzung einer Trennung zwischen Spezifikation und Implementierung sind dabei zwei Bedingungen (vgl. Kap. 3) zu erfüllen: Zum einen muß die Spezifikation gewisse gegebene Zusicherungen (policies) erfüllen, zum anderen muß die Implementierung die in der Spezifikation nichtprozedural festgelegten Vorgaben korrekt in prozeduralen Code umsetzen.

Die zweite Fragestellung - die Korrektheit der Implementierung - soll hier, da sie stark anwendungsorientiert ist, im folgenden Kapitel 6 behandelt werden. Für die erste Frage, ob eine Spezifikation gegebene policies erfüllt (design verification), können bereits im formalen Modell, das ja den Rahmen für die Spezifikation vorgibt, Ansätze zur Verifikation gegeben werden. Die Erfüllung gegebener policies durch eine Spezifikation wird dann "Sicherheit" (bzgl. der Spezifikation) genannt.

Übertragen auf das angegebene formale Modell läßt sich die Sicherheit von ADTs durch zwei Eigenschaften zeigen:

- den Nachweis, daß die gegebenen policies Pol durch die im ADT gegebene Spezifikation der möglichen Zugriffe und Zugriffsbeschränkungen durchgesetzt wird, und
- den Nachweis, daß die angegebene Verträglichkeitsrelation VTGL konsistent ist.

Implizite Voraussetzung hierfür ist natürlich, daß von der OV-Maschine die spezifizierten Zugriffsbeschränkungen B durchgesetzt werden, was sich aber unmittelbar aus der Definition der OV-Maschine ergibt. Man kann deshalb den Sicherheitsbegriff für ADTs wie folgt definieren:

Sei $\tau = (\Omega, T, B, Pol, \omega_o)$ ein ADT. τ heißt sicher, wenn gilt

(i) Pol = (NACC,NREF,NMOD,NFLOW) und die policies NACC, NREF, NMOD und NFLOW werden von τ erfüllt.

(ii) VTGL ist konsistent.

Mit dieser Definition ist ein präziser Sicherheitsbegriff für ADTs gegeben, der der Forderung nach strikter Trennung zwischen policy und Mechanismus genügt.

- Die policy besteht aus den gegebenen Schutzpolicies Pol und der Konsistenzeigenschaft für Verträglichkeitsrelationen.

- Der Mechanismus ist durch die OV-Maschine zusammen mit der Spezifikation des ADT`s (Operationen und Zugriffseinschränkungen) gegeben.

- Die Sicherheit eines ADT`s besteht darin, daß der gegebene Mechanismus die policy durchsetzt, d.h. Pol von τ erfüllt wird und VTGL konsistent ist.

Der eingeführte Sicherheitsbegriff bezieht sich auf ADTs und bringt somit eine lokale Definition von Sicherheit. Daneben lassen sich bei der Integration von ADTs in eine SBSM systemweite bzw. globale Eigenschaften angeben. Im nächsten Abschnitt soll kurz gezeigt werden, wie sich solche

globalen Eigenschaften definieren lassen und auf lokale policies zurückführen lassen.

5.4.4.4 Systemweite Eigenschaften

Die bisherige Definition von policies beschränkte sich auf lokale, ADT-bezogene policies, die durch Prädikate über Aktivitäten bzw. Beziehungen zwischen Aktivitäten formuliert wurden. Im gegebenen Modell ist eine solche lokale Policydefinition eigentlich völlig ausreichend, da eine Kommunikation nur über globale ADT-Objekte möglich ist, da deshalb ADT-bezogene policies systemweit wirken, und das System auch nur ADT-bezogen kontrolliert werden kann.

Jedoch lassen sich aufbauend auf lokalen Eigenschaften für globale ADT-Objekte systemweite Eigenschaften definieren, die Subjekte bzw. Prozesse explizit mit einbeziehen, und Subjekt-Objekt- bzw. Subjekt-Subjekt-Beziehungen formulierbar machen. Natürlich sind auch im bisherigen Konzept Prozesse in policies einbeziehbar, jedoch lediglich als einer der möglichen Eingabeparameter, und nicht herausgehoben aus den übrigen Eingabeparametern.

Im folgenden werden der Vollständigkeit halber Möglichkeiten zur Formulierung systemweiter Eigenschaften vorgestellt, wobei sich auf eine kurze und informale Darstellung beschränkt werden soll.

Zur Formulierung systemweiter Eigenschaften muß auf den in den vorhergehenden Abschnitten definierten lokalen Eigenschaften für ADT-Objekte aufgebaut werden. Zusätzlich muß für die Klärung systemweiter Eigenschaften noch die Frage gestellt werden, ob ein Prozeß P die entsprechende Aktivität (t,π) auf das Objekt o überhaupt aufrufen kann. Hier kann allerdings nur gefragt werden, ob ein solcher Aufruf prinzipiell möglich ist, d.h. ob in der Aktivität z.B. als Eingabeparameter die richtige Prozeßidentifikation angegeben

ist und ähnliches. Man bezeichne dies mit "auf(P,o,t,π)".

Subjekt-Objekt-Beziehungen

Analog zum lokalen Fall kann man hier die Eigenschaft definieren, daß ein Prozeß mit einer Aktivität in einem bestimmten Zustand auf ein Objekt nicht zugreifen kann. Man kann dafür "nacc(P,o,t,π,ω)" schreiben und definieren

$$\text{nacc}(P,o,t,\pi,\omega) \quad \text{gdw} \quad \neg\text{auf}(P,o,t,\pi) \vee \exists f\,(f \in \mathcal{F} \wedge (t,\pi,\omega,f) \in \text{REJ})$$

Die Eigenschaft, daß ein Prozeß bestimmte Unterobjekte eines Objekts nicht abfragen oder verändern kann, werde mit "nref(P,o,t,π,ω,di)" bzw. "nmod(...)" bezeichnet.

Man kann dann mit $x \subseteq UO$ definieren

$$\text{nref}(P,o,t,\pi,\omega,di) \quad \text{gdw} \quad \neg\text{auf}(P,o,t,\pi) \vee \neg\text{abf}(t,\pi,\omega,di)$$

$$\text{nref}(P,o,t,\pi,\omega,x) \quad \text{gdw} \quad \neg\text{auf}(P,o,t,\pi) \vee \forall di \in x\,(\neg\text{abf}(t,\pi,\omega,di))$$

$$\text{nref}(P,o,t,\pi,\Phi,x) \quad \text{gdw} \quad \neg\text{auf}(P,o,t,\pi) \vee \forall di \in x\,(\neg\text{abf}(t,\pi,\Phi,di))$$

$$\text{nref}(P,o,x) \quad \text{gdw} \quad \forall t,\pi,\omega\,(\text{nref}(P,o,t,\pi,\omega,x))$$

$$\text{nref}(P,o) \quad \text{gdw} \quad \forall x \subseteq UO\,(\text{nref}(P,o,x))$$

Analoges läßt sich für nmod definieren.

Subjekt-Subjekt-Beziehungen

Die Definition von Subjekt-Subjekt-Beziehungen baut auf dem Informationsflußbegriff des lokalen Falls auf. Auch hier unterscheidet man zwischen direktem und indirektem Fluß und kann somit definieren

ndflow(P,P`,o,Φ) gdw $\forall (t,\pi),(t`,\pi`) \in \mathcal{O}l$

$(\neg auf(P,o,t,\pi) \vee \neg auf(P,o,t`,\pi`) \vee \neg flow(t,\pi,t`,\pi`, \Phi))$

ndflow(P,P`) gdw $\forall o \in \bar{O} \quad \forall \Phi$ (ndflow(P,P`,o,Φ))

nflow(P,P`) gdw $\neg \exists$ endliche Folge <Pi| 1≤i≤m> mit

$P = P1 \wedge P` = Pm \wedge \forall Pi, Pi+1 \ (1 \leq i \leq m-1 \ \text{---}> \ dflow(P,P`))$

Die Verifikation systemweiter Eigenschaften wird somit auf die Verifikation lokaler Eigenschaften zurückgeführt, und ist deshalb i.a. nicht entscheidbar

Analog zum lokalen Fall lassen sich auch systemweite policies definieren, die Prozesse explizit beinhalten. Die Einhaltung dieser policies kann dann durch Verifikation der entsprechenden systemweiten Eigenschaften gezeigt werden.

5.5 Leistungen und Möglichkeiten des vorgestellten Modells

In diesem Abschnitt sollen die Leistungen des vorgestellten formalen Modells zusammengefaßt und vor allem an den an das Modell gestellten Forderungen (vgl. 3.2.3, 4.5 und 5.1) gemessen werden.

Hauptziel beim Entwurf des formalen Modells war die Entwicklung einer Basis für die Spezifikation, die korrekte Implementierung und die Verifikation der Zuverlässigkeit asynchroner, geschützter Systeme. Dieser Forderung wird das Modell dadurch gerecht, daß es eine direkte Grundlage für die Spezifikation, Implementierung und Verifikation geschützter, asynchroner Systeme, im Sinne eines einheitlichen Ansatzes vom formalen Modell bis hin zur Realisierung, bietet. Dem trägt auch die Tatsache Rechnung, daß das Modell problemadäquat ist, d.h. daß eine Klärung anwendungsbezogener Fragen wie z.B. Verträglichkeits- oder Informationsflußfragen bereits im formalen Modell möglich ist.

Der Forderung nach einer Berücksichtigung moderner Software-Entwurfsprinzipien bereits im formalen Modell trägt die Unterstützung des Abstraktionsprinzips durch die Integration des Strukturierungskonzepts der Abstrakten Datentypen Rechnung. Dies beinhaltet automatisch den Zwang zu einer objektorientierten Betrachtungsweise, was ebenfalls modernen softwaretechnologischen Erkenntnissen entspricht. Das Modell bietet ein einheitliches Konzept für die Aufgaben der allgemeinen Verwaltung abstrakter Objekte, für die Synchronisation der Zugriffe auf Objekte und den Schutz vor unerlaubten Zugriffen. Alle diese Verwaltungsaufgaben werden im jeweiligen Abstrakten Datentyp für das abstrakte Objekt definiert, und von der zugeordneten OV-Maschine als Laufzeit-Verwaltungsmechanismus erfüllt. Diese räumliche Nähe aller für die Verwaltung der Zugriffe auf das Objekt verantwortlichen Mechanismen untereinander, und zum Objekt selbst, trägt dem Lokalitätsprinzip Rechnung und erleichtert damit auch die Modifizierbarkeit von mit der auf dem Modell

aufbauenden Entwurfsmethode erstellten Systemen. Schließlich zeigt sich die Anwendbarkeit des Ansatzes auf SW-Systeme zur Fertigungsautomatisierung und Produktionssteuerung und dessen leichte Verständlichkeit am zugeordneten Spezifikations- und Implementierungskonzept, das im folgenden Kapitel präsentiert wird, und an einem konkreten Anwendungsbeispiel in Kapitel 7 dieser Arbeit.

Für den Bereich des Schutzes von abstrakten Objekten vor unerlaubten Zugriffen bietet das Modell eine strikte Trennung von policy und Mechanismus und die Möglichkeit der Formulierung beliebiger policies. So können im Modell neben Standard-Schutzpolicies (z.B über Rechte definiert oder militärische Sicherheit) sowohl content dependent policies als auch history dependent policies formuliert werden. Die policies beziehen sich dabei sowohl auf reinen Zugriffsschutz als auch auf Informationsfluß. Ein klarer Sicherheitsbegriff im formalen Modell erleichtert schließlich die Verifizierbarkeit von policies bzw. von Eigenschaften konkreter Systeme.

Im folgenden Kapitel soll nun die durch das formale Modell festgelegte Spezifikations- und Implementierungsmethode für asynchrone, geschützte Systeme präsentiert werden, um dann im Kapitel 7 die Anwendungen der Methoden an einem Beispielsystem aus der Fertigungsautomatisierung darzulegen.

6. Spezifikation und korrekte Implementierung synchronisierter, geschützter Abstrakter Datentypen

Ziel dieser Arbeit ist die Entwicklung von Methoden zur Konstruktion zuverlässiger SW-Systeme unter Berücksichtigung der Anforderungen, die aus einem Einsatz der Systeme in der Fertigungsautomatisierung und Produktionssteuerung resultieren (vgl. Kapitel 2). Für den potentiellen Benutzer dieser Methoden bedeutet dies die Angabe eines Spezifikations- und Implementierungskonzepts für zuverlässige Software. Um die Allgemeinheit eines solchen Konzepts gewährleisten zu können, war es notwendig, zuerst ein formales Modell zur Definition der Semantik des Konzepts zu erstellen, in dem auch gewisse Eigenschaften des Konzepts nachgewiesen werden können.

Ausgehend vom Modell der Betriebssystemmaschine <Ker82> wurden im Kapitel 5 dieser Arbeit durch eine Interpretation und die Integration von prozeduraler und Datenabstraktion formale Modelle für synchronisierte, geschützte ADTs und die OV-Maschine - als Verwaltungsmechanismus für Zugriffe auf ADT-Objekte - präsentiert. Wesentliche Gesichtspunkte dabei waren die Einhaltung der SW-Entwurfsprinzipien aus Kapitel 3, die Möglichkeit der Formulierung beliebiger OV-Strategien, eine strikte Trennung zwischen policy und Mechanismus sowie ein klarer Sicherheitsbegriff, und nicht zuletzt die Problemadäquatheit des Modells. Hauptziel bei der Erstellung des formalen Modells war jedoch eine direkte Ableitbarkeit der zugeordneten Spezifikations- und Implementierungsmethode im Sinne eines einheitlichen Ansatzes.

Im vorliegenden Kapitel 6 soll nun dieses Spezifikations- und Implementierungskonzept vorgestellt werden. Das Kapitel wendet sich also an den Benutzer der Methode. Aus diesem Grund soll hier, soweit möglich, weitgehend auf abstrakte Formalismen verzichtet werden und eine Erklärung der Konstrukte an einfachen Beispielen gegeben werden. Vor einer genaueren Erläuterung soll aber hier zunächst ein Überblick über das Konzept gegeben werden, und insbesondere

der Zusammenhang zwischen formalem Modell, Spezifikation und Implementierung dargestellt werden.

6.1 Einordnung des Spezifikations- und Implementierungskonzepts

Bereits im formalen Modell wurde streng zwischen Abstrakten Datentypen als Konzept zur Definition des Verhaltens abstrakter Objekte und der OV-Maschine als Verwaltungsmechanismus für Zugriffe auf abstrakte Objekte differenziert. Diese Trennung setzt sich auch auf der Ebene der Spezifikation und Implementierung fort:

- Das Verhalten und die Regeln zur Verwaltung abstrakter Objekte wird im Spezifikationsteil des ADT`s, der auch die Schnittstelle zur Außenwelt ist, festgelegt. Der Implementierungsteil stellt dann die konkrete programmtechnische Realisierung dar und setzt auf einem vorhandenen Mechanismus zur Verwaltung der abstrakten Objekte auf. Bei der Implementierung handelt es sich also um die Lösung eines Objektverwaltungsproblems mit Hilfe eines gegebenen OV-Mechanismus, dessen Funktion durch das formale Modell der OV-Maschine festgelegt ist.

- Davon zu trennen ist die Spezifikation und Implementierung des OV-Mechanismus selbst, d.h. die Umsetzung des formalen Modells der OV-Maschine in Programmcode. In der Praxis wird dieser OV-Mechanismus i.a. in das Betriebssystem integriert werden. Im Rahmen dieser Arbeit soll jedoch auf die Spezifikation und Implementierung des OV-Mechanismus nicht eingegangen werden, da hier mehr anwendungsorientierte Gesichtspunkte der Methode im Vordergrund stehen. Der an dieser Thematik interessierte Leser sei auf <Ker82>, <Mac83> und <Rei83> verwiesen.

Einen Überblick über die Zusammenhänge zwischen formalem Modell und Spezifikations- und Implementierungskonzept einerseits sowie ADTs und OV-Mechanismus andererseits gibt nun Abbildung 6-1.

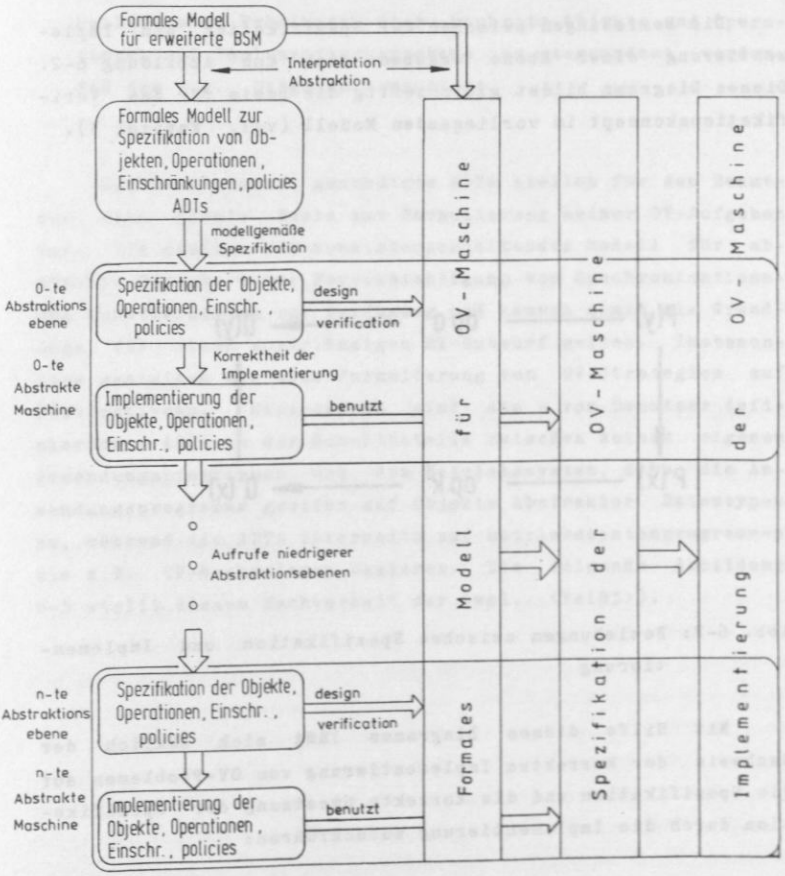


Abb. 6-1: Zusammenhang zwischen formalem Modell und Spezifikations- und Implementierungskonzept

Hier wurde auch ein Entwurf des Systems in mehreren Abstraktionsebenen zugrundegelegt. Wie in Kapitel 3 dargestellt, benutzen die Implementierungen der ADTs einer Ebene i die Spezifikationen der nächstniedrigen Ebene i+1 durch Aufruf der Operatione dieser Ebene. Dabei bleibt die Implementierung dieser Operationen der Ebene i verborgen.

Die Beziehungen zwischen der Spezifikation und Implementierung einer Ebene ergeben sich aus Abbildung 6-2. Dieses Diagramm bildet gleichzeitig die Basis für das Verifikationskonzept im vorliegenden Modell (vgl. Kapitel 3).

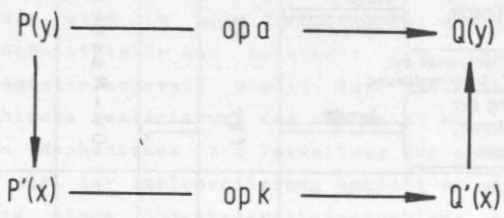


Abb. 6-2: Beziehungen zwischen Spezifikation und Implementierung

Mit Hilfe dieses Diagramms läßt sich nämlich der Nachweis der korrekten Implementierung von OV-Problemen auf die Spezifikation und die korrekte Umsetzung der Spezifikation durch die Implementierung zurückführen:

- Bei der Verifikation der Spezifikation (design verification) handelt es sich um den Nachweis invarianter Systemeigenschaften, der Erfüllung von policies und der Konsistenz der gegebenen Verträglichkeitsrelationen.

- Zum Nachweis einer korrekten Implementierung muß die Schnittstelle zwischen Spezifikation und Implementierung - die sogenannte Repräsentationsfunktion - betrachtet werden. Sie ordnet konkreten Objekten der Implementierung abstrakte Objekte der Spezifikation zu. Hierbei ist besonders zu berücksichtigen, daß den Prädikaten über abstrakte Objekte und Operationen auf Spezifikationsebene äquivalente Prädikate über konkrete Objekte und Operationen auf Implementierungsebene so zugeordnet werden, daß das o.g. Diagramm kommutiert.

Synchronisierte, geschützte ADTs stellen für den Benutzer eine ideale Basis zur Formulierung seiner OV-Aufgaben dar. Sie stellen ein konsistenzerhaltendes Modell für abstrakte Objekte unter Berücksichtigung von Synchronisations- und Schutzaspekten zur Verfügung und können somit als Grundlage für einen zuverlässigen SW-Entwurf gelten. Insbesondere gestatten sie eine Formulierung von OV-Strategien auf Problemniveau. Einzuordnen sind die - vom Benutzer definierten - ADTs an der Schnittstelle zwischen seinen eigenen Anwendungsprogrammen und dem Betriebssystem, d.h. die Anwendungsprogramme greifen auf Objekte Abstrakter Datentypen zu, während die ADTs ihrerseits auf Betriebssystemprogrammen wie z.B. OV-Mechanismen basieren. Die folgende Abbildung 6-3 stellt diesen Sachverhalt dar (vgl. <Rei83>).

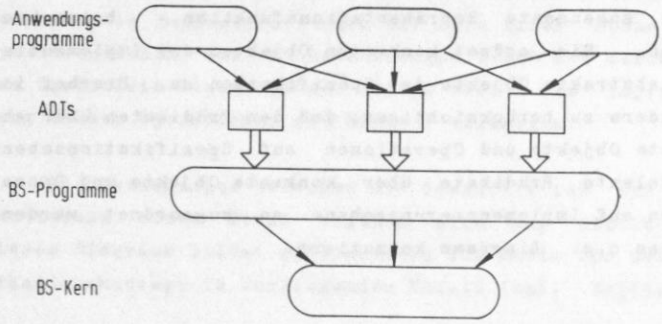


Abb. 6-3: Einordnung Abstrakter Datentypen

6.2 Die Spezifikation synchronisierter, geschützter ADTs

Basis des hier zu entwickelnden Spezifikationskonzepts ist das formale Modell für synchronisierte, geschützte ADTs, wie es im Kapitel 5 eingeführt wurde. Hierdurch ist der formale Aufbau der Spezifikation, d.h. die Einteilung in Deklarationsteil (Abstrakte Repräsentation, Parameter, lokale Größen), Operationen, Zugriffseinschränkungen und policies bereits klar vorgegeben. Die Art der Definition dieser Größen in Form von uninterpretierten Wertebereichen bzw. Relationen muß jedoch für eine Spezifikation konkreter Objekte, Operationen und OV-Strategien in Sprachkonstrukte einer Spezifikationssprache übertragen werden. Charakteristisch an dieser Spezifikationssprache ist deren nichtprozeduraler Charakter, d.h. die Operationen werden nicht durch Angabe eines Algorithmus auf die abstrakten Objekte, sondern allein durch ihre Wirkung auf die Objekte (in Form von Prädikaten, d.h. Pre- und Postconditions) beschrieben.

Die hier eingesetzte Spezifikationsmethode basiert im wesentlichen auf der Methode der Prädikatentransformation von Hoare (<Hoa69>, <Wul76>), die in <Ker82> um Synchronisations- und in dieser Arbeit um Schutzkonstrukte erweitert wurde. Bemerkenswert am hier verwendeten Spezifikationskonzept ist die Tatsache, daß sich die (prozedurale) Implementierung fast direkt aus der (nichtprozeduralen) Spezifikation ergibt. Dies resultiert aus dem zugrundeliegenden gemeinsamen formalen Modell für synchronisierte, geschützte ADTs und zeigt nochmals deutlich die Notwendigkeit einer formalen Modellbildung auf.

Bei der Angabe der Spezifikationsmethode wird auf eine formale Definition der Sprache durch Angabe einer Grammatik (z.B. in BNF) verzichtet. Gleiches gilt auch für eine formale Angabe der Semantik, die sich aber unmittelbar aus dem formalen Modell der OV-Maschine ergibt. Eine formale Definition hierfür wäre zu aufwendig und würde dem Charakter dieses Kapitels als Einführung für den Benutzer der Methode nicht entsprechen. Hier werden stattdessen die konkreten Sprachkonstrukte informal eingeführt und deren Bedeutung anhand von einfachen Beispielen erklärt. Die Anwendbarkeit der Methode auch bei größeren Systemen soll sich dann aus den Ausführungen im Kapitel 7 dieser Arbeit ergeben.

6.2.1 Die Spezifikation von ADTs ohne Schutzkonstrukte

Das Konzept zur Spezifikation von ADTs ohne Schutzkonstrukte entspricht im wesentlichen den bei <Ker82> angegebenen Konstrukten, die sich aus Gründen der leichten Handhabbarkeit ihrerseits stark an PASCAL anlehnen. Trotzdem sollen diese Sprachkonstrukte hier nochmals vorgestellt werden, da dies für ein allgemeines Verständnis der Methode und insbesondere der im nächsten Abschnitt einzuführenden Spezifikationen von Schutzkonstrukten unerläßlich erscheint. Durch den hier gegebenen Überblick wird dem Leser ein Nachschlagen in der o.g. Literatur erspart.

Die Darstellung des Spezifikationskonzepts erfolgt in kompakter, informaler Form unter Einbeziehung von Beispielen. Der wesentliche Unterschied zu <Ker82> ist, daß hier eine direkte Entsprechung zwischen dem formalen Modell für ADTs und den Spezifikationskonstrukten besteht, während dort auf der uninterpretierten BSM aufgesetzt wird. Aus diesem Grund ergeben sich die konkreten Sprachkonstrukte in sehr natürlicher Weise aus dem formalen Modell.

Man betrachte nochmals kurz das formale Modell für ADTs, jedoch ohne Schutzkonstrukte. Ein ADT ist ein Quadrupel $\tau = (\Omega, T, B, \omega_o)$ mit

Ω ist Abstrakte Repräsentation für Objekte des ADT`s

T ist endliche Operationsmenge $T = \{t1, \ldots, tn\}$ mit
$tj \subseteq (Ej \times \Omega) \times (\Omega \times Aj)$, wobei Ej und Aj die Wertebereiche der Ein- und Ausgabeparameter sind

B ist ein Paar B = (VTGL,PRIO) mit

$VTGL \subseteq \alpha \times \mathcal{P}(\alpha) \times \Omega$ ist Verträglichkeitsrelation
$PRIO \subseteq \alpha \times \alpha \times \Omega$ ist Prioritätsrelation

ω_o ist die initiale Belegung der Objekte mit $\omega_o \in \Omega$

Die entsprechende sprachliche Darstellung der Spezifikation eines ADT`s ist dann

```
SPECIFICATION

    DECLARATIONS
        ABSTRACT REPRESENTATION   ...   INITIALLY
        PARAMETERS
        LOCAL TYPES

    SYN
        COMP
        PRIO

    OPERATIONS
        NBL
        EFFECTS
```

Hier enthält der Deklarationsteil die Abstrakte Repräsentation mit der entsprechenden initialen Belegung, die Ein- und Ausgabeparameter der Operationen und eventuelle lokale Typdefinitionen. Im Synchronisationsteil SYN sind die Verträglichkeits- und Prioritätsrelation in Form von Prädikaten über Aktivitätenmengen und Objektzuständen angegeben. Die Operationen werden schließlich durch ihre Wirkung auf die Abstrakte Repräsentation und die Parameter mit Hilfe der Methode der Prädikatentransformation (vgl. Kapitel 3) beschrieben.

Deklarationsteil

Im Deklarationsteil müssen sämtliche im ADT verwendeten Daten deklariert werden. Es handelt sich um eine typorientierte Deklaration, wie sie z.B. auch in PASCAL üblich ist. Die konkreten Konstrukte sind deshalb stark an die PASCAL-Notation angelehnt. Diese Konstrukte entsprechen auch direkt den Größen des formalen Modells. Die jeweiligen Korrespondenzen sind dabei

ABSTRACT REPRESENTATION	Ω
INITIALLY	ω_0
PARAMTERS	Ej, Aj ($1 \leq j \leq n$) und Hilfsgrößen
LOCAL TYPES	Def. lokaler Typen

Die sprachliche Angabe der einzelnen Komponenten geschieht nun in der Form

Komponentenname : Komponententyp

Die verwendeten Typen sollen hier kurz aufgeführt werden:

- Standardtypen
  Die in PASCAL üblichen Standardtypen INTEGER, REAL, CHAR, STRING und BOOLEAN werden hier übernommen.

- einfache Typen
  Hier handelt es sich um skalare oder Aufzählungstypen und Unterbereiche dieser Typen. Beispiele hierfür sind
  TYPE Farbe = (rot,blau,grün,gelb)
  TYPE Jahr = 1900 .. 1999

- strukturierte Typen
  Hierunter fallen Felder, Mengen, Records und als Ergänzung zu PASCAL Sequenzen.

  Felder werden durch das ARRAY-Konstrukt definiert. Die allgemeine Form ist

  feld : ARRAY [1..n] OF elementtyp

  Hierdurch wird ein Feld mit Namen "feld" der Länge "n" mit Komponenten vom Typ "elementtyp" definiert. Natürlich sind auch mehrdimensionale Felder möglich. "feld[i]" bezeichnet das i-te Element des Felds "feld".

Die Mengendefinition erfolgt durch das SET-Konstrukt. Durch

menge : SET OF elementtyp

wird eine endliche Menge mit Namen "menge" von Elementen des Typs "elementtyp" definiert. Als Standardoperationen auf Mengen werden die Vereinigung " ∪ ", der Durchschnitt " ∩ " und die Differenz " - " von Mengen eingeführt. CARD gibt die Kardinalität einer Menge an.

Zur Strukturierung von Daten und zur Deklaration neuer Typen durch kartesische Produktbildung dient das RECORD-Konstrukt, das auch bei der Definition lokaler Typen angewandt wird. Der allgemeine Aufbau ist

```
TYPE newtype = RECORD
                 name 1 : oldtype 1;
                 ....
                 name n : oldtype n;
                 END;
```

Hier wird ein neuer Typ mit Namen "newtype" definiert, der aus dem kartesischen Produkt der Typen "oldtype 1",..,"oldtype n" entsteht. Die Größen vom Typ "newtype" lassen sich dann als Tupel (name 1,...,name n) darstellen. Einzelne Unterkomponenten eines RECORDS werden durch Qualifizierung (".") angesprochen. Ein konkretes Beispiel soll dies verdeutlichen:

```
TYPE datum = RECORD
               tag     : 1 .. 31;
               monat   : (Januar, ... ,Dezember);
               jahr    : 1900 .. 1999;
               END;
```

definiert Größen des Typs "datum" als Tripel (tag,monat,jahr). Sei "dat" eine Größe dieses Typs, so greift man auf die Unterkomponenten durch "dat.tag", "dat.monat" und "dat.jahr" zu.

Zusätzlich zu diesen in PASCAL bekannten strukturierten Typen werden hier Sequenzen eingeführt. Das Konstrukt

```
sequ : SEQUENCE OF elementtyp
```

definiert eine Folge mit Namen "sequ" von Elementen des Typs "elementtyp". Als Standardoperationen auf Sequenzen werden definiert

```
LENGTH ..... Länge der Sequenz
FIRST ..... erstes Element der Sequenz
REMAINDER ... Sequenz ohne erstes Element
ADD ..... Konkatenation (Hinzufügen eines Elements)
```

- weitere Typen und Typbildungsoperatoren
  Analog zu PASCAL werden hier noch der Typ POINTER ("↑.") als Zeiger auf Daten und der Operator UNION als Vereinigung von Typen eingeführt. Auch eine Verwendung importierter, d.h. anderswo deklarierter, Typen sei möglich.

Ein Beispiel für eine korrekte Deklaration von abstrakten Ojekten findet sich nach der Erläuterung des Operationsteils.

Operationsteil

Im formalen Modell werden die Operationen tj als Relationen $tj \in (Ej \times \Omega) \times (\Omega \times Aj)$ dargestellt. Sie ordnen einem Zustand aus $Ej \times \Omega$ einen Zustand aus $\Omega \times Aj$ zu. Da diese Operationen i.a. nicht funktional sind, wurde die allgemeine relationale Darstellung gewählt. In Anlehnung an Keramidis werden die Operationen hier in der Form

$$tj = \{(\pi, \omega, \omega', \alpha) \mid P(\pi, \omega) \wedge Q(\omega', \alpha)\}$$

angegeben, d.h. sie werden durch Preconditions und Postconditions in der Hoare`schen Schreibweise charakterisiert. $P(\pi, \omega)$ charakterisiert dabei diejenigen Werte von π und ω, für die tj definiert ist, d.h. wo gilt $<t(\pi, \omega)>$, $Q(\omega', \alpha)$ charakterisiert den Zustand und die Ausgabe nach der Ausführung von tj.

Auf spezifikationssprachlicher Ebene werden die Operationen dargestellt als

```
op (x1,...,xm) ---> y1,...,yn
NBL:      P(ak1,...,akp,x1,...xm)
EFFECTS: Q(ak1,...,akp,`ak1,...,`akp,x1,...,xm,y1,...,yn)
```

wobei op der Operationsname ist, also tj entspricht, x1,...,xm die Eingabeparameter und y1,...,yn die Ausgabeparameter der Operation, und ak1,...,akp die Komponenten der abstrakten Repräsentation bezeichnen. Aus rein schreibtechnischen Gründen wird statt "P {op} Q" die Notation "op NBL:P EFFECTS:Q" verwendet. P und Q sind prädikatenlogische Formeln über die Eingabeparameter und die Komponenten der abstrakten Repräsentation bzw. die Eingabe-, Ausgabeparameter und die Komponenten der Abstrakten Repräsentation.

Im EFFECTS-Teil benötigt man häufig Beziehungen zwischen dem Zustand von Komponenten der Abstrakten Repräsentation vor und nach der Ausführung der Operation. Sei aki

eine Komponente der Abstrakten Repräsentation. Im EFFECTS-Teil bezeichnet aki den Wert der Komponente nach der Ausführung und `aki deren Wert vor der Ausführung der Operation.

Beispiel

An einem einfachen Beispiel seien die bisher eingeführten Konstrukte erläutert. Hier werden ein Pufferpool der Länge "n" und zwei Operationen "get" und "put" zum Belegen und Freigeben der Puffer spezifiziert. Die Operation get (Belegen) kann nur ausgeführt werden, wenn ein Puffer frei ist, ein Freigeben (put) kann nur erfolgen, wenn der entsprechende Puffer belegt ist.

Der Typ "buffer" sei hier importiert, d.h. außerhalb definiert, da dessen Struktur hier nicht relevant ist. Die Spezifikation stellt sich nun wie folgt dar:

```
MODULE bufferpool
SPECIFICATION

DECLARATIONS
ABSTRACT REPRESENTATION
bfrei:      SET OF buffer INITIALLY {b1,...,bn};
bbelegt:    SET OF buffer INITIALLY  0;
PARAMETERS
n                 : INTEGER
b1,..,bn,a        : buffer

OPERATIONS

get () ---> a;
NBL:        CARD(bfrei) ≠ 0
EFFECTS:    ∃ b ∈ `bfrei (bfrei = `bfrei -{b} ∧
                          bbelegt = `bbelegt∪{b}∧a = b)

put(a);
NBL:        a ∈ bbelegt
EFFECTS:    bbelegt = `bbelegt-{a} ∧
            bfrei   = 'bfrei∪{a}

MODULEEND
```

Abb. 6-5: Modul "bufferpool"

Der Pufferpool wird repräsentiert durch zwei Mengen, die Menge der freien Puffer "bfrei" und der belegten Puffer "bbelegt". Anfänglich sind alle Puffer frei, d.h. bfrei = {b1,...,bn} und bbelegt = 0.

Die Operation get kann nur ausgeführt werden, wenn ein Puffer frei ist, d.h. CARD(bfrei) ≠ 0. Sie nimmt irgendeinen der freien Puffer (∃ b ∈ `bfrei) aus der Menge der freien Puffer (bfrei = `bfrei -{b}) und kennzeichnet ihn als

belegt (bbelegt = `bbelegt $\cup$ {b}). Der Name dieses Puffers wird ausgegeben (a=b).

Mit der Operation put kann ein Puffer nur freigegeben werden, wenn er belegt ist (a $\in$ bbelegt). Dieser Puffer wird aus bbelegt genommen und in bfrei eingereiht.

In den bisherigen Ausführungen wurde davon ausgegangen, daß alle Operationen auf ein ADT-Objekt streng sequentiell ablaufen. Abstrahiert man von dieser Einschränkung, so muß analog zu den Ausführungsbeschränkungen des formalen Modells ein Synchronisationsteil in der Spezifikation eingeführt werden, der im folgenden beschrieben werden soll.

Synchronisationsteil

Hier werden die Verträglichkeiten und Prioritäten zwischen den verschiedenen Operationen auf das abstrakte Objekt spezifiziert. Die durch

VTGL $\subseteq \mathcal{OL} \times \mathcal{P}(\mathcal{OL}) \times \Omega$ und
PRIO $\subseteq \mathcal{OL} \times \mathcal{OL} \times \Omega$

definierten Ausführungsbeschränkungen des formalen Modells werden durch die entsprechenden Sprachkonstrukte COMP ("compatibility" = Verträglichkeit) und PRIO wiedergegeben.

Ganz entscheidend für diese Spezifikation ist der Begriff der "Aktivität". Der Sinn dieser Definition liegt darin, mit Hilfe des Aktivitätsbegriffs eine Spezifikation der Synchronisation eines abstrakten Objekts unabhängig von der Art und Anzahl der Prozesse, die Operationen auf dieses Objekt ausführen, zu ermöglichen. In dieser Arbeit wurde der Aktivitätsbegriff allerdings bereits im formalen Modell eingeführt, was wiederum einen Beweis für die direkte Entsprechung von Modell und Spezifikationsmethode darstellt.

Eine Aktivität eines ADT`s $\tau = (\Omega, T, B, \omega_o)$ wurde definiert als ein Element der Menge

$$\mathcal{A} = \{(t, \pi) \mid \exists j \; (t = t_j \in T \wedge \pi \in E_j)\},$$

d.h. sie besteht aus einer Operation auf Objekte des Typs mit zugehörigen aktuellen Eingabeparametern.

Für die Spezifikation wird nun analog zu <Ker82> eine spezielle Notation von Aktivitätenmengen eingeführt. Man stellt die Menge aller Aktivitäten, die zu einer bestimmten abstrakten Operation des ADT`s gehören, durch den Operationsnamen in Großbuchstaben dar. Eine feinere Zerlegung dieser Aktivitätenmenge ist durch Einbeziehung der Operationsparameter möglich. Sei z.B. eine Operation op(x1,x2) mit zwei formalen Eingabeparametern x1 und x2 gegeben, so bezeichnet OP die Menge aller möglichen Aktivitäten, die zur Operation op gehören, während OP(x1) diejenigen Aktivitäten bezeichnet, deren erster Eingabeparameter den Wert x1 besitzt. Analog dazu ergibt sich OP(x1,x2). Daraus läßt sich folgern

$$OP = \bigcup_{x1} OP(x1) = \bigcup_{x2} OP(x2) = \bigcup_{x1} \bigcup_{x2} OP(x1,x2).$$

Die Angabe der Verträglichkeits- und Prioritätsrelationen in der Spezifikation geschieht nun durch Prädikate über diese Aktivitätenmengen, wobei auch noch der Zustand des Objekts mit einbezogen werden kann. Allgemein gibt es für die Angabe der Relation COMP im SYN-Teil vier Möglichkeiten (vgl. <Mac83>):

a) COMP(x,Y): PY(x,Y,ω)

b) COMP(x,y): Py(x,y,ω)

c) NCOMP(x,Y): PY(x,Y,ω)

d) NCOMP(x,y): Py(x,y,ω)

wobei $x,y \in \mathfrak{A}$, $Y \subseteq \mathfrak{A}$, $PY \subseteq \mathfrak{A} \times \mathfrak{P}(\mathfrak{A}) \times \Omega$, $Py \subseteq \mathfrak{A} \times \mathfrak{A} \times \Omega$

Wird im SYN-Teil keine dieser Möglichkeiten angegeben, oder fehlt der SYN-Teil, so bedeutet dies, daß alle Operationen untereinander verträglich sein sollen. Die allgemeinen Formen a) und c), d.h. die Definition von Verträglichkeiten einer Aktivität mit einer Menge von Aktivitäten, sind so zu interpretieren, daß alle nicht angegebenen Kombinationen nicht verträglich (bzw. bei c) verträglich) sind. Die Differenzierung zwischen a) und b) bzw. c) und d) ergibt sich aus der Tatsache, daß aus einer paarweisen Verträglichkeit mehrerer Operationen nicht unbedingt deren gemeinsame Verträglichkeit folgen muß (vgl. 5.3/5.4). Soll diese "Transitivität" gelten, so wird die Verträglichkeitsrelation in der Form b) angegeben. Auch bei der Definition über NCOMP kann dann auf das einfachere Prädikat d) zurückgegriffen werden. Die Formen a) und c) werden nur notwendig, wenn die Transitivität nicht automatisch erfüllt sein soll.

Die Unterscheidung zwischen einer Angabe als Verträglichkeits- oder Nichtverträglichkeitsbedingung in der Spezifikation wurde deshalb eingeführt, weil es in der Praxis oftmals günstiger ist, die Verträglichkeit von Aktivitäten über Nichtverträglichkeitsprädikate zu spezifizieren. Hier sind dann automatisch alle nicht angegebenen Kombinationen miteinander verträglich.

Analog zur Verträglichkeitsrelation läßt sich dann die Prioritätsrelation angeben als:

PRIO(x,y) : Py(x,y,ω).

Eine Erläuterung dieser Synchronisationskonstrukte sei am Beispiel des Pufferpools aus Abbildung 6-5 gegeben.

Beispiel

Es werden hier noch zwei zusätzliche Operationen "read" und "write" eingeführt, die das Lesen/Schreiben eines Puffers darstellen sollen. Der Typ "buffer" sei nun folgendermaßen definiert:

```
TYPE buffer :  RECORD
               name:       string;
               cont:       information;
               END;
```

Jeder Puffer wird also über einen Namen identifiziert und enthält "Nutzinformation" vom Typ "information" in seiner Komponente "cont". Die Namen seien durch voneinander verschiedene strings vorbelegt, die Nutzinformation sei anfänglich leer.

```
MODULE bufferpool

SPECIFICATION

DECLARATIONS
ABSTRACT REPRESENTATION
bfrei:      SET OF buffer INITIALLY {b1,..,bn} ∧
            ∀ bi ∈ {b1,..,bn} (bi.cont = 0) ∧
            ∀ bi,bj ∈ {b1,...,bn} (bi ≠ bj -->
                                   bi.name ≠ bj.name);
bbelegt:    SET OF buffer INITIALLY 0;

PARAMETERS
n           : INTEGER
b1,..,bn,b  : buffer
a           : string
inf         : information

SYN
COMP ................
PRIO ................

OPERATIONS

get ( ) ---> a;
NBL:        CARD(bfrei) ≠ 0
EFFECTS:    ∃ b ∈ `bfrei (bfrei = `bfrei - {b} ∧
            bbelegt = `bbelegt ∪ {b} ∧ a = b.name)

put(a);
NBL:        ∃ b ∈ bbelegt (b.name = a)
EFFECTS:    bbelegt = `bbelegt - {b} ∧
            bfrei = `bfrei ∪ {b}

read (a) ---> inf;
NBL:        ∃ b ∈ bbelegt (b.name = a)
EFFECTS:    inf = b.cont
```

```
write (a,inf);
NBL:        ∃ b ∈ bbelegt (b.name = a)
EFFECTS:    b.cont = inf

MODULEEND
```

Abb. 6-6: Erweiterter Modul "bufferpool"

Anhand dieses Beispiels lassen sich nun verschiedene Synchronisations- und Schedulingstrategien durch die Angabe unterschiedlicher COMP- und PRIO-Konstrukte spezifizieren. Man betrachte zunächst das COMP-Konstrukt.

Sollen alle möglichen Aktivitäten unter gegenseitigem Ausschluß stattfinden, so kann dies durch die Angaben

COMP(x,Y): false

COMP(x,y): false

NCOMP(x,Y): $x \in GET \cup PUT \cup READ \cup WRITE \ \wedge$
$Y \subseteq GET \cup PUT \cup READ \cup WRITE$

NCOMP(x,y): $x,y \in GET \cup PUT \cup READ \cup WRITE$

festgelegt werden, wobei alle vier Möglichkeiten völlig äquivalent sind.

Eine andere Strategie wäre z.B., beliebig viele READ-Aktivitäten gleichzeitig zuzulassen, wobei diese jedoch mit anderen Aktivitäten und auch alle anderen Aktivitäten untereinander nicht verträglich sein sollen. Dies stellt sich einfach dar durch

COMP(x,y): x,y ∈ READ oder

COMP(x,y): x ∈ READ ∧ y ∈ READ oder

COMP(x,y): x ∈ READ(a1) ∧ y ∈ READ(a2)

d.h. man definiert am besten über das COMP-Konstrukt.

Eine kompliziertere Strategie wäre z.B., zuzulassen, daß get-Aktivitäten und put-Aktivitäten mit sich selbst und allen anderen Aktivitäten nur unter gegenseitigem Ausschluß ausgeführt werden dürfen, aber

- read- und write-Aktivitäten gemeinsam stattfinden dürfen, wenn sie sich nicht auf den gleichen Puffer beziehen
- zwei oder mehrere write-Aktivitäten gemeinsam stattfinden dürfen, wenn sie sich nicht auf den gleichen Puffer beziehen
- zwei oder mehrere read-Aktivitäten immer gleichzeitig ausgeführt werden dürfen.

Da hier die Transitivität gelten soll, kann man die Angabe der Spezifikation über die einfacheren Konstrukte, die sich nur auf die Beziehungen zwischen zwei Aktivitäten beziehen, vornehmen. Auch hier erweist sich eine Spezifikation über das COMP-Konstrukt als günstiger. Man kann somit definieren

COMP(x,y): (x ∈ READ(a1) ∪ WRITE(a1) ∧
y ∈ READ(a2) ∪ WRITE(a2) ∧ a1 ≠ a2) ∨
(x,y ∈ READ)

Da die Fälle innerhalb x,y ∈ READ, in denen die gelesenen Puffer voneinander verschieden sind, schon im ersten Teil des Prädikats (durch a1 ≠ a2) enthalten sind, kann, um

diese Redundanz zu vermeiden, statt

$x,y \in READ$ auch

$x,y \in READ(a)$ oder

$x \in READ(a1) \wedge y \in READ(a2) \wedge a1 = a2$

geschrieben werden.

An den gezeigten Beispielen für die Formulierung von Synchronisationsstrategien fällt besonders auf, daß die formale Spezifikation der Synchronisation wesentlich einfacher und natürlicher ist, als die entsprechende sprachliche Darstellung. Insbesondere lassen sich Modifikationen in der Synchronisationsstrategie sehr einfach durchführen.

Analog zu Synchronisationsstrategien lassen sich nun auch Schedulingstrategien spezifizieren. Hierfür seien nur zwei kleine Beispiele gegeben. So drückt z.B.

$PRIO(x,y): \quad x \in PUT \wedge y \in GET$

aus, daß Freigabeaktivitäten höhere Priorität als Belegungsaktivitäten besitzen sollen, oder

$$PRIO(x,y): \quad (x \in PUT \wedge y \in GET \cup READ \cup WRITE) \vee (x \in READ \wedge y \in WRITE)$$

legt fest, daß eine Freigabeaktivität absolute Priorität vor allen anderen Aktivitäten besitzt, und innerhalb der anderen Aktivitäten eine READ-Aktivität höherprior als eine WRITE-Aktivität sein soll.

Mit diesen Beispielen soll der Abschnitt 6.2.1, der die Spezifikation von ADTs ohne Schutzkonstrukte erläutern sollte, beschlossen werden. Weitere Beispiele für Spezifikationen finden sich in <Ker82>, <Mac83> oder <Rei83>.

Hier sollte die Spezifikationsmethode lediglich deshalb nochmals präsentiert werden, um dem Leser ein Nachschlagen in der o.g. Literatur zu ersparen, und andererseits ein gewisses Vor-Verständnis für die im nächsten Abschnitt einzuführenden Schutzspezifikationen herzustellen.

6.2.2 Die Spezifikation von Schutzeinschränkungen und policies in ADTs

Den in dieser Arbeit für das formale Modell definierten Schutzkonstrukten sollen nun Sprachkonstrukte zur Formulierung von Schutzeinschränkungen in der Spezifikation zugeordnet werden. Ausgangspunkt hierfür ist die formale Definition von abstrakten Datentypen mit policies, d.h. das Tupel $\tau = (\Omega, T, B, Pol, \omega o)$. Zusätzlich zu Abstrakten Datentypen ohne Schutzkonstrukte wurden hier folgende Komponenten definiert:

B ist hier ein Tripel B = (REJ,VTGL,PRIO), wobei für die Zurückweisungsrelation REJ gilt REJ $\subseteq (\mathcal{OL} \times \Omega) \times \mathcal{F}$, d.h. REJ definiert die Aktivitäten, die bei den angegebenen Zuständen zurückgewiesen werden und ordnet diesen Aktivitäten Fehlermeldungen aus $\mathcal{F}$ zu.

Pol ist ein Quadrupel von policies

POL = (NACC,NREF,NMOD,NFLOW) mit

NACC	$\subseteq \mathcal{OL} \times \Omega$	ist Nichtzugriffspolicy
NREF	$\subseteq \mathcal{OL} \times \Omega \times$ Ind	ist Nichtabfragepolicy
NMOD	$\subseteq \mathcal{OL} \times \Omega \times$ Ind	ist Nichtveränderungspolicy
NFLOW	$\subseteq \mathcal{OL} \times \mathcal{OL} \times$ PRÄD	ist Informationsflußpolicy

Hiermit ergibt sich die vollständige Darstellung der Spezifikation eines ADT`s als

```
SPECIFICATION

DECLARATIONS
    ABSTRACT REPRESENTATION ... INITIALLY
    PARAMETERS
    LOCAL TYPES

POLICIES

    NACC
    NREF
    NMOD
    NFLOW

SYN

    COMP
    PRIO

OPERATIONS

    REJ
    NBL
    EFFECTS
```

Im folgenden soll nun die allgemeine Form der eingeführten Schutzkonstrukte für die Spezifikation präsentiert und anhand von Beispielen ihre Anwendung erläutert werden.

6.2.2.1 Erweiterung des Operationsteils um Schutzeinschränkungen

Die Zurückweisungsrelation REJ wird in der Spezifikation im Operationsteil angegeben, da sie sich auf die Zurückweisung einzelner Aktivitäten bezieht und im Prinzip ebenfalls eine Precondition für die Operation - analog zu

NBL - ist, wobei hier allerdings die Ausführung nicht blockiert, sondern zurückgewiesen wird. Die Aktivitäten und Objektzustände werden hier wie bei der NBL-Bedingung durch Prädikate charakterisiert, wobei noch die entsprechenden Fehlermeldungen angegeben werden.

Im formalen Modell stellt sich REJ dar als REJ $\subseteq (\mathcal{A} \times \Omega) \times \mathcal{F}$. Dem entspricht auf spezifikationssprachlicher Ebene für jede Operation die Form

REJ: $\bigwedge_j$ Pj (ak1,..,akp,x1,..,xm) ERROR Fj (y1,..,yn)

Hier sind die ak1,..,akp wieder die Komponenten der Abstrakten Repräsentation, x1,..,xm die Eingabe- und y1,..,yn die Ausgabeparameter der entsprechenden Operation. Die Pj sind beliebige Prädikate erster Ordnung, die die Zurückweisungsfälle kennzeichnen, und die Fj charakterisieren die Werte der Ausgabevariablen im Fehlerfall. Die Konjunktion sieht dabei eine Unterscheidung zwischen mehreren Fehlerfällen vor.

Man beachte, daß in den Prädikaten Fj keine Verknüpfung mit den Komponenten der Abstrakten Repräsentation oder den Eingabeparametern stattfindet, d.h. die Ausgabe besteht aus Konstanten. Dies soll unerlaubte Informationsflüsse im Zurückweisungsfall verhindern. Natürlich kann hier immer noch von der ausgegebenen Fehlermeldung auf die entsprechenden Werte der im jeweiligen Prädikat Pj enthaltenen Größen geschlossen werden. Diese Art von "Informationsfluß" soll jedoch hier nicht verhindert werden, obwohl dies durch eine Nicht-Differenzierung der Fehlermeldungen natürlich prinzipiell leicht möglich wäre. Der Grund für das Bestehenlassen dieses "Informationsflusses" liegt in der praktischen Einsetzbarkeit der Methode: Dem Benutzer einer Operation eines ADT`s sollte im Fehlerfall auf jeden Fall mitgeteilt werden, warum sein Zugriff abgewiesen wurde.

Insgesamt stellt sich somit der Operationsteil auf spezifikationssprachlicher Ebene folgendermaßen dar

```
OPERATIONS

op (x1,..,xm) ---> y1,..,yn

REJ:     P1 (ak1,..,akp,x1,..,xm) ERROR F1 (y1,..,yn)
...      ...                      ...
REJ:     Ps (ak1,..,akp,x1,..,xm) ERROR Fs (y1,..,yn)
NBL:     P (ak1,..,akp,x1,..,xm)
EFFECTS: Q (ak1,..,akp,`ak1,..,`akp,x1,..,xm,y1,..,yn)

...
```

Im folgenden seien nun die präsentierten Sprachkonstrukte für die Zurückweisungsrelation an einem Beispiel erläutert.

Beispiel

Ausgehend vom Pufferpool aus Abbildung 6-6 werden hier einige Modifikationen eingeführt, um an diesem Beispiel die Schutzkonstrukte auf spezifikationssprachlicher Ebene einfacher erläutern zu können. Zunächst wird die Abstrakte Repräsentation dahingehend modifiziert, daß klar zwischen dem eigentlichen Puffer und den für Schutzaspekte notwendigen Verwaltungslisten getrennt wird. Diese Trennung ist besonders für eine eindeutige Informationsflußanalyse von Bedeutung. Weiterhin wird eine Vereinfachung der Darstellung dadurch eingeführt, daß die einzelnen Puffer nicht mehr über einen Namen, sondern über ihren Index im Feld der Puffer identifiziert werden.

Zusätzlich zu diesen Modifikationen der Abstrakten Repräsentation werden hier die Operationen erweitert. In Abbildung 6-6 waren die vier Operationen get (Belegen eines Puffers), put (Freigabe), read (Lesen) und write (schreiben) so definiert, daß entweder mindestens ein Puffer frei (get),

oder der entsprechende Puffer belegt sein muß (put, read, write). Es wurde allerdings nicht berücksichtigt, daß z.B. ein Benutzer des Pufferpools nur auf eigene Puffer, d.h. auf solche, die er zuvor belegt hatte, zugreifen sollte. Diese Einschränkung erfordert eine Identifikation des Benutzers beim Operationsaufruf und ein Merken des jeweiligen "Besitzers" bei den einzelnen Puffern des Pools.

Prinzipiell könnte diese Erweiterung natürlich auch durch eine entsprechende Wahl der NBL-Bedingungen realisiert werden, jedoch wird dann bei unerlaubten Zugriffen der aufrufende Prozeß immer blockiert und kann nie mehr fortgesetzt werden. In solchen Fällen ist also eine Zurückweisung wesentlich adäquater. Analog trifft dies übrigens auch auf die bisherige Blockierung bei put-, read- und write-Zugriffen zu, wenn der entsprechende Puffer nicht belegt ist.

Für die erweiterte und modifizierte Aufgabenstellung benötigt man nun in der Abstrakten Repräsentation zwei Felder "buffer" und "bufowner" der Länge "n". Das Feld "bufowner" repräsentiert dabei die Verwaltungsliste der Puffer, die durch das Feld "buffer" dargestellt werden.

Das Feld "bufowner" bestehe hier einfach aus der Identifikation der Besitzer der Puffer, die vom Typ "user-id" seien. Der Typ "user-id" sei hierbei die im System übliche Benutzeridentifikation und muß hier nicht näher spezifiziert werden. Zusätzlich sei in den Ausgabeparametern der Operationen noch eine Variable vorgesehen, die bei Zurückweisung den Fehlercode beinhaltet. Auf die Synchronisation der Operationen wird in diesem Beispiel nicht näher eingegangen, es können aber alle für die Abbildung 6-6 angegebenen möglichen Strategien verwendet werden.

```
MODULE bufferpool

SPECIFICATION

DECLARATIONS
ABSTRACT REPRESENTATION
bufowner:   ARRAY [1..n] OF user-id
            INITIALLY ∀ i ε {1,..,n} (bufowner[i] = λ);
buffer:     ARRAY [1..n] OF information
            INITIALLY ∀ i ε {1,..,n} (buffer[i] = λ);

PARAMETERS
n           : INTEGER
i,j,index   : (1..n)
f           : STRING
inf         : information
uid         : user-id

SYN
    COMP .....
    PRIO .....

OPERATIONS

get (uid) ---> index;
NBL:        ∃ i ε {1,..,n} (bufowner[i] = λ)
EFFECTS: bufowner[i] = uid ∧ index = i

put (uid,index) ---> f;
REJ:        bufowner[index] = λ  ERROR f = "no such buffer";
            bufowner[index] ≠ uid ERROR f = "no permission";
EFFECTS: bufowner[index] = λ ∧ f = λ
```

```
read (uid,index) ---> (inf,f);
REJ:     .....
EFFECTS: inf = buffer[index] ∧ f = λ

write (uid,index,inf) ---> f;
REJ:     .....
EFFECTS: buffer[index] = inf ∧ f = λ
```

Abb. 6-7: Modul "bufferpool" mit Schutzeinschränkungen

Durch die entsprechenden Fehlerfälle in den REJ-Bedingungen werden hier also Zugriffe, die auf einen nicht vorhandenen Puffer oder auf einen Puffer eines anderen Benutzers ausgeführt werden sollen, zurückgewiesen. Abschließend sei zu diesem Beispiel noch bemerkt, daß hier, obwohl kein Benutzer auf einen fremden Puffer zugreifen kann, Information zwischen Benutzern fließen kann. Dies liegt daran, daß beim Freigeben eines Puffers die enthaltene Information nicht gelöscht wird, und somit vom nächsten "Besitzer" des Puffers gelesen werden könnte. Eine policy NFLOW(write,(uid1,index,inf),read,(uid2,index),uid1≠uid2) wäre also hier nicht erfüllt.

Im folgenden werde nun die Frage, wann eine policy durch die gegebenen Operationen und Zurückweisungsbedingungen erfüllt ist, behandelt.

6.2.2.2 Die Spezifikation von policies

In diesem Abschnitt sollen zuerst die Sprachkonstrukte zur Spezifikation von policies präsentiert werden und dann das allgemeine Vorgehen bei der Schutzspezifikation dargestellt werden. Anhand von Beispielen werden anschließend die eingeführten Konstrukte und ihre Verwendung erläutert.

Die policies werden im Spezifikationsteil innerhalb des Konstrukts POLICIES definiert. Analog zum formalen Modell unterscheidet man zwischen Nichtzugriffs-, Nichtabfrage-, Nichtveränderungs- und Informationsflußpolicies. Wie im Kapitel 5 dieser Arbeit ausführlich erläutert und begründet wurde, sind die policies im vorliegenden Modell und damit auch in der Spezifikation als Zusicherungen zu betrachten, deren Einhaltung bewiesen werden muß. Sie können also nicht direkt von einem Laufzeit-OV-Mechanismus durchgesetzt werden, sondern nur indirekt über die Zurückweisungsrelation REJ.

Man betrachte zunächst die Nichtzugriffspolicy NACC, die angibt, bei welchen Objektzuständen mit einer Aktivität nicht auf das Objekt zugegriffen werden darf. Im formalen Modell ist sie als NACC $\subseteq \mathcal{A} \times \Omega$ gegeben, d.h. für alle $(x, \omega) \in$ NACC darf mit der Aktivität x beim Objektzustand ω nicht zugegriffen werden.

Spezifikationssprachlich wird NACC durch Prädikate über Aktivitäten und Objektzustände dargestellt. Die einzelnen Prädikate, die die Aktivitäten charakterisieren, beziehen sich hier ebenso wie bei den Nichtabfrage- und Nichtveränderungspolicies jeweils auf eine Operation. Aus diesem Grund ist es sinnvoll, im POLICIES-Teil der Spezifikation diese drei policies operationsspezifisch zusammengefaßt darzustellen.

Die Nichtabfrage- und Nichtveränderungspolicies NREF bzw. NMOD sichern zu, daß bei bestimmten Objektzuständen von einer Aktivität gewisse Objektkomponenten nicht gelesen oder verändert werden können. Die formale Definition dieser policies ist gegeben durch

NREF $\subseteq \mathcal{A} \times \Omega \times$ Ind und
NMOD $\subseteq \mathcal{A} \times \Omega \times$ Ind

wobei Ind die Menge der Indizes der abstrakten Objektkomponenten ist. Jedes Tripel $(x, \omega, i) \in$ NREF/NMOD gibt dabei

an, daß mit der Aktivität x im Objektzustand ω die i-te Objektkomponente nicht abgefragt/verändert werden darf.

Auch hier werden die entsprechenden Kombinationen spezifikationssprachlich über Prädikate definiert. Die allgemeine Form der zusammengefaßten policies NACC, NREF und NMOD stellt sich dann auf spezifikationssprachlicher Ebene dar als

POLICIES

op1: NACC : P1 (ak1,..,akp,x11,..,x1m1)
NREF : $\bigwedge_j$ P1j (ak1,..,akp,x11,..,x1m1)
NREF T1j ({ak1,..,akp})
NMOD : $\bigwedge_j$ P1j` (ak1,..,akp,x11,..,x1m1)
NMOD T1j` ({ak1,..,akp})

.....

opn: NACC : Pn (ak1,..,akp,xn1,..,xnmn)
NREF : $\bigwedge_j$ Pnj (ak1,..,akp,xn1,..,xnmn)
NREF Tnj ({ak1,..,akp})
NMOD : $\bigwedge_j$ Pnj` (ak1,..,akp,xn1,..,xnmn)
NMOD Tnj` ({ak1,..,akp})

wobei die Tj({ak1,..,akp}) Teilmengen der Menge der Objektkomponenten sind.

Die Informationsflußpolicy NFLOW bezieht sich im Gegensatz zu den bisherigen policies auf die Interaktion zwischen jeweils zwei Aktivitäten. Sie legt fest, unter welchen Voraussetzungen Information von einer Aktivität zu einer anderen fließen darf. Im formalen Modell ist sie als NFLOW $\subseteq \mathcal{A} \times \mathcal{A} \times$ PRÄD definiert, wobei PRÄD die Menge aller möglichen Prädikate über Berechnungen der dem ADT zugeordneten OV-Maschine ist. Diese dynamische policy, die im allgemeinen Fall die möglichen Berechnungen der OV-Maschine mit einbezieht, enthält dann Tupel (x,y,Φ), die bedeuten, daß

in einer Berechnung, die dem Prädikat Φ genügt, keine Information von der Aktivität x zur Aktivität y fließen darf.

Spezifikationssprachlich stellt sich die policy NFLOW dar als

$$\text{NFLOW}(x,y): \bigvee_j (Pj1(x) \wedge Pj2(y) \wedge \Phi j\langle qi,ei,ai\rangle)$$

wobei x,y Aktivitäten sind und $\langle qi,ei,ai\rangle$ eine Berechnung der zugeordneten OV-Maschine. Hierzu sei noch bemerkt, daß in der Praxis sicherlich Prädikate über Berechnungen relativ selten vorkommen werden, und stattdessen häufiger einfache Prädikate über die beiden beteiligten Aktivitäten und die jeweiligen Objektzustände. Für den allgemeinen Fall müssen jedoch Prädikate über Berechnungen berücksichtigt werden, um z.B. history dependent policies formulieren zu können.

Nach dieser Darstellung der allgemeinen Form der policies auf spezifikationssprachlicher Ebene sei hier nun kurz das allgemeine Vorgehen bei der Schutzspezifikation in ADTs erläutert:

Grundsätzlich lassen sich hierbei zwei mögliche Vorgehensweisen unterscheiden - eine analytische und eine synthetische. Bei der analytischen Vorgehensweise werden ausgehend von gegebenen Operationen und einer gegebenen Zurückweisungsrelation die hierdurch erfüllten policies ermittelt. Dem gegenüber steht das synthetische Vorgehen, wo man von Operationen und zu erfüllenden policies ausgeht und die zugehörige Zurückweisungsrelation ermittelt. In der Praxis werden diese beiden Vorgehensweisen nun insofern kombiniert, daß im allgemeinen zunächst aufgrund der gewüschten policies die Zurückweisungsrelation bestimmt, d.h. synthetisiert wird, und anschließend bei der Verifikation der Durchsetzung dieser policies eine Analyse der durch die Zurückweisungsrelation erfüllten policies durchgeführt wird. Natürlich ist grundsätzlich auch das umgekehrte Vorgehen denkbar, d.h. man analysiert die durch eine Zurückweisungsrelation

durchgesetzten policies, vergleicht sie mit erwünschten policies und führt bei Nichtübereinstimmung die Synthese einer erweiterten Zurückweisungsrelation durch.

Zur Erläuterung der eingeführten policies und der beiden Vorgehensweisen für die Erstellung von Schutzspezifikationen seien nun im folgenden zwei Beispiele präsentiert.

Beispiel 1

Im ersten Beispiel zur Spezifikation von policies sei untersucht, welche policies vom in Abbildung 6-7 spezifizierten Pufferpool erfüllt werden. Es handelt sich hier also um eine analytische Vorgehensweise. Man betrachte zunächst die policies für eine Aktivität.

Hier ergibt sich für jede Operation die Nichtzugriffspolicy NACC, die von der Operation zusammen mit der Zurückweisungsrelation REJ durchgesetzt wird, sehr einfach aus der Disjunktion aller in REJ enthaltenen Prädikate Pi und der Negation des NBL-Prädikats der Operation.

Im Gegensatz dazu ergeben sich die Nichtabfrage- und Nichtveränderungspolicy der Operationen aus deren EFFECTS-Teil. Man kann die Nichtabfragepolicy NREF als Zusicherung zum Schutz vor unerlaubter Enthüllung, d.h. als privacy-policy, interpretieren, während die Nichtveränderungspolicy NMOD als Zusicherung zum Schutz vor unerlaubter Modifikation oder als integrity-policy betrachtet werden kann.

Man beachte, daß in den Fällen, in denen die Aktivität zurückgewiesen oder blockiert wird, trivialerweise Objektkomponenten weder gelesen noch verändert werden können. Da diese Fälle bereits durch die Nichtzugriffspolicy NACC charakterisiert wurden, sind sie für die Ermittlung der Nichtabfragepolicies und Nichtveränderungspolicies nicht von

Interesse. Man beschränkt sich deshalb, um Redundanzen zu vermeiden, auf Zusicherungen, die sich auf Fälle beziehen, bei denen die Aktivität ausgeführt werden kann. Diese Zusicherungen können dann aus dem EFFECTS-Teil der jeweiligen Operation ermittelt werden.

Man betrachte nun die einzelnen Operationen des ADT`s. Für die Operation get existiert keine Zurückweisungsrelation REJ, d.h. die Nichtzugriffspolicy NACC ergibt sich einfach aus der Negation des NBL-Prädikats. Als Nichtabfragepolicy NREF ergibt sich hier, daß vom EFFECTS-Teil der Operation weder das Feld "bufowner" noch "buffer" abgefragt werden. Die Nichtveränderungspolicy NMOD sichert zu, daß im Feld "buffer" durch die Operation get nichts verändert werden kann. Weiterhin kann zugesichert werden, daß ein bereits belegter Puffer ($bufowner[i] \neq \lambda$) nicht nochmals belegt werden kann. Spezifikationssprachlich läßt sich dies folgendermaßen formulieren:

POLICIES

get (uid): NACC : $\neg \exists i \in \{1,..,n\}\ (bufowner[i] = \lambda)$

NREF : true NREF bufowner $\cup$ buffer

NMOD : true NMOD buffer $\cup$ $\{bufowner[i] \mid bufowner[i] \neq \lambda\}$

Analog lassen sich für die Operationen put, read und write folgende policies definieren:

put (uid,index): NACC : bufowner[index] = λ ∨
bufowner[index] ≠ uid
NREF : true NREF bufowner ∪ buffer
NMOD : true NMOD buffer ∪
{bufowner[i] | i ≠ index}

read (uid,index): NACC : bufowner[index] = λ ∨
bufowner[index] ≠ uid
NREF : true NREF bufowner ∪
{buffer[i] | i ≠ index}
NMOD : true NMOD bufowner ∪ buffer

write(uid,index,inf): NACC : bufowner[index] = λ ∨
bufowner[index] ≠ uid
NREF : true NREF bufowner ∪ buffer
NMOD : true NMOD bufowner ∪
{buffer[i] | i ≠ index}

Für die drei Operationen put, read und write wird durch die Nichtzugriffspolicy NACC zugesichert, daß vom Aufrufer nur auf Puffer, die er selbst besitzt, zugegriffen werden kann. Die Nichtabfragepolicy von put sichert zu, daß keine Komponente der Abstrakten Repräsentation von put abgefragt werden kann, während durch die Nichtveränderungspolicy ausgedrückt wird, daß nur in der durch die Eingabevariable "index" bestimmten Komponente der Pufferverwaltungsliste etwas verändert werden kann. Völlig analog sind die entsprechenden policies für die beiden anderen Operationen zu interpretieren.

Betrachtet man den Informationsfluß zwischen den Aktivitäten, so ist sicherlich zu fordern, daß weder die in "bufowner" gespeicherten user-ids noch die eigentlichen Nutzinformationen in "buffer" von einer Aktivität eines Benutzers zu einer eines anderen Benutzers gelangen. Da das Feld "bufowner" von keiner Aktivität abgefragt wird, und die Abfrage eine notwendige Voraussetzung für den Informationsfluß ist, fließt über "bufowner" sicher keine Information

zwischen zwei Aktivitäten verschiedener Benutzer.

Anders verhält es sich mit dem Feld "buffer", das die eigentlichen Informationen enthält. Hierauf wird mit den Operationen "read" lesend und mit "write" verändernd zugegriffen. Da bei einer beliebigen read- oder write-Aktivität auf den Puffer immer nur auf den dem jeweiligen Aufrufer gehörenden Pufferplatz lesend oder verändernd zugegriffen werden kann, kann Information zwischen zwei Aktivitäten verschiedener Benutzer nur dadurch fließen, daß sie von einer write-Aktivität in einen bestimmten Pufferplatz geschrieben wird und aus demselben Pufferplatz später von einer read-Aktivität eines anderen Benutzers gelesen wird. Es muß also eine Berechnung der zugeordneten OV-Maschine geben, in der zuerst die Aktivität "write(uid1,index,inf)" und später die Aktivität "read(uid2,index)" mit uid1 $\neq$ uid2 stattfindet und dazwischen keine Aktivität die Komponente buffer[index] verändert.

Eine solche Berechnung existiert nun tatsächlich: Gibt nämlich der Benutzer uid1 nach der write-Aktivität mit "put(uid1,index)" den Puffer frei und bekommt anschließend ein Benutzer uid2 durch einen Aufruf "get(uid2)" denselben Puffer zugewiesen, so könnte er mit "read(uid2,index)" die noch von uid1 enthaltene Information lesen.

Dies bedeutet, daß die policy

NFLOW(x,y): $x \in \text{WRITE}(uid1) \wedge y \in \text{READ}(uid2) \wedge uid1 \neq uid2$

nicht erfüllt wäre. Der Grund hierfür ist, daß beim Freigeben des Puffers durch die Operation put der Pufferinhalt nicht gelöscht, d.h. mit λ überschrieben, wird. Fügt man diese Ergänzung noch in die Definition von put ein, so stellt sich die Operation put dar als

```
put (uid,index) ---> f;
REJ:        bufowner[index] = λ ERROR f = "no such buffer";
            bufowner[index] ≠ uid ERROR f = "no permission";
EFFECTS:    bufowner[index] = λ ∧ buffer[index] = λ ∧ f = λ
```

und der POLICIES-Teil von put verändert sich zu

```
put (uid,index): NACC : bufowner[index] = λ   ∨
                        bufowner[index] ≠ uid
                 NREF : true NREF bufowner ∪ buffer
                 NMOD : true NMOD {bufowner[i] | i ≠ index} ∪
                                  {buffer[i] | i ≠ index}
```

Als Informationsflußpolicy läßt sich dann

```
NFLOW(x,y): [x ∈ WRITE(uid1) ∧ y ∈ READ(uid2) ∧ uid1 ≠ uid2] ∨
            [x ∈ PUT ∧ y ∈ READ ∧ read(...).inf ≠ λ]
```

angeben. Der erste Teil der policy drückt hierbei aus, daß kein Informationsfluß von einer write-Aktivität eines Benutzers zu einer read-Aktivität eines anderen Benutzers stattfinden kann, und zwar völlig unabhängig vom jeweiligen Pufferplatz. Der zweite Teil der policy drückt aus, daß von der Freigabeoperation put zu einer nachfolgenden Leseaktivität nur die leere Information λ (die ja put in den entsprechenden Pufferplatz schrieb) fließen kann.

Beispiel 2

In diesem Beispiel soll der Pufferpool dahingehend erweitert werden, daß eine Kommunikation zwischen verschiedenen Benutzern ermöglicht wird. Als Restriktion sei hier das militärische Sicherheitsmodell vorgesehen, d.h. ein Benutzer darf den Inhalt eines Puffers nur dann lesen, wenn seine Sicherheitsklasse mindestens so hoch wie die des Besitzers des Puffers (der ja die Information in den Puffer schrieb) ist.

Für dieses Beispiel soll das militärische Sicherheitsmodell trotz seiner geringen Bedeutung für die Praxis (vgl. Kapitel 4) benutzt werden. Der Grund hierfür liegt darin, daß sich die synthetische Vorgehensweise, d.h. die Ermittlung der Zurückweisungsrelation REJ aus den gewünschten policies, bei der Umsetzung dieses Modells besonders gut erläutern läßt, da hier ja die durchzusetzende policy von vorne herein gegeben ist.

Als Erweiterung benötigt man bei den Operationen als zusätzlichen Eingabeparameter die Sicherheitsklasse des aufrufenden Benutzers. In der Pufferverwaltungsliste "bufowner" ist zur Verhinderung nicht erlaubter Zugriffe noch ein Feld nötig, in dem pro Pufferplatz die Sicherheitsklasse des Besitzers gemerkt wird.

Bei der Konstruktion des entsprechenden Pufferpools muß man nun von der gewünschten Informationsflußpolicy ausgehen. Informal muß hier gefordert werden, daß von einer write-Aktivität eines Benutzers mit Sicherheitsklasse prot1 zu einer read-Aktivität eines anderen Benutzers mit Sicherheitsklasse prot2 keine Information fließen darf, wenn prot1 > prot2 gilt. Zusätzlich muß natürlich noch gelten, daß von einer put-Aktivität zu einer nachfolgenden read-Aktivität nur die leere Information λ fließen kann. Die entsprechende policy stellt sich dann dar als

$$\begin{aligned} \text{NFLOW}(x,y)\colon\ & [x \in \text{WRITE}(\text{prot1}) \wedge y \in \text{READ}(\text{prot2}) \wedge \text{prot1} > \text{prot2}] \\ & \vee [x \in \text{PUT} \wedge y \in \text{READ} \wedge \text{read}(\ldots).\text{inf} \neq \lambda] \end{aligned}$$

In Abbildung 6-8 wird nun die Spezifikation des entsprechenden Pufferpools, der diese Informationsflußpolicy erfüllt, präsentiert. Man betrachte die einzelnen Operationen mit ihren policies:

Die Operation get verändert sich hier nur dahingehend, daß der zusätzliche Eingabeparameter "prot" in den entsprechenden Platz der Pufferverwaltungsliste "bufowner" einge-

tragen wird. Die policies für get bleiben gleich. Analoges gilt auch für die Operationen put und write.

Wesentlich verändert wird hier durch die zu erfüllende Informationsflußpolicy nur die Zurückweisungsrelation und damit auch die Nichtzugriffspolicy der Operation read. So werden read-Aktivitäten, bei denen die Sicherheitsklasse des Aufrufers kleiner ist, als die des entsprechenden Pufferplatzes, zurückgewiesen. Auf diese Weise ist sichergestellt, daß keine Information von einer write-Aktivität zu einer read-Aktivität mit niedrigerer Sicherheitsklasse fließt, und der erste Teil der oben geforderten Informationsflußpolicy ist erfüllt. Der zweite Teil ergibt sich analog zum vorhergehenden Beispiel.

Nach diesen Beispielen für die Spezifikation von policies im vorgestellten Modell (weitere Beispiele finden sich im Kapitel 7 der Arbeit) soll nun im folgenden Abschnitt die wichtige Frage der Sicherheit eines ADT`s behandelt werden. Zunächst wird dabei auf die Verifikation der Sicherheit der Spezifikation (design verification) eingegangen.

```
MODULE bufferpool

SPECIFICATION

DECLARATIONS
ABSTRACT REPRESENTATION
bufowner:  ARRAY [1..n] OF RECORD
                              owner      : user-id;
                              prot-class : class;
                              END
           INITIALLY ∀ i ε {1,..,n} (bufowner[i] = (λ, λ));
buffer:    ARRAY [1..n] OF information
           INITIALLY ∀ i ε {1,..,n} (buffer[i] = λ);
```

```
PARAMETERS
n            : INTEGER
i,index      : (1..n)
f            : STRING
inf          : information
uid          : user-id
prot,prot1
prot2        : (unbeschränkt,vertraulich,geheim,streng geheim)

POLICIES

get (uid,prot):  NACC : ¬ ∃ i ε {1,..,n}
                        (bufowner[i].owner = λ)
                 NREF : true NREF bufowner ∪ buffer
                 NMOD : true NMOD buffer ∪
                        {bufowner[i] | bufowner[i].owner ≠ λ}

put (uid,index): NACC : bufowner[index].owner = λ  ∨
                        bufowner[index].owner ≠ uid
                 NREF : true NREF bufowner ∪ buffer
                 NMOD : true NMOD {bufowner[i] | i≠ index} ∪
                        {buffer[i] | i ≠ index}

read (prot,index): NACC : bufowner[index].owner = λ  ∨
                          bufowner[index].prot-class > prot
                   NREF : true NREF bufowner ∪
                          {buffer[i] | i ≠ index}
                   NMOD : true NMOD bufowner ∪ buffer

write (uid,index,inf): NACC : bufowner[index].owner = λ  ∨
                              bufowner[index].owner ≠ uid
                       NREF : true NREF bufowner ∪ buffer
                       NMOD : true NMOD bufowner ∪
                              {buffer[i] | i ≠ index}
```

```
NFLOW(x,y): [x ∈ WRITE(prot1) ∧ y ∈ READ(prot2) ∧ prot1 > prot2] ∨
            [x ∈ PUT ∧ y ∈ READ ∧ read(...).inf ≠ λ]

SYN
     COMP .....
     PRIO .....


OPERATIONS

get (uid,prot) ---> index;
NBL:      ∃ i ∈ {1,..,n} (bufowner[i].owner = λ)
EFFECTS: bufowner[i] = (uid,prot) ∧ f = λ

put(uid,index) ---> f;
REJ:      bufowner[index].owner = λ  ERROR f="no such buffer"
          bufowner[index].owner ≠ uid ERROR f="no permission"
EFFECTS: bufowner[index] = (λ,λ) ∧ buffer[index] =λ ∧ f = λ

read (prot,index) ---> (inf,f);
REJ:      bufowner[index].owner = λ  ERROR f="no such buffer"
          bufowner[index].prot-class > prot
          ERROR f="no permission"
EFFECTS: inf = buffer[index] ∧ f = λ

write (uid,index,inf) ---> f;
REJ:      bufowner[index].owner = λ  ERROR f="no such buffer"
          bufowner[index].owner ≠ uid ERROR f="no permission"
EFFECTS: buffer[index] = inf ∧ f = λ
```

Abb. 6-8: Modul "bufferpool" mit militärischer Sicherheit

6.2.3 Bemerkungen zur Verifikation der Sicherheit von Spezifikationen Abstrakter Datentypen

Wie bereits zu Anfang dieses Kapitels erwähnt wurde, zerfällt der Beweis der Sicherheit eines ADT`s in zwei Schritte, die Verifikation der Spezifikation (design verification) und den Nachweis einer korrekten Implementierung. Zunächst soll in diesem Abschnitt der erste Verifikationsschritt betrachtet werden.

Die bei der design verification durchzuführenden Beweise dienen dem Nachweis der Konsistenz und Sicherheit der gegebenen Spezifikationen. Informal ausgedrückt handelt es sich hierbei um den Nachweis, daß die gegebenen policies durch die Operationen mit ihren Zurückweisungsrelationen erfüllt werden und daß die gegebene Verträglichkeitsrelation konsistent ist. Beides ist nämlich Voraussetzung dafür, daß sich der ADT "wie gewünscht" verhält, also sicher ist.

Da in dieser Arbeit dem Spezifikationskonzept ein formales Modell zugrunde liegt und da sich die Sprachkonstrukte zur Spezifikation unmittelbar aus den entsprechenden formalen Begriffen ableiten, können hier die Überlegungen zum Nachweis der Sicherheit, die bereits für das formale Modell angestellt wurden (vgl. Abschnitt 5.4.4), voll übernommen werden. Die nachzuweisenden Verifikationsbedingungen müssen hier lediglich in die Notation der spezifikationssprachlichen Konstrukte umgesetzt werden.

Man betrachte zuerst die Erfüllung der policies für eine Aktivität NACC, NREF und NMOD. Im formalen Modell ist die Erfüllung dieser policies definiert als

- NACC wird erfüllt, wenn gilt

$$\forall (t, \mathcal{H}, \omega) \quad (t, \mathcal{H}, \omega) \in \text{NACC} \longrightarrow$$
$$\exists f \; (f \in \mathcal{F} \wedge (t, \mathcal{H}, \omega, f) \in \text{REJ}) \vee \neg \langle t(\mathcal{H}, \omega) \rangle$$

- NREF/NMOD wird erfüllt, wenn gilt

$$\forall (t, \mathcal{H}, \omega, i) \; [(t, \mathcal{H}, \omega, i) \in \text{NREF/NMOD} \longrightarrow$$
$$\neg \text{abf/änd} \; (t, \mathcal{H}, \omega, di)]$$

Für die Darstellung der Verifikationsbedingungen auf spezifikationssprachlicher Ebene muß von der Darstellung der policies auf dieser Ebene ausgegangen werden. Da diese operationsspezifisch angegeben wurden, ist es sinnvoll, auch die Verifikationsbedingung operationsspezifisch zu definieren.

Bezeichne ANACC das die policy NACC charakterisierende abstrakte Prädikat, AREJj das j-te Prädikat der Zurückweisungsrelation der Operation und ANBL das NBL charakterisierende Prädikat. Weiterhin seien ANREFj und ANMODj die einzelnen Prädikate der Nichtabfrage- bzw. Nichtveränderungspolicy und Tj bzw. Tj` die zugeordneten Teilmengen abstrakter Komponenten.

Für jede Operation muß dann gelten

(i) $\forall$ (ak1,..,akp,x1,..,xm)

$$[\text{ANACC}(ak1,..,akp,x1,..,xm) \longrightarrow$$
$$\bigvee_j \text{AREJj} \; (ak1,..,akp,x1,..,xm) \vee$$
$$\neg \text{ANBL} \; (ak1,..,akp,x1,..,xm)] \quad \wedge$$

$$[\forall j \; (\text{ANREFj} \; (ak1,..,akp,x1,..,xm) \longrightarrow$$
$$\neg \text{abf} \; (op,x1,..,xm,ak1,..,akp,Tj)] \quad \wedge$$

$$[\forall j \; (\text{ANMODj} \; (ak1,..,akp,x1,..,xm) \longrightarrow$$
$$\neg \text{änd} \; (op,x1,..,xm,ak1,..,akp,Tj`)]$$

Für die Informationsflußpolicy NFLOW wurde im formalen Modell die Eigenschaft

$$\forall (t, \pi, t`, \pi`, \Phi)\ [(t, \pi, t`, \pi`, \Phi) \in \text{NFLOW} \dashrightarrow \neg\ \text{flow}\ (t, \pi, t`, \pi`, \Phi)]$$

gefordert. Zur Darstellung dieser Bedingung auf spezifikationssprachlicher Ebene bezeichne Aj1, Aj2 und Φ j die die policy NFLOW charakterisierenden Prädikate und (op1,x11,..,x1m1) und (op2,x21,..,x2m2) Aktivitäten.

Die policy NFLOW ist dann erfüllt, wenn gilt

$$\text{(ii)}\quad \forall (op1,x11,..,x1m1)\ \forall (op2,x21,..,x2m2)\ \forall j\ [Aj1\ (op1,x11,..,x1m1) \wedge Aj2\ (op2,x21,..,x2m2) \wedge \Phi j \dashrightarrow \neg\ \text{flow}\ (op1,x11,..,x1m1,op2,x21,..,x2m2,\Phi j)]$$

Neben der Erfüllung der policies ist die Konsistenz der angegebenen Verträglichkeitsrelationen eine Voraussetzung für die Sicherheit der Spezifikation. Im formalen Modell stellt sich die Konsistenz einer VTGL-Relation dar als

$$\forall (t, \pi, x, \omega)\ [(t, \pi, x, \omega) \in \text{VTGL} \dashrightarrow \text{vtgl}\ (\{t, \pi\} \cup x, \omega)]$$

Für die spezifikationssprachliche Darstellung sei ACOMP das die Verträglichkeitsrelation COMP charakterisierende Prädikat, x eine Aktivität und Y eine Menge von Aktivitäten.

Die spezifizierte Verträglichkeitsrelation ist konsistent, wenn gilt

(iii) $\forall x \forall Y [ACOMP(x,Y,ak1,..,akp) \rightarrow vtgl (\{x\} \cup Y, ak1,..,akp)]$

Mit den gegebenen Bedingungen (i), (ii) und (iii) sind nun die notwendigen Voraussetzungen für die Sicherheit der Spezifikation gegeben. Natürlich ist die Erfüllung dieser Bedingungen im allgemeinen nicht entscheidbar, da sie auf der Abfrage- und Veränderungsdefinition basieren. Für reale Anwendungsfälle sind diese Bedingungen jedoch meistens ohne Probleme verifizierbar.

An dieser Stelle sei noch ein bei der Spezifikation von ADTs häufig eingesetztes Hilfsmittel erwähnt. Oft lassen sich für einen ADT sogenannte Typinvarianten angeben, die bestimmte invariante Eigenschaften der Objekte des Typs beschreiben. Diese Invarianten sind insbesondere für den Übergang von einer informellen zu einer formalen Spezifikation von Interesse. Werden für die Spezifikation solche Typinvarianten angegeben, so ist nachzuweisen, daß durch die Ausführung der Operationen nur solche Objektzustände entstehen, die die entsprechende Invariante erfüllen. Natürlich muß auch nach der Initialisierung die Invariante erfüllt sein (vgl. <Wul76>, <Ker82>).

Für eine formale Darstellung der für die Korrektheit der Spezifikation zusätzlich nachzuweisenden Bedingungen bezeichne AINIT die Initialisierungsbedingungen und AI die abstrakte Invariante der Spezifikation. AEFFECTS sei das Prädikat im EFFECTS-Teil der Operation und ak1`,..,akp` seien die Werte der Objektkomponenten nach der Ausführung der Operation.

Für die Initialisierungsbedingung muß dann gelten

(iv) $\forall$ (ak1,..,akp) [AINIT (ak1,..,akp) ---> AI (ak1,..,akp)]

Für jede Operation op $\in$ OPERATIONS muß gelten

(v) $\forall$ (ak1,..,akp,x1,..,xm)

[AI (ak1,..,akp) $\wedge$ $\neg \bigvee_j$ AREJj (ak1,..,akp,x1,..,xm) $\wedge$

ANBL (ak1,..,akp,x1,..,xm) $\wedge$

AEFFECTS (ak1`,..,akp`,ak1,..,akp,x1,..,xm,y1,..,yn)

---> AI (ak1`,..,akp`)]

In der Praxis wird die Einbeziehung von Typinvarianten das Vorgehen bei der Erstellung der formalen Spezifikation sicherlich erleichtern und übersichtlicher gestalten. Man beachte, daß Bedingung (v) natürlich nur für die Operationen verifiziert werden muß, die irgendwelche Objektkomponenten verändern.

Im folgenden Abschnitt sollen nun der Übergang von der Spezifikation zur Implementierung und die dabei zu beachtenden Bedingungen präsentiert werden.

6.3 Bemerkungen zur korrekten Implementierung

Bereits im Kapitel 3 wurde der Zusammenhang zwischen Spezifikation und Implementierung bei Abstrakten Datentypen erläutert. Wesentlich für die Verifikation der Korrektheit der Implementierung ist dabei die Repräsentationsfunktion REP, die eine Zuordnung zwischen konkreten und abstrakten

Objekten schafft. Sie bildet den Wertebereich der konkreten Repräsentation auf eine Abstrakte Repräsentation surjektiv ab, und ist im allgemeinen eine partielle Funktion, d.h. nicht für jeden Zustand der konkreten Repräsentation muß es ein entsprechendes abstraktes Pendant geben.

Anhand des Diagramms in Abbildung 6-9 läßt sich der Zusammenhang zwischen konkreter und Abstrakter Repräsentation und die zu fordernden Beziehungen, die Voraussetzung für die Korrektheit der Implementierung sind, gut darstellen (vgl <Rei83>):

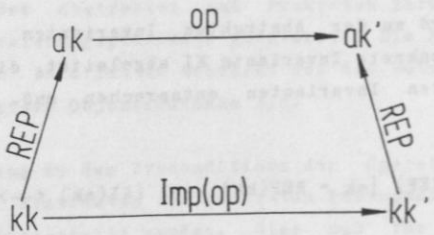


Abb. 6-9: Zusammenhang zwischen konkreter und abstrakter Repräsentation

ak sei hier ein Zustand des abstrakten Objekts, der durch die Operation op in den Zustand ak` übergehe. Der konkrete Zustand der Implementierung kk gehe durch die Implementierung Imp(op) der abstrakten Operation op in den konkreten Zustand kk` über. Die anzugebenden Verifikationsbedingungen fordern nun die Kommutativität des Diagramms, d.h. es muß gelten

op (REP(kk)) = REP (Imp(op)(kk)).

Basis für diese Verifikationsbedingungen sind die in <Wul76>, <Ker82> oder <Rei83> angegebenen Bedingungen, die auf den hier eingeführten Sicherheitsbegriff erweitert werden müssen. So kann dann durch die Verifikation der Sicherheit der Spezifikation und eine Verifikation der Korrektheit der Implementierung bzgl. der Spezifikation letztendlich die Sicherheit der Implementierung nachgewiesen werden.

In dieser Arbeit soll auf die Umsetzung der Spezifikation in die Implementierung und die Verifikation deren Korrektheit nur kurz eingegangen werden, da die zu verwendenden Techniken bereits ausführlich bei Keramidis beschrieben wurden. So werden hier im folgenden nur die formalen Verifikationsbedingungen für die Korrektheit angegeben.

Zunächst wird zu der Abstrakten Invarianten AI eine entsprechende konkrete Invariante KI abgeleitet, die natürlich der Abstrakten Invarianten entsprechen muß. Formal heißt dies

(vi) $\forall ak \in RAN(REP)\ [ak = REP(kk) \dashrightarrow (AI(ak) \leftrightarrow KI(kk))]$

Für die Initialisierung sei die entsprechende konkrete Initialisierungsbedingung mit KINIT bezeichnet. Es muß dann gelten:

(vii) $\forall ak \in RAN(REP)\ [ak = REP(kk) \dashrightarrow (KINIT(kk) \rightarrow AINIT(ak) \wedge KI(kk))]$

d.h. wenn die konkrete Initialisierungsbedingung KINIT für einen Wert kk der konkreten Repräsentation erfüllt ist, muß auch die entsprechende abstrakte Initialisierungsbedingung für den entsprechenden Wert ak erfüllt sein. Die zusätzliche Forderung nach der Erfüllung der konkreten Invarianten KI ist eine Forderung auf Implementierungsebene und entspricht der Bedingung (iv) bei der Spezifikation.

Zur Formulierung der weiteren Bedingungen bezeichne jeweils A... die abstrakten Prädikate bzw. policies und K... die entsprechenden konkreten Prädikate. Man betrachte zunächst den Zusammenhang zwischen den abstrakten und konkreten Preconditions. Hier muß für jede Operation op OPERATIONS gelten

(viii) $\forall j \forall ak \in RAN(REP)\ [ak = REP(kk) \rightarrow (KI(kk) \wedge AREJj\ (ak,x1,..,xm) \leftrightarrow KREJj\ (kk,x1,..,xm))] \wedge$

$\forall ak \in RAN(REP)\ [ak = REP(kk) \rightarrow (KI(kk) \wedge ANBL\ (ak,x1,..,xm) \leftrightarrow KNBL\ (kk,x1,..,xm))]$

Durch diese Bedingung wird im wesentlichen die Äquivalenz der abstrakten und konkreten Zurückweisungs- bzw. Nichtblockierungsprädikate gefordert. Die Einbeziehung der konkreten Invarianten schränkt nur die Betrachtung auf gültige konkrete Objektzustände ein.

Analog zu den Preconditions der Operationen muß auch zwischen abstrakten und konkreten Postconditions ein Zusammenhang hergestellt werden. Hier muß für jede Operation gelten

(ix) $\forall ak,ak` \in RAN(REP)\ [ak = REP(kk) \wedge ak` = REP(kk`) \rightarrow$

$(KEFFECTS\ (kk`,kk,x1,..,xm,y1,..,yn) \wedge KI(kk`) \rightarrow$

$AEFFECTS\ (ak`,ak,x1,..,xm,y1,..,yn))]$

Auch hier beschränkt die konkrete Invariante KI(kk`) wieder die Betrachtung auf gültige Zustände.

Auch bei den Synchronisations- und Schutzprädikaten wird eine ähnliche Entsprechung von konkreten und abstrakten Prädikaten bzw. policies gefordert. Man muß hier - um eine übersichtliche Darstellung zu erhalten - zwischen operationsspezifischen policies und solchen, die mehrere Akti-

vitäten betreffen, differenzieren. In der formalen Darstellung der entsprechenden Bedingungen seien mit x und y Aktivitäten bzw. mit Y eine Aktivitätenmenge bezeichnet.

Die erste (operationsspezifische) Bedingung fordert für jede Operation

$$
\begin{aligned}
(x)\ \forall\, ak \in RAN(REP)\ [ak = REP(kk)\ \text{--->}\\
(ANACC\ (x,ak)\ \text{<-->}\ KNACC\ (x,kk)) \wedge\\
\forall j\ (ANREFj\ (x,ak,Tj(ak))\ \text{<-->}\ KNREFj\ (x,kk,\widetilde{Tj}(kk))) \wedge\\
\forall j\ (ANMODj\ (x,ak,Tj\grave{}(ak))\ \text{<-->}\ KNMODj\ (x,kk,\widetilde{Tj\grave{}}(kk)))]
\end{aligned}
$$

Hierbei bedeuten $\widetilde{Tj}(kk)$ und $\widetilde{Tj\grave{}}(kk)$ die den Mengen $Tj(ak)$ und $Tj\grave{}(ak)$ von abstrakten Objektkomponenten entprechenden Mengen konkreter Objektkomponenten.

Weiterhin muß für den gesamten ADT gelten

$$
\begin{aligned}
(xi)\ \forall\, ak \in RAN(REP)\ [ak = REP(kk)\ \text{--->}\\
(APRIO\ (x,y,ak)\ \text{<-->}\ KPRIO\ (x,y,kk)) \wedge\\
(ACOMP\ (x,Y,ak)\ \text{<-->}\ KCOMP\ (x,Y,kk)) \wedge\\
(ANFLOW\ (x,y,\Phi_a(ak,\ldots))\ \text{<-->}\ KNFLOW\ (x,y,\Phi_k(kk,\ldots)))]
\end{aligned}
$$

Hier bezeichne $\Phi_k(kk,\ldots)$ das aus $\Phi_a(ak,\ldots)$ durch Ersetzen der Abstrakten durch die konkrete Repräsentation entstehende Prädikat.

Mit diesen Bedingungen (vi) bis (xi) sind die Voraussetzungen für eine korrekte Implementierung der Spezifikation gegeben. Im wesentlichen handelt es sich um eine Entsprechung der Prädikate und policies auf Spezifikations- und Implementierungsebene. Als wichtigste Basis für die Beziehungen zwischen Spezifikations- und Implementierungsebene dient hierbei die Repräsentationsfunktion REP, die konkreten Objekten der Implementierungsebene abstrakte Objekte der Spezifikationsebene zuordnet.

Zusätzlich zu den o.g. Bedingungen, die ja die Schnittstelle zwischen Spezifikation und Implementierung betreffen, ist von der Implementierung selbst noch die Bedingung zu erfüllen, daß durch die Operation - ausgehend von konkreten Objekten, die die konkrete Invariante erfüllen - nur wieder solche Objekte entstehen, und daß bei Erfüllung der konkreten Preconditions nach Ausführung der Operation die konkreten Postconditions gelten. Formal ausgedrückt muß für jede konkrete Operation Imp(op) gelten

$$(xii)\quad \forall\, kk \in DOM(REP)\,[\,KI(kk) \wedge \neg \bigvee_{j} KREJj\,(kk,x1,..,xm) \wedge$$

$$KNBL\,(kk,x1,..,xm)\ \{\,Imp(op)(kk)\,\}$$

$$KEFFECTS\,(kk`,kk,x1,..,xm,y1,..,yn) \wedge KI(kk`)\,]$$

7. Beispielhafte Spezifikation eines Systems aus dem Gebiet der Fertigungsautomatisierung

Die in dieser Arbeit entwickelte Methodik zur Spezifikation und Implementierung asynchroner, geschützter SW-Systeme soll nun im vorliegenden Kapitel an einem Beispiel aus der Praxis angewandt werden. Es handelt sich hier um die Verwaltung und automatische Verteilung von Programmen für programmierbare Speicherbauelemente (PROMs,PALs) - eine Anwendung, die von ihrer Struktur her eine gewisse Ähnlichkeit mit DNC-Systemen besitzt.

Nach einer Darstellung des Istzustandes wird zunächst die Aufgabenstellung erläutert und die Struktur der zugrundeliegenden HW-Konfiguration sowie die SW-Grobstruktur aufgezeigt. Für die Verwaltung der Zugriffe auf die Dateien des Systems kann dann ein Abstrakter Datentyp definiert werden, der schließlich mit der hier entwickelten Methodik spezifiziert wird.

7.1 Istzustand und Aufgabenstellung

Im Gerätewerk Erlangen der Firma SIEMENS AG werden speicherprogrammierbare numerische Steuerungen (CNCs) vom Typ SINUMERIK hergestellt. Diese Steuerungen sind gekennzeichnet durch eine einheitliche Hardwarebasis mit einem Mikroprozessor als zentralem Bauelement und die Möglichkeit der Anpassung an verschiedene Fertigungsaufgaben (Drehen, Fräsen, Schleifen, ...) durch den Einsatz unterschiedlicher Steuerprogramme (CNC-Betriebssystem). Die Steuerprogramme werden dabei in programmierbaren Speicherbauelementen (PROMs, PALs) in der Steuerung hinterlegt. Zum Herstellungsprozeß der numerischen Steuerung gehört daher auch das Laden der Steuerprogramme in die Speicherbauelemente.

Bisher werden diese Speicherbauelemente durch Kopieren der Programme von sogenannten Masterproms programmiert. Die Masterproms selbst werden zum Schutz vor Beschädigungen in Stahlschränken in der Werkstatt aufbewahrt, wobei die zu einem Steuerprogramm gehörenden Masterproms gemeinsam abgelegt werden. Bei Bedarf wird dann dieser "Masterpromstapel" zur Programmierstation getragen. Jedes Masterprom ist dabei durch ein Schild mit der Erzeugnisnummer zu identifizieren.

Es ist unmittelbar einsichtig, daß diese Vorgehensweise sehr umständlich ist, zumal die große Anzahl an unterschiedlichen Masterpromstapeln eine laufende manuelle Buchführung über Bestand, Einlagerplatz, Zugang und Entnahme aus den Stahlschränken und auch eine große Anzahl an Stahlschränken erfordert.

Dieses manuelle Programmierverfahren durch Kopieren von Masterpromstapeln soll nun durch einen automatischen Herstellungsprozeß ersetzt werden. Die auf die Speicherbauelemente zu kopierenden Steuerprogramme werden hierbei nicht mehr auf Masterproms, sondern in einer Programmbibliothek auf einem Prozeßrechner archiviert, und das Laden der Programme in die Programmiergeräte in der Werkstatt erfolgt automatisch durch eine online Datenübertragung. Die zugrundeliegende HW-Konfiguration zeigt die folgende Abbildung 7-1.

Die vorhandenen Programmiergeräte sind hier online mit dem übergeordneten Archivierungs- und Versorgungsrechner verbunden, wobei jedem Programmiergerät ein Bildschirm und ein Drucker fest zugeordnet ist. Über eine Bildschirmmaske wird nun das zu ladende Steuerprogramm vom Bediener angefordert und unter der Voraussetzung der Erfüllung gewisser Bedingungen in die mit den Speicherbauelementen bestückten Programmiergeräte geladen. Der zugeordnete Drucker dient zur Ausgabe eines "Ladeprotokolls". Neben diesen dezentralen Peripheriegeräten sind an den Rechner noch eine Magnetplatteneinheit zur Archivierung der Daten und zentrale Bildschirmterminals zur Pflege des Programmarchivs ange-

schlossen. Vom Fertigungsrechner aus besteht zusätzlich noch eine Kopplungsmöglichkeit, um Steuerprogramme auch von außen her übernehmen zu können.

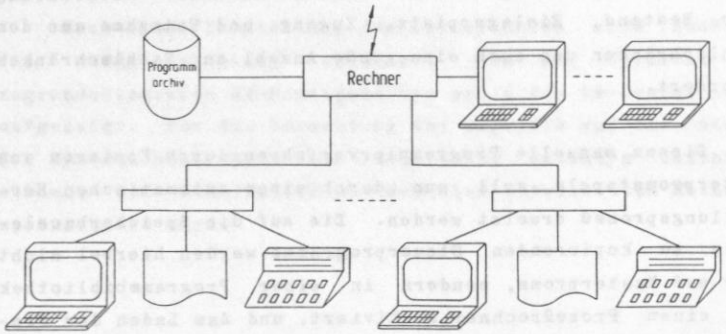


Abb. 7-1: HW-Konfiguration des Anwendungsbeispiels

Abbildung 7-2 zeigt die Grobstruktur der Verwaltungs- und Versorgungs-Software. Grundlage des Systems ist das Programmarchiv mit der Apparatelistendatei (AL-Datei), der Sachnummerndatei (SA-Datei) und dem Fertigungsarchiv (FA-Datei). Die eigentlichen zu ladenden und zu verwaltenden Steuerprogramme sind im Fertigungsarchiv enthalten, während die Sachnummerndatei Parameter für die verschiedenen einsetzbaren Typen von Speicherbauelementen enthält. Die AL-Datei definiert dann die verschiedenen Masterpromstapel

dadurch, daß durch einen Verweis auf die FA-Datei die entsprechende Software angegeben wird und durch einen Verweis auf die SA-Datei die einzusetzenden Speicherbauelementtypen. Eine genaue Beschreibung der Datenstruktur - insbesondere der Verknüpfungen zwischen den Dateien - erfolgt im Abschnitt 7.2.

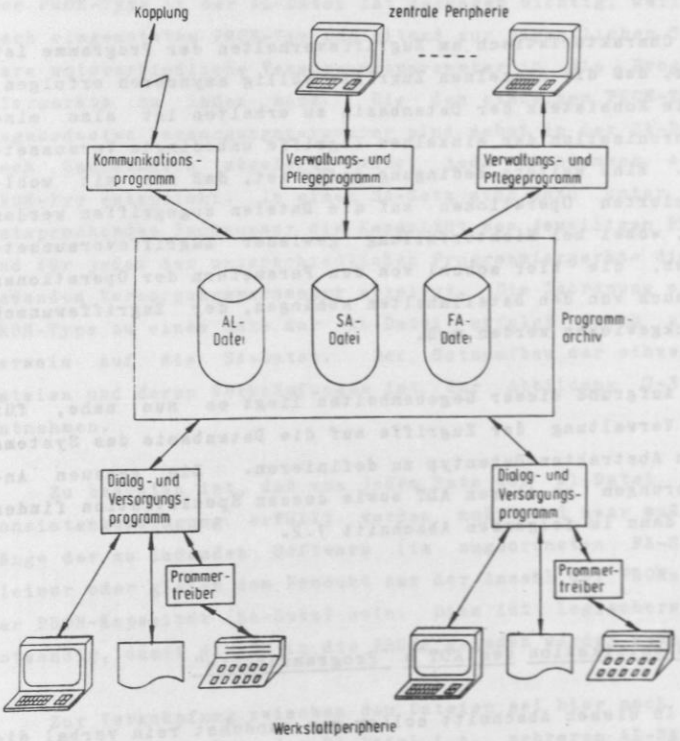


Abb. 7-2: Grobstruktur der Verwaltungs- und Versorgungs-Software

Die verschiedenen Programme des Systems greifen nun auf diese gemeinsame Datenbasis zu. Die Dialog- und Versorgungsprogramme für die einzelnen Bildschirme sowie die zugeordneten Drucker und Programmiergeräte wickeln dabei den Dialog mit dem Benutzer ab und versorgen anschließend das Programmiergerät mit der angeforderten Software aus der FA-Datei und den Parametern aus der SA-Datei. Ihre Zugriffe auf die Dateien sind also ausschließlich lesend. Im Gegensatz dazu verändern das Kommunikationsprogramm und die Verwaltungs- und Pflegeprogramme auch Dateiinhalte.

Charakteristisch am Zugriffsverhalten der Programme ist dabei, daß die einzelnen Zugriffe völlig asynchron erfolgen. Um die Konsistenz der Datenbasis zu erhalten ist also eine Synchronisation der einzelnen Zugriffe unbedingte Voraussetzung. Eine weitere Bedingung dafür ist, daß nur mit wohldefinierten Operationen auf die Dateien zugegriffen werden darf, wobei bei Nichterfüllung gewisser Zugriffsvoraussetzungen, die hier sowohl von den Parametern der Operationen als auch von den Dateiinhalten abhängen, der Zugriffswunsch zurückgewiesen werden muß.

Aufgrund dieser Gegebenheiten liegt es nun nahe, für die Verwaltung der Zugriffe auf die Datenbasis des Systems einen Abstrakten Datentyp zu definieren. Die genauen Anforderungen an diesen ADT sowie dessen Spezifikation finden sich dann im folgenden Abschnitt 7.2.

7.2 Spezifikation des ADT`s "Programmarchiv"

In diesem Abschnitt sollen nun zunächst rein verbal die Anforderungen an den ADT "Programmarchiv" definiert und darauf aufbauend schließlich dessen Spezifikation präsentiert werden.

Wie bereits erwähnt, ist ein "Masterpromstapel" im Programmarchiv durch einen Satz in der AL-Datei definiert. Hier wird festgelegt, welche Software beim Vorliegen der entsprechenden AL-Nummer geladen wird, und um welche PROM-Typen es sich handelt. Schließlich enthält ein Satz der AL-Datei noch die Anzahl der PROMs im Masterstapel sowie deren Erzeugnisnummern. Die Zuordnung der eigentlichen zu ladenden Software erfolgt dabei durch einen Verweis auf die FA-Datei, die sämtliche verfügbare SW-Versionen mit ihrer Länge unter dem Schlüssel FA-Nummer enthält. Die Festlegung des PROM-Typs in der AL-Datei ist deswegen wichtig, weil je nach eingesetztem PROM-Typ begleitend zur eigentlichen Software unterschiedliche Versorgungsparameter in die Programmiergeräte zu laden sind. Die den einzelnen PROM-Typen zugeordneten Versorgungsparameter sind dabei in der SA-Datei nach Sachnummern abgelegt, wobei jede Sachnummer einem PROM-Typ entspricht. In einem SA-Satz sind dann unter der entsprechenden Sachnummer die Kapazität des jeweiligen PROMs und für jedes der unterschiedlichen Programmiergeräte die zu ladenden Versorgungsparameter abgelegt. Die Zuordnung eines PROM-Typs zu einem Satz der AL-Datei erfolgt durch einen Verweis auf die SA-Datei. Der Satzaufbau der einzelnen Dateien und deren Verknüpfungen ist der Abbildung 7-3 zu entnehmen.

Zu beachten ist, daß von jedem Satz der Al-Datei eine Konsistenzbedingung erfüllt werden muß, und zwar muß die Länge der zu ladenden Software (im zugeordneten FA-Satz) kleiner oder gleich dem Produkt aus der Anzahl der PROMs mit der PROM-Kapazität (SA-Satz) sein. Dies ist logischerweise notwendig, damit die SW in die PROMs geladen werden kann.

Zur Verknüpfung zwischen den Dateien sei hier noch bemerkt, daß jeder Satz der SA-Datei i.a. mehreren AL-Sätzen zugeordnet ist, da ja die einzelnen verwendbaren PROM-Typen bei mehreren Steuerungstypen eingesetzt werden. Auch für die Verknüpfung zwischen FA-Datei und AL-Datei ergibt sich, daß ein FA-Satz mehreren AL-Sätzen zugeordnet werden kann. Dies bedeutet, daß die gleiche Software in unterschiedlichen

Masterpromstapeln eingesetzt wird. Dieser Fall tritt ein, wenn für einen Steuerungstyp und somit für eine SW-Version Speicherbausteine unterschiedlicher Kapazität alternativ eingesetzt werden können. Für diese unterschiedlichen Bestückungsmöglichkeiten müssen - da sich ja wegen der veränderten PROM-Kapazität auch die Anzahl der PROMs und deren Erzeugnisnummern ändern - auch verschiedene AL-Nummern definiert werden, die sich jedoch auf die gleiche Software in der FA-Datei beziehen.

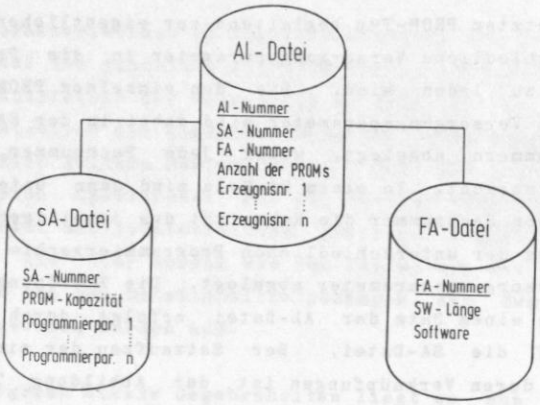


Abb. 7-3: Dateistruktur des ADT`s "Programmarchiv"

Auf dieser Datenbasis müssen nun eine Reihe von Operationen definiert werden, die teils von den Verwaltungs- und Pflegeprogrammen, teils von den Dialog- und Versorgungsprogrammen benötigt werden. Auch diese Operationen seien hier zunächst verbal beschrieben.

Für jede der Dateien müssen zunächst Operationen zum Einfügen eines neuen und zum Löschen eines nicht mehr benötigten Satzes existieren. Aufgrund der Verknüpfungen zwischen den Dateien müssen bei der Ausführung der Operationen einige Restriktionen beachtet werden, die vom ADT durchgesetzt werden müssen. So kann ein neuer AL-Satz nur dann

eingefügt werden, wenn für die anzugebende FA- und SA-Nummer entsprechende Sätze in der FA- und SA-Datei existieren. Weiterhin muß bei dieser Operation die o.g. Konsistenzbedingung erfüllt sein. Analog dazu kann ein SA- bzw. FA-Satz nur dann gelöscht werden, wenn die entsprechende SA-/FA-Nummer in keinem existierenden AL-Satz mehr enthalten ist.

Neben diesen Einfüge- und Löschoperationen sind zur Pflege des Programmarchivs noch zwei Änderungsoperationen erforderlich:

- Beim Hinzukommen eines neuen Programmiergeräts oder bei Änderung der Versorgungsparameter eines bereits vorhandenen Geräts müssen in der SA-Datei die entsprechenden Programmierparameter eingefügt bzw. geändert werden. Die Einfügung bzw. Änderung erfolgt dabei satzweise, da sich diese Veränderungen nicht notwendigerweise auf jeden PROM-Typ beziehen.

- Wegen Lieferschwierigkeiten einzelner PROM-Typen kommt es relativ häufig vor, daß für eine AL-Nummer Ersatztypen gleicher Kapazität eingesetzt werden müssen. Dies erfordert eine Änderung der Zuordnung der SA-Nummer im entsprechenden AL-Satz. Zu beachten ist hierbei, daß wegen der Konsistenzbedingung nur PROMs mit gleicher Kapazität eingesetzt werden können.

Ein Umsteigen auf PROMs mit unterschiedlicher Kapazität ist mit diesen Änderungsoperationen nicht zu realisieren. Hierfür ist die Definition eines neuen AL-Satzes erforderlich, da sich ja mit der Kapazität auch die PROM-Anzahl und Erzeugnisnummern ändern. Dies entspricht allerdings genau der bisherigen Vorgehensweise beim Kopieren der Software von Masterpromstapeln.

Analog dazu wird bei Änderung der SW-Version immer ein neuer Satz in der FA-Datei angelegt, und nicht der alten FA-Nummer eine neue Software zugeordnet. Auch in der AL-Datei wird hier nicht die Zuordnung zur FA-Datei geändert, sondern ein neuer AL-Satz geschaffen, so daß hier keine weitere Änderungsoperation erforderlich wird. Somit ist also gewährleistet, daß pro SW-Ausgabestand und PROM-Kapazität jeweils ein AL-Satz existiert.

Zuletzt seien hier die reinen Leseoperationen, die von den Dialog- und Versorgungsprogrammen benutzt werden, beschrieben. Es handelt sich hier um zwei verschiedene Operationen und zwar zum einen um das Lesen der gesamten zu einer AL-Nummer gehörenden Software und deren Zuordnung zu den einzelnen PROMs, die durch die Erzeugnisnummern charakterisiert werden, und zum anderen um das gezielte Auswählen der zu einem bestimmten PROM gehörenden Teilsoftware. Zusätzlich zur eigentlichen Software werden hier noch die Programmierparameter aus der SA-Datei benötigt, wobei natürlich für das angegebene Programmiergerät ein Eintrag im entsprechenden SA-Satz existieren muß.

Hauptaufgabe des zu spezifizierenden ADT`s ist nun, die erwähnten Bedingungen zur Konsistenzerhaltung dadurch durchzusetzen, daß Zugriffswünsche, die diese verletzen würden, zurückgewiesen werden. Weiterhin müssen die einzelnen anfallenden Aktivitäten synchronisiert werden, um Inkonsistenzen aufgrund eines gleichzeitigen Zugriffs auf gemeinsame Daten zu verhindern. Prinzipiell könnten in den ADT auch noch weitere Zugriffsschutzmechanismen, wie z.B. die Vergabe von Rechten zur Ausführung bestimmter Operationen an die einzelnen Benutzerprogramme, integriert werden. Aus Gründen der Übersichtlichkeit soll hier allerdings davon abgesehen werden.

Die Spezifikation des ADT`s "Programmarchiv" wird nun schrittweise entwickelt, d.h. ausgehend von der Spezifikation der Abstrakten Repräsentation werden zunächst die einzelnen Operationen mit ihren policies und Ausführungsbe-

schränkungen spezifiziert. Anschließend werden dann die Synchronisationsbedingungen angegeben.

Die Abstrakte Repräsentation des ADT`s "Programmarchiv" (vgl. Abbildung 7-4) wird hier völlig analog zur Struktur aus Abbildung 7-3 spezifiziert, d.h. sie besteht aus den drei Dateien ALDAT, SADAT und FADAT. Spezifikationssprachlich wird hierfür das FILE-Konstrukt verwendet, das dem normalen ARRAY-Konstrukt für Felder entspricht. Zusätzlich zur Identifikation der einzelnen Sätze über den Index (z.B. aldat[i]) wird hier allerdings noch die Identifikationsmöglichkeit über den Primärschlüssel eingeführt; der Primärschlüssel wird dabei in der Spezifikation durch den Begriff KEY deklariert. Die Identifikation eines Datensatzes über den Schlüssel schreibt sich dann durch eine doppelte eckige Klammer, also z.B. "aldat[[anr]]". Wie sich bei der Spezifikation der Operationen herausstellen wird, erleichtert dieses Konstrukt die Übersichtlichkeit der Spezifikation wesentlich.

Zusätzlich zur Skizze aus Abbildung 7-3 werden hier noch die Programmierparameter in der SA-Datei näher detailliert. "propak" steht hier für die Bezeichnung des Programmiergeräts, während "family" und "pinout" die eigentlichen Parameter bezeichnen. Der in Abbildung 7-4 bereits spezifizierte SYN-Teil wird nach der Darstellung der Operationen erläutert.

```
MODULE PROGRAMMARCHIV (maxal,maxsa,maxfa:INTEGER≥1)

SPECIFICATION

DECLARATIONS

   ABSTRACT REPRESENTATION
   aldat: FILE [1..maxal] OF alrec INITIALLY empty;
   sadat: FILE [1..maxsa] OF sarec INITIALLY empty;
   fadat: FILE [1..maxfa] OF farec INITIALLY empty;

   TYPE alrec:        RECORD
                      alnr       : ALPHANUMERIC KEY;
                      sa         : ALPHANUMERIC;
                      fa         : ALPHANUMERIC;
                      anzprom    : INTEGER;
                      erznr      : SET OF INTEGER;
                      END;

   TYPE sarec:        RECORD
                      sanr       : ALPHANUMERIC KEY;
                      promcap    : INTEGER;
                      progpar    : ARRAY[1..n] OF prommerpar;
                      END;

   TYPE prommerpar:   RECORD
                      propak     : STRING;
                      family     : INTEGER;
                      pinout     : INTEGER;
                      END;

   TYPE farec:        RECORD
                      fanr       : ALPHANUMERIC KEY;
                      length     : INTEGER;
                      software   : STRING;
                      END;
```

```
PARAMETERS

f, gerät, sw, swerzn          : STRING
i, n, erzn                    : INTEGER
posit                         : (1..n)
erzeugnisnr                   : SET OF INTEGER

newsa                         : sarec
newfa                         : farec
newal                         : alrec
san, fan, aln                 : ALPHANUMERIC KEY
newprompar, gerätepar         : prommerpar

SYN

COMP(x,y): x ∈ INSERTSA ∧ y ∈ INSERTFA ∪ INSERTAL  ∨
           x ∈ INSERTFA ∧ y ∈ INSERTAL  ∨

           x ∈ DELETESA ∧ y ∈ DELETEFA ∪ DELETEAL  ∨
           x ∈ DELETEFA ∧ y ∈ DELETEAL  ∨

           x ∈ CHSAPROMPAR(san1) ∧ y ∈ CHSAPROMPAR(san2) ∧
             san1 ≠ san2  ∨
           x ∈ CHSAPROMPAR ∧ y ∈ CHALSANR  ∨
           x ∈ CHSAPROMPAR(san1) ∧ y ∈ READAL(aln1) ∪
             READALERZ(aln1) ∧ aldat[[aln1]].sa ≠ san1  ∨

           x ∈ CHALSANR(aln1) ∧ y ∈ CHALSANR(aln2) ∧
             aln1 ≠ aln2  ∨
           x ∈ CHALSANR(aln1) ∧ y ∈ READAL(aln2) ∪
             READALERZ(aln2) ∧ aln1 ≠ aln2  ∨

           x,y ∈ READAL ∪ READALERZ
```

```
PRIO(x,y): x ∈ INSERTSA ∪ INSERTFA ∪ INSERTAL ∪
              DELETESA ∪ DELETEFA ∪ DELETEAL   ∧
           y ∈ CHSAPROMPAR ∪ CHALSANR ∪ READAL ∪ READALERZ ∨

           x ∈ CHSAPROMPAR ∪ CHALSANR ∧
           y ∈ READAL ∪ READALERZ
```

Abb. 7-4: Deklarations- und Synchronisationsteil des Moduls "Programmarchiv"

Die Operationen und deren policies sind dann in Abbildung 7-5 spezifiziert. Auch hier wurde exakt gemäß der verbalen Beschreibung vorgegangen.

Die Operation insertsa (newsa) --> f fügt einen neuen Satz in die SA-Datei ein. Sie kann nur ausgeführt werden, wenn die SA-Nummer des einzufügenden Satzes (Schlüssel) noch nicht in der SA-Datei existiert und wenn noch ein Platz in der Datei frei ist. Im EFFECTS-Teil der Operation wird dann ein bisher leerer Satz mit den Daten des einzufügenden Satzes belegt und der Fehlerkennung f das leere Wort zugewiesen. Völlig analog hierzu ist die Operation insertfa (newfa) --> f für die FA-Datei definiert.

Bei der Operation insertal (newal) --> f wird ein neuer Satz in die AL-Datei eingefügt. Zusätzlich zur Forderung, daß kein Satz unter dem gleichen Schlüssel bereits existieren darf und daß noch mindestens ein Satz in der Datei frei ist, müssen hier für die in "newal" angegebenen SA- und FA-Nummern Sätze in den entsprechenden Dateien existieren. Eine weitere Voraussetzung für die Ausführbarkeit der Operationen ist die Einhaltung der erwähnten Konsistenzbedingung. Die Spezifikation der entsprechenden REJ-Bedingung sei hier etwas genauer erklärt.

Die dem einzufügenden AL-Satz zugeordneten FA- bzw. SA-Nummern werden mit "newal.fa" bzw. "newal.sa" bezeichnet. Dann bezeichnen "fadat[[newal.fa]]" und "sadat[[newal.sa]]" die durch den jeweiligen Schlüssel identifizierten Sätze der FA- und SA-Datei. "fadat[[newal.fa]].length" ist dann die Länge der im FA-Satz befindlichen Software und "sadat[[newal.sa]].promcap" die Kapazität des verwendeten PROM-Typs. Somit stellt sich die Konsistenzbedingung dar als

```
fadat[[newal.fa]].length ≤
newal.anzprom  *  sadat[[newal.sa]].promcap
```

Von allen drei insert-Operationen muß die policy erfüllt werden, daß bei ihrer Ausführung nur leere, d.h. noch nicht belegte Sätze, in den entsprechenden Dateien verändert werden und keine lesenden Zugriffe stattfinden können.

Das Gegenstück zu den insert-Operationen bilden die delete-Operationen, bei denen nicht mehr benötigte Sätze aus den entsprechenden Dateien gelöscht werden. Die Spezifikation erfolgt völlig analog zu den insert-Operationen, so daß hier auf eine detaillierte Beschreibung der Spezifikation verzichtet werden kann. Auch die Spezifikation der Operation chsaprompar zum Ändern von Programmierparametern in der SA-Datei ergibt sich direkt aus der verbalen Beschreibubg.

Bei der Operation chalsanr, die die Zuordnung eines PROM-Typs (SA-Nummer) in einem AL-Satz ändert, muß - wie bereits erläutert - darauf geachtet werden, daß die Kapazität des alten und neuen PROM-Typs übereinstimmen. Sei "aln" der entsprechende Schlüssel für den AL-Satz, dann ist die zugeordnete alte SA-Nummer durch "aldat[[aln]].sa" gegeben. Dieser Ausdruck muß nun seinerseits als Schlüssel für die SA-Datei verwendet werden, so daß der entsprechende SA-Satz durch "sadat[[aldat[[aln]].sa]]" und die gewünschte Kapazität durch "sadat[[aldat[[aln]].sa]].promcap" identifiziert wird. Die Kapazität des neuen PROM-Typs (Schlüssel "san") ergibt sich einfacher durch "sadat[[san]].promcap". Im

EFFECTS-Teil der Operation wird nun bei Nichterfüllung der REJ-Bedingungen die entsprechende SA-Zuordnung des AL-Satzes geändert, oder spezifikationssprachlich ausgedrückt "aldat[[aln]].sa = san".

Die beiden read-Operationen des ADT`s können genau dann ausgeführt werden, wenn der entsprechende durch aln identifizierte AL-Satz existiert und für das durch "gerät" angegebene Programmiergerät im zugeordneten SA-Satz ein Eintrag ist. Auch hier ist ein Verweis auf die SA-Datei notwendig, wobei die Identifikation des zugeordneten SA-Satzes wieder durch "sadat[[aldat[[aln]].sa]]" geschieht.
Durch "sadat[[aldat[[aln]].sa]].progpar[i].propak" wird dann der Name des i-ten Geräts im Feld progpar bezeichnet. Als Nichtabfragepolicy muß von beiden Operationen erfüllt werden, daß außer dem durch "anr" identifizierten AL-Satz nur noch der jeweils zugeordnete FA- und SA-Satz gelesen werden können. Veränderungen dürfen nicht stattfinden. Im folgenden seien nun in Abbildung 7-5 die Operationen und deren policies spezifiziert.

POLICIES

insertsa (newsa)

NACC: $\exists$ i (1≤i≤maxsa) (sadat[i].sanr = newsa.sanr) $\vee$
$\neg \exists$ i (1≤i≤maxsa) (sadat[i].sanr = λ)

NREF: true NREF aldat $\cup$ sadat $\cup$ fadat

NMOD: true NMOD aldat $\cup$ fadat $\cup$ {sadat[i] | sadat[i].sanr $\neq \lambda$}

insertfa (newfa)

NACC: $\exists$ i (1≤i≤maxfa) (fadat[i].fanr = newfa.fanr) $\vee$
$\neg \exists$ i (1≤i≤maxfa) (fadat[i].fanr = λ)

NREF: true NREF aldat $\cup$ sadat $\cup$ fadat

NMOD: true NMOD aldat $\cup$ sadat $\cup$ {fadat[i] | fadat[i].fanr $\neq \lambda$}

insertal (newal)

NACC: $\exists$ i (1≤i≤maxal) (aldat[i].alnr = newal.alnr) $\vee$
$\neg \exists$ i (1≤i≤maxal) (aldat[i].alnr = λ) $\vee$
$\neg \exists$ i (1≤i≤maxsa) (sadat[i].sanr = newal.sa) $\vee$
$\neg \exists$ i (1≤i≤maxfa) (fadat[i].fanr = newal.fa) $\vee$
fadat[[newal.fa]].length >
newal.anzprom * sadat[[newal.sa]].promcap

NREF: true NREF aldat $\cup$ sadat $\cup$ fadat

NMOD: true NMOD sadat $\cup$ fadat $\cup$ {aldat[i] | aldat[i].alnr $\neq \lambda$}

```
deletesa (san)

NACC:¬∃ i (1≤i≤maxsa) (sadat[i].sanr = san) ∨
     ∃ i (1≤i≤maxal) (aldat[i].sa = san)

NREF: true NREF aldat ∪ sadat ∪ fadat

NMOD: true NMOD aldat ∪ fadat ∪ {sadat[i] | sadat[i].sanr ≠ san}

deletefa (fan)

NACC:¬∃ i (1≤i≤maxfa) (fadat[i].fanr = fan) ∨
     ∃ i (1≤i≤maxal) (aldat[i].fa = fan)

NREF: true NREF aldat ∪ sadat ∪ fadat

NMOD: true NMOD aldat ∪ sadat ∪ {fadat[i] | fadat[i].fanr ≠ fan}

deleteal (aln)

NACC:¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)

NREF: true NREF aldat ∪ sadat ∪ fadat

NMOD: true NMOD sadat ∪ fadat ∪ {aldat[i] | aldat[i].alnr ≠ aln}


chsaprompar (san,posit,newprompar)

NACC:¬∃ i (1≤i≤maxsa) (sadat[i].sanr = san)

NREF: true NREF aldat ∪ sadat ∪ fadat

NMOD: true NMOD aldat ∪ fadat ∪ sadat -
     {sadat[[san]].progpar[posit]}
```

```
chalsanr (aln,san)

NACC:¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)  ∨
     ¬∃ i (1≤i≤maxsa) (sadat[i].sanr = san)  ∨
      sadat[[aldat[[aln]].sa]].promcap ≠
      sadat[[san]].promcap

NREF: true NREF aldat∪sadat∪fadat

NMOD: true NMOD aldat∪fadat∪sadat - {aldat[[aln]].sa}


readal (aln,gerät)

NACC:¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)  ∨
     ¬∃ i (1≤i≤n) (gerät =
          sadat[[aldat[[aln]].sa]].progpar[i].propak)

NREF: true NREF {aldat[i] | aldat[i].alnr ≠ aln} ∪
                {sadat[i] | sadat[i].sanr ≠ aldat[[aln]].sa} ∪
                {fadat[i] | fadat[i].fanr ≠ aldat[[aln]].fa}

NMOD: true NMOD aldat∪sadat∪fadat


readalerz (aln,erzn,gerät)

NACC:¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)  ∨
     ¬∃ i (1≤i≤n) (gerät =
          sadat[[aldat[[aln]].sa]].progpar[i].propak  ∨
      erzn ∉ aldat[[aln]].erznrn

NREF: true NREF      ...

NMOD: true NMOD      ...
```

```
OPERATIONS

insertsa (newsa) ---> f

REJ:      ∃ i (1≤i≤maxsa) (sadat[i].sanr = newsa.sanr)
          ERROR f := "sa-satz existiert bereits"

         ¬∃ i (1≤i≤maxsa) (sadat[i].sanr = λ)
          ERROR f := "kein satz mehr frei"

EFFECTS:  ∃ i (`sadat[i]=λ ∧ sadat[i] = newsa∧ f = λ)


insertfa (newfa) ---> f

REJ:      ∃ i (1≤i≤maxfa) (fadat[i].fanr = newfa.fanr)
          ERROR f := "fa-satz existiert bereits"

         ¬∃ i (1≤i≤maxfa) (fadat[i].fanr = λ)
          ERROR f := "kein satz mehr frei"

EFFECTS:  ∃ i (`fadat[i]=λ ∧ fadat[i] = newfa∧ f = λ)


insertal (newal) ---> f

REJ:      ∃ i (1≤i≤maxal) (aldat[i].alnr = newal.alnr)
          ERROR f := "al-satz existiert bereits"

         ¬∃ i (1≤i≤maxal) (aldat[i].alnr = λ)
          ERROR f := "kein satz mehr frei"

         ¬∃ i (1≤i≤maxsa) (sadat[i].sanr = newal.sa)
          ERROR f := "angegebene sa-nummer existiert nicht"
```

```
        ¬∃ i (1≤i≤maxfa) (fadat[i].fanr = newal.fa)
         ERROR f := "angegebene fa-nummer existiert nicht"

         fadat[[newal.fa]].length >
         newal.anzprom * sadat[[newal.sa]].promcap
         ERROR f := "konsistenzbedingung nicht erfüllt"

EFFECTS: ∃ i (`aldat[i]=λ ∧ aldat[i] = newal∧ f = λ)


deletesa (san) ---> f

REJ:     ¬∃ i (1≤i≤maxsa) (sadat[i].sanr = san)
         ERROR f := "sa-satz existiert nicht"

         ∃ i (1≤i≤maxal) (aldat[i].sa = san)
         ERROR f := "promtyp wird noch in al-datei verwendet"

EFFECTS: ∃ i (`sadat[i].sanr = san
              sadat[i].sanr = λ ∧ sadat[i].promcap = 0 ∧
              ∀ j ε {1,..,n} (sadat[i].progpar[j] = (λ,0,0)) ∧
              f = λ)


deletefa (fan) ---> f

REJ:     ¬∃ i (1≤i≤maxfa) (fadat[i].fanr = fan)
         ERROR f := "fa-satz existiert nicht"

         ∃ i (1≤i≤maxal) (aldat[i].sa = san)
         ERROR f := "software wird noch in al-datei verwendet)

EFFECTS: ∃ i (`fadat[i].fanr = fan ∧ fadat[i].fanr = λ ∧
              fadat[i].length = 0 ∧ fadat[i].software = λ ∧
              f = λ)
```

```
deleteal (aln) ---> f

REJ:     ¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)
          ERROR f := "al-satz existiert nicht"

EFFECTS: ∃ i (`aldat[i].alnr = aln ∧ aldat[i].alnr = λ ∧
              aldat[i].sa = λ ∧ aldat[i].fa = λ ∧
              aldat[i].anzprom = 0 ∧ aldat[i].erznrn = 0 ∧
              f = λ)


chsaprompar (san,posit,newprompar) ---> f

REJ:     ¬∃ i (1≤i≤maxsa) (sadat[i].sanr = san)
          ERROR f := "sa-satz existiert nicht"

EFFECTS: sadat[[san]].progpar[posit] = newprogpar ∧ f = λ


chalsanr (aln,san) ---> f

REJ:     ¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)
          ERROR f := "al-satz existiert nicht"

         ¬∃ i (1≤i≤maxsa) (sadat[i].sanr = san)
          ERROR f := "sa-satz existiert nicht"

          sadat[[aldat[[aln]].sa]].promcap ≠
          sadat[[san]].promcap
          ERROR f := "falsche promkapazität"

EFFECTS: aldat[[aln]].sa = san ∧ f = λ
```

```
readal (aln,gerät) ---> (gerätepar,erzeugnisnr,sw,f)

REJ:    ¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)
        ERROR f := "al-satz existiert nicht"

        ¬∃ i (1≤i≤n) (gerät =
             sadat[[aldat[[aln]].sa]].progpar[i].propak)
        ERROR f := "falsches programmiergerät"

EFFECTS: gerätepar = ι progpar[i] (gerät =
        sadat[[aldat[[aln]].sa]].progpar[i].propak) ∧
        erzeugnisnr = aldat[[aln]].erznr ∧
        sw = fadat[[aldat[[aln]].fa]].software ∧ f = λ


readalerz (aln,erzn,gerät) ---> (gerätepar,swerzn,f)

REJ:    ¬∃ i (1≤i≤maxal) (aldat[i].alnr = aln)
        ERROR f := "al-satz existiert nicht"

        erzn ∉ aldat[[aln]].erznrn
        ERROR f := "erzn im al-satz nicht vorhanden"

        ¬∃i (1≤i≤n) (gerät =
             sadat[[aldat[[aln]].sa]].progpar[i].propak)
        ERROR f := "falsches programmiergerät"

EFFECTS: gerätepar = ι progpar[i] (gerät=
        sadat[[aldat[[aln]].sa]].progpar[i].propak)
        swerznr =
        TEILSOFTWARE (erzn,fadat[[aldat[[aln]].fa]].software)
        ∧ f = λ
```

Abb. 7-5: Policies und Operationen des Moduls "Programm-archiv

Nach dieser Erläuterung der Spezifikation der Operationen und deren policies soll hier noch kurz auf die Synchronisationsbedingungen und globale policies eingegangen werden. Zunächst sei bemerkt, daß im vorliegenden Anwendungsfall keine Informationsflußpolicies notwendig sind, und somit bei der Spezifikation das NFLOW-Konstrukt entfallen kann.

Die Synchronisationsbedingungen wurden bereits in Abbildung 7-4 dargestellt. Hierzu sei bemerkt, daß durch das COMP-Konstrukt festgelegt wird, daß zwar verschiedene insert-Operationen miteinander verträglich sind, da sie sich ja auf unterschiedliche Dateien beziehen, jedoch nicht zwei insertsa, insertfa oder insertal-Operationen miteinander. Analoges gilt für die delete-Operationen. Weiterhin sei die chsaprompar-Operation zur Änderung der Programmierparameter in einem SA-Satz mit der chalsanr-Operation zur Änderung der SA-Zuordnung in einem AL-Satz verträglich, sowie mit solchen Leseoperationen auf AL-Sätze, denen nicht der in der chsaprogpar-Operation zu ändernde SA-Satz zugeordnet ist. Auch zwei chsaprompar-Operationen auf verschiedene SA-Sätze seien miteinander verträglich.

Die chalsanr-Operation zur Änderung der SA-Zuordnung in einem AL-Satz ist dann mit Leseoperationen auf andere AL-Sätze verträglich. Dies gilt natürlich auch für zwei chalsanr-Operationen auf unterschiedliche AL-Sätze. Außerdem seien alle Leseoperationen untereinander verträglich und somit parallel ausführbar.

Zuletzt sei hier noch kurz auf das PRIO-Konstrukt eingegangen, wo festgelegt wird, daß insert und delete-Operationen höherprior als die beiden anderen Verwaltungs- und die Leseoperationen sind. Die übrigen Verwaltungsoperationen chsaprompar und chalsanr sollen schließlich Vorrang vor den reinen Leseoperationen besitzen.

LITERATUR

<Anc83> Ancilotti P., Boari M., Lijtmaer N.
Language Features for Access Control
IEEE Transact. on SW-Engineering Vol. SE-9, No. 1
Jan 1983, pp 16 - 25

<And80> Andrews G.R., Reitman R.P.
An Axiomatic Approach for Information Flow in Programs
ACM Transact. on Progr. Languages, Vol. 2, No. 1
(Jan 1980), pp 56 - 76

<Apt81> Apt K.R.
Ten Years of Hoare`s Logic: A Survey - Part I
ACM Transact. on Progr. Languages, Vol. 3, No. 4
(Okt 1981), pp 431 - 483

<Bal82> Balzert H.
Die Entwicklung von Software Systemen
BI-Verlag Reihe Informatik /34, Zürich 1982

<Bel73> Bell D.E., LaPadula L.J.
Secure Computer Systems: A Mathematical Model
MTR-2547, MITRE Corporation Bedford, Mass.,
Mai 1973

<Bel74> Bell D.E., LaPadula L.J.
Secure Computer Systems: Mathematical Foundations and Model
M74-244, MITRE Corporation Bedford, Mass.,
Sept 1974

<Bel75> Bell D.E., LaPadula L.J.
Secure Computer System: Unified Exposition and Multics Interpretation
MTR-2997, MITRE Corporation Bedford, Mass., Juni 1975

<Ber79> Berson T.A., Barksdale G.L.
KSOS - Development Methodology for A Secure Operating System
Proc. AFIPS Nat. Computer Conf. Vol. 48, AFIPS Press 1979, pp 345 - 353

<Ber80> Berg H.K., Giloi W.K.
The Use of Formal Specification of Software
Informatik Fachberichte Bd. 36, Springer Verlag Berlin 1980

<Bib77> Biba K.J.
Integrity Considerations for Secure Computer Systems
ESD-TR-76-372, MITRE Corporation Bedford, Mass., April 1977

<Bie79> Biewald J. et al.
EPOS - A Specification and Design Technique for Computer-Controlled Real-Time Automation Systems
Proc. 4th Int. Conf. on SW-Engineering, Sept 1979 pp 245 - 250

<Bie79> Biewald J.
Rechnergestützte Erzeugung der Dokumentation für den Funktions- und Softwareentwurf in EPOS
Fachtagung Prozeßrechner 1981, Informatik Fachberichte Nr. 39, Springer Verlag Berlin 1981, pp 97 - 106

<Bis79> Bishop M., Snyder L.
The Transfer of Information and Authority in A Protection System
ACM SIGOPS Operating System Review 13,4 (Dez 1979) pp 45 - 54

<Bis81> Bishop M.
Hierarchical Take-Grant Protection Systems
ACM Operating System Reviews 1981, pp 109 - 122

<Bri72> Brinch Hansen P.
A Comparision of Two Synchronisation Concepts
Acta Informatica 1,3 (1972), pp 190 - 199

<Cai75> Caine S.H., Gordon E.K.
PDL - A Tool for Software Design
Proc. AFIPS Nat. Computer Conf. 1975,
AFIPS Press 1975, pp 271 - 276

<Cam74> Campbell R.H., Habermann A.N.
The Specification of Process Synchronisation by Path Expressions
Lecture Notes in Computer Science vol. 16,
Springer Verlag Berlin 1974, pp 89 - 102

<Cau79> Mc Cauley E.J., Drongowski P.R.
KSOS - The Design of A Secure Operating System
Proc. AFIPS Nat. Computer Conf. Vol. 48
AFIPS Press 1979, pp 345 - 353

<Che81> Cheheyl M.H. et al.
Verifying Security
ACM Computing Surveys Vol. 13, No. 3 (Sept 1981), pp 279 - 339

<Coh75> Cohen E., Jefferson D.
Protection in the HYDRA Operating System
Proc. 5th Symp. on Operating System Principles
Nov 1975, pp 141 - 160

<Coh77> Cohen E.
Information Transmission in Computational Systems
Proc. 6th Symp. on Operating System Principles
Nov 1977, pp 133 - 139

<Coh78> Cohen E.
Information Transmission in Sequential Programs
in Foundations of Secure Computation, Hrsg.
De Millo et al., Academic Press 1978
pp 297 - 335

<DeB80> De Bakker J.W.
Mathematical Theory of Program Correctness
Prentice Hall Englewood Cliffs, New York 1980

<DeM78> De Millo R.A. et al.
Foundations of Secure Computation
Academic Press, New York 1978

<Den66> Dennis J.B., Van Horn E.C.
Programming Semantics for Multiprogrammed
Computation
CACM 9,3 (Maerz 1966), pp 143 - 155

<Den75> Denning D.E.
Secure Information Flow in Computer Systems
Ph.D. Thesis, Purdue University, Computer Science
Dept., Technical Report 145, 1975

<Den76> Denning D.E
A Lattice Model of Secure Information Flow
CACM 19,5 (Mai 1976), pp 236 - 243

<Den76a> Denning D.E., Denning P.J.
Certification of Programs for Secure Information Flow
Purdue Univ., Technical Report CSD-TR181, 1976

<Den77> Denning D.E., Denning P.J.
Certification of Programs for Secure Information Flow
CACM 20,7 (Juli 1977), pp 504 - 513

<Den79> Denning D.E., Denning P.J.
Data Security
ACM Computing Surveys Vol. 11, No. 3, Sept 1979
pp 227 - 249

<Den79a> Denert E.
Software-Modularisierung
Informatik Spektrum 2 (1979), pp 204 - 218

<Den83> Denert E.
Möglichkeiten und Grenzen des Software-Engineering
in Fachberichte Messen Steuern Regeln Nr.10,
Springer Verlag Berlin 1983, pp 589 - 604

<Dij68> Dijkstra E.W.
Cooperating Sequential Processes
in Programming Languages, Ed. F. Genuys,
Academic Press, New York, 1968, pp 43 - 112

<Dio81> Dion L.C.
A Complete Protection Model
Proc. 1981 IEEE Symp. on Security and Privacy
April 1981, pp 49 - 55

<Fab74> Fabry R.S.
Capability Based Adressing
CACM 17,7 (Juli 1974), pp 403 - 412

<Fei77> Feiertag R.J., Levitt K.N., Robinson L.
Proving Multilevel Security of A System Design
Proc. 6th Symp. on Operating System Principles
(Nov 1977), pp 57 - 65

<Fei79> Feiertag R.J., Neumann P.G.
The Foundations of A Provably Secure Operating
System (PSOS)
Proc. AFIPS Nat. Computer Conf. Vol. 48,
AFIPS Press 1979, pp 329 - 334

<Fei80> Feiertag R.J.
A Technique for Proving Specifications Are
Multilevel Secure
Techn. Report CSL-109, Computer Science Lab.,
SRI Intern. Menlo Park Calif., Jan 1980

<Fen74> Fenton J.S.
Memoryless Subsystems
The Computing Journal, Vol. 17, No. 2 (Mai 1974),
pp 143 - 147

<Flo76> Flon L., Habermann A.N.
Towards the Construction of Verifiable Software
Systems
ACM SIGPLAN Notices 1976, pp 141 - 154

<Flo78> Floyd Ch. et al.
PJ Software Engineering
TU Berlin Inst. f. Angewandte Informatik 1978

<Flo81> Floyd Ch., Kopetz H.
Software Engineering - Entwurf und Spezifikation
Berichte des German Chapter of the ACM Nr. 5,
Teubner Verlag Stuttgart 1981

<Fur78> Furtek F.
Constraints and Compromise
in Foundations of Secure Computation, Hrsg.
De Millo et al., Academic Press 1978, pp 189 - 204

<Gew79> Gewald K., Haake G., Pfadler W.
Software - Engineering
Oldenbourg Verlag München 1979

<Göh81a> Göhner P.
Spezifikation der Synchronisierung paralleler
Rechenprozesse in EPOS
Fachtagung Prozeßrechner 1981, Informatik
Fachberichte Nr. 39, Springer Verlag Berlin 1981,
pp 107 - 118

<Göh81b> Göhner P.
Ingenieurgerechte Spezifikation der
Synchronisierung paralleler Rechenprozesse
Dissertation TH Stuttgart 1981

<Gog76> Goguen J.A. et al.
An Initial Algebraic Approach to the Specification, Correctness and Implementation of
Abstract Data Types
in Current Trends in Programming Methodology IV
(Data Structuring), Prentice Hall 1978,
pp 80 - 144

<Gog82> Goguen J.A., Meseguer J.
Security Policies and Security Models
Proc. 1982 IEEE Symp. on Security and Privacy
April 1982, pp 11 - 20

<Gra68> Graham R.M.
Protection in An Information Processing Utility
CACM 11,5 (Mai 1968), pp 365 - 369

<Gra72> Graham G.S., Denning P.J.
Protection - Principles and Practice
Proc. 1972 AFIPS Nat. Computer Conf. Vol. 40,
AFIPS Press 1972, pp 417 - 429

<Gra79> McGraw J.R., Andrews G.R.
Access Control in Parallel Programs
IEEE Trans. on SW-Engineering Vol. SE-5, No. 1,
Jan 1979, pp 1 - 9

<Gro76> Grohn M.J.
A Model of A Protected Data Management System
ESD-TR-76-289, Bedford, Mass., Juni 1976

<Gut76> Guttag J.V.
The Design of Data Type Specifications
Techn. Report, Information Science Instit.
Marina del Rey, Calif., 1976

<Gut77> Guttag J.V.
Abstract Data Types and the Development of Data Structures
CACM 20,6 (Juni 1977), pp 396 - 404

<Gut78> Guttag J.V., Horning J.J.
The Algebraic Specification of Data Types
Acta Informatica 1978

<Har75> Harrison M.A. et al.
On Protection in Operating Systems
Proc. 5th Symp. on Operating System Principles
Nov 1975, pp 14 - 24

<Har75a> Harrison M.A.
On Models of Protection in Operating Systems
in Lecture Notes in Computer Science Vol. 32,
Springer Verlag Berlin 1975, pp 46 - 60

<Har76> Harrison M.A., Ruzzo W.L., Ullman J.P.
Protection in Operating Systems
CACM 19,8 (August 1976), pp 461 - 471

<Har78> Harrison M.A., Ruzzo W.L.
Monotonic Protection Systems
in Foundations of Secure Computation, Hrsg.
De Millo, Academic Press 1978, pp 337 - 363

<Her78> Hermes H.
Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit
Springer Verlag Berlin 1978

<Hoa69> Hoare C.A.R.
An Axiomatic Basis for Computer Programming
CACM 12,10 (Okt 1969), pp 576 - 583

<Hoa71> Hoare C.A.R.
Proof of A Program: FIND
CACM 14,1 (Jan 1971), pp 39 - 45

<Hoa72a> Hoare C.A.R.
Proof of Correctness of Data Representations
Acta Informatica 1 (1972), pp 271 - 281

<Hoa72b> Hoare C.A.R.
Towards A Theory of Parallel Programming
in Operating Systems Techniques, Academic
Press 1972, pp 61 - 71

<Hoa74> Hoare C.A.R.
Monitors: An Operating System Structuring Concept
CACM 17,10 (Okt 1974), pp 549 - 557

<Hof78> Hofer H.
Datenfernverarbeitung
Springer Verlag Berlin 1978

<Hof79> Hofmann F., Keramidis S.
Erfahrungen mit neueren Methoden zur Konstruktion zuverlässiger Software bei der Realisierung eines Vermittlungssystems für Rechnernetze in der Fertigungssteuerung
GI 9. Jahrestagung, Informatik Fachberichte Nr. 19, Springer Verlag Berlin 1979

<How76> Howard J.
Proving Monitors
CACM 19,5 (Mai 1976), pp 273 - 279

<Hul81> Hultzsch H.
Prozeßdatenverarbeitung
Teubner Verlag Stuttgart 1981

<Jac75> Jackson M.A.
Principles of Program Design
Academic Press New York 1975

<Jac76> Jackson M.A.
Data Structures as A Basis for Program Design
in Structured Programming, Infotech 1976

<Jäp77> Jäpel D.
Ein Begriffssystem zur Formulierung von Schutzaspekten und seine Anwendung auf Programmsysteme
IMMD-Berichte Bd. 10, Nr. 7, Univ. Erlangen
Juni 1977

<Jon73> Jones A.K.
Protection in Programmed Systems
Ph.D. Thesis, Carnegie Mellon University 1973

<Jon75> Jones A.K., Lipton R.J.
The Enforcement of Security Policies for Computation
ACM SIGOPS Operating System Review 9,5 (Nov 1975)
pp 197 - 206

<Jon76> Jones A.K., Lipton R.J., Snyder
A Linear Time Algorithm for Deciding Security
Proc. 17th Annual Symp. on Foundations of Computer Science, 1976, pp 33 - 41

<Jon78a> Jones A.K.
Protection Mechanism Models: Their Usefulness
in Foundations of Secure Computation, Hrsg. De Millo et al., Academic Press 1978
pp 237 - 252

<Jon78b> Jones A.K., Lipton R.J.
The Enforcement of Security Policies for Computation
Journal of Computer and System Sciences 17 (1978)
pp 35 - 55

<Jon78c> Jones A.K.
The Object Model: A Conceptual Tool for Structuring Software
in Lecture Notes in Computer Science Bd. 60, Springer Verlag Berlin 1978, pp 8 - 16

<Jon78d> Jones A.K.
Protection Mechanisms and the Enforcement of Security Policies
in Lecture Notes in Computer Science Bd. 60, Springer Verlag Berlin 1978, pp 229 - 251

<Kap79> Kapp K., Daum R.
Sicherheit und Zuverlässigkeit von Automatisierungssoftware
Informatik Spektrum 2 (1979), pp 25 - 36

<Kel76> Keller R.M.
Formal Verifications of Parallel Programs
CACM 19,7 (Juli 1976), pp 371 - 384

<Ker77> Keramidis S.
Ein allgemeines formales Modell für Betriebssysteme und dessen Implementierung
Dissertation Univ. Erlangen 1977

<Ker79> Keramidis S., Friedl A., Steinbauer D.
Entwurf und Implementierung eines Rechnernetzes für die Fertigungssteuerung - Eine Fallstudie zur Konstruktion zuverlässiger Systeme
IMMD-Berichte Bd. 12, Nr. 7, Univ. Erlangen
Sept 1979

<Ker82> Keramidis S.
Eine Methode zur Spezifikation und korrekten Implementierung von asynchronen Systemen
IMMD-Berichte Bd. 15, Nr. 4, Univ. Erlangen
Juni 1982

<Keu82> Keutgen G.
Design Aid for Real Time Systems (DARTS)
KfK-Bericht KFK-PFT 17, Karlsruhe 1982

<Koc79> Koch W.
SPEZI - Eine Sprache zur Formulierung von Spezifikationen
TU Berlin, Institut für Angewandte Informatik
Sept 1979

<Koc80> Koch W.
Ein Beitrag zur Entwicklung entwurfsunterstützender Spezifikationssprachen für Automatisierungssysteme
Dissertation TH Stuttgart 1980

<Kop76> Kopetz H.
Softwarezuverlässigkeit
Carl Hanser Verlag Muenchen 1976

<Koz83> Kozma L., Laborczi Z.
On Implementation Problems of Shared Abstract Data Types
in Lecture Notes in Computer Science Vol. 152, Springer Verlag Berlin 1983, pp 146 - 152

<Kur83> Kurbel K.
Software Engineering im Produktionsbereich
Gabler Verlag Wiesbaden 1983

<Lam69> Lampson B.W.
Dynamic Protection Structures
Proc. 1969 AFIPS Nat. Computer Conf. Vol. 35
AFIPS Press 1969, pp 27 - 38

<Lam71> Lampson B.W.
Protection
Proc. 5th Princeton Symp. on Inform. Sciences and Systems, Princeton University, März 1971, pp 437 - 443

<Lam73> Lampson B.W.
A Note on the Confinement Problem
CACM 16,10 (Okt 1973), pp 613 - 615

<Lan81> Landwehr C.E.
Formal Models of Computer Security
ACM Computing Surveys Vol. 13, No. 3, Sept 1981
pp 247 - 278

<Lan82> Landauer C., Crocker S.
Precise Information Flow Analysis by Program Verification
Proc. 1982 IEEE Symp. on Security and Privacy
April 1982, pp 74 - 80

<Lau76> Lauber R.
Prozeßautomatisierung I
Springer Verlag Berlin 1976

<Lau81> Lauber R.
Zuverlässigkeit und Sicherheit in der Prozeßautomatisierung
Fachtagung Prozeßrechner 1981, Informatik Fachberichte Nr. 39, Springer Verlag Berlin 1981
pp 52 - 64

<Lav78> Laventhal M.S.
Synthesis of Synchronisation Code for Data Abstractions
Ph.D. Thesis, MIT, Lab. for Computer Science 1978

<Lin76> Linden T.A.
Operating System Structures to Support Security and Reliable Software
ACM Computing Surveys Vol. 8, No. 4, Dez 1976
pp 409 - 445

<Lin76a> Linden T.A.
The Use of Abstract Data Types to Simplify Program Modification
ACM SIGPLAN Notices 8,2 (März 1976), pp 12 - 23

<Lip75> Lipner S.B.
A Comment on the Confinement Problem
ACM Operating Systems Review 9,5 (Nov 1975),
pp 192 - 196

<Lip77> Lipton R.J., Snyder L.
A Linear Time Algorithm for Deciding Subject Security
JACM Vol. 24, No. 3 (1977), pp 455 - 464

<Lip78> Lipton R.J., Snyder L.
On Synchronisation and Security
in Foundations of Secure Computation, Hrsg.
De Millo et al., Academic Press 1978
pp 367 - 385

<Lip78a> Lipton R.J., Budd T.A.
On Classes of Protection Systems
in Foundations of Secure Computation, Hrsg.
De Millo et al., Academic Press 1978
pp 281 - 296

<Lis74> Liskov B., Zilles S.
Programming with Abstract Data Types
ACM SIGPLAN Notices Vol. 9, No. 4, April 1974,
pp 50 - 59

<Lis75> Liskov B.
Data Types and Program Correctness
Proc. AFIPS 1975 Nat. Computer Conf Vol. 44,
AFIPS Press 1975, pp 285 - 286

<Lis75a> Liskov B., Zilles S.
Specification Techniques for Data Abstractions
ACM SIGPLAN Notices Vol. 10, No. 6,
Juni 1975, pp 72 - 87

<Lis77> Liskov B. et al.
Abstraction Mechanisms in CLU
CACM 20,8 (Aug 1977), pp 564 - 576

<Lon77> London T.B.
The Semantics of Information Flow
Ph.D. Thesis Dept. of Comp. Sciences, Cornell University 1977

<Lud78> Ludewig J., Streng W.
Überblick und Vergleich verschiedener Mittel für die Spezifikation und den Entwurf von Software
KfK-Bericht 2506, Karlsruhe 1978

<Lud81a> Ludewig J.
PCSL und ESPRESO - zwei Ansätze zur Formalisierung der Prozeßrechner-Softwarespezifikation
Fachtagung Prozeßrechner 1981, Informatik Fachberichte Nr. 39, Springer Verlag Berlin 1981, pp 76 - 86

<Lud81b> Ludewig J.
Zur Erstellung der Spezifikation von Prozeßrechner-Software
Dissertation TU München 1981

<Mac83> Mackert L.
Modellierung, Spezifikation und korrekte Realisierung von asynchronen Systemen
Dissertation Univ. Erlangen 1983

<Mer80> Merbeth G.
Schutzmechanismen in Betriebssystemen
Interner Bericht Institut für Informatik III Nr. 17/80, Univ. Karlsruhe, Juni 1980

<Mil78> Millen J.
Constraints and Multilevel Security
in Foundations of Secure Computation, Hrsg.
De Millo et al., Academic Press 1978
pp 205 - 222

<Mil81> Millen J.
Information Flow Analysis of Formal Specifications
Proc. 1981 IEEE Symp. on Security and Privacy
April 1981, pp 3 - 8

<Min78> Minsky N.H.
An Operation-Control Scheme for Authorisation in
Computer Systems
Int. Journal of Comp. and Inform. Sciences
Juni 1978

<Min81a> Minsky N.H.
On the Transport of Privileges
Proc. 1981 IEEE Symp. on Security and Privacy
April 1981, pp 41 - 48

<Min81b> Minsky N.H.
Locally Controlled Transport of Privileges
Techn. Report Rutgers Univ. LCSR-TR-17, Juni 1981

<Nee77> Needham R.M., Walker R.D.H.
The Cambridge CAP Computer and Its Protection
System
Proc. 6th Symp. on Operating System Principles
1977, p 1 - 10

<Neu77> Neumann P.G. et al.
A Provable Secure Operating System
Final Report 4331, SRI International Menlo Park
Calif., Feb 1977

<Owi76a> Owicki S., Gries D.
Verifying Properties of Parallel Programs: An Axiomatic Approach
CACM 19,5 (Mai 1976), pp 279 - 285

<Owi76b> Owicki S., Gries D.
An Axiomatic Proof Technique for Parallel Programs
Acta Informatica 6 (1976), pp 319 - 340

<Par72a> Parnas D.L.
A Technique for Software Module Specification with Examples
CACM 15,5 (Mai 1972), pp 330 - 336

<Par72b> Parnas D.L.
On the Criteria to Be Used in Decomposing Systems into Modules
CACM 15,12 (Dez 1972), pp 1053 - 1058

<Pop78> Popek G.J., Farber D.A.
A Model for Verification of Data Security in Operating Systems
CACM 21,9 (Sept 1978), pp 737 - 749

<Pop79> Popek G.J. et al.
UCLA Secure UNIX
Proc. 1979 AFIPS Nat. Computer Conf. Vol. 48, AFIPS Press 1979, pp 355 - 364

<Ran75> Randell B.
System Structure for Software Fault-Tolerance
Proc. 1975 Internat. Conf. on Reliable SW, ACM, New York 1975, pp 437 ff.

<Rei79> Reitman R.P.
A Mechanism for Information Control in Parallel Systems
ACM SIGOPS Operating Systems Reviews 13,4 (Dez 1979), pp 55 - 63

<Rei83> Reitenspiess M.
Sprachkonstrukte zur Spezifikation und korrekten Implementation von Schutzproblemen
Dissertation Univ. Erlangen 1983

<Rem79> Rembold U.
Prozeß- und Mikrorechnersysteme
Oldenbourg Verlag München 1979

<Rob77> Robinson L., Roubine O.
SPECIAL - A Specification and Assertion Language
Techn. Report CSL-46, SRI, Jan 1977

<Rot73> Rotenberg L.
Making Computers Keep Secrets
Ph.D. Thesis MAC, MAC TR-116, 1973

<Rüd77> Rüdel W.
Darstellung und Vergleich abstrakter Modelle zur Beschreibung geschützter Systeme
IMMD-Bericht Bd. 10, Nr. 9, Univ. Erlangen Juli 1977

<Rus81a> Rushby J.
Design and Verification of Secure Systems
ACM Operating System Reviews, 1981, p 12 - 21

<Rus81b> Rushby J.
Specification and Design of Secure Systems
Techn. Report Univ. Newcastle SSM/6, 1981

<Rus81c> Rushby J.
Verification of Secure Systems
Techn. Report Univ. Newcastle SSM/9, 1981

<Rus82> Rushby J.
Proof of Separability
Techn. Report Univ. Newcastle SSM/11, 1981

<Sha77> Shankar K.S.
The Total Computer Security Problem: An Overview
The Computer Juni 1977, pp 50 - 73

<Sha82> Shankar K.S.
A Functional Approach to Module Verification
IEEE Trans. on SW-Engineering vol. SE-8, No. 2
März 1982, pp 147 - 160

<Sny81> Snyder L.
Formal Models of Capability-Based Protection Systems
IEEE Trans. on Computers Vol. C-30, No. 3
März 1981, pp 172 - 181

<Ste83> Steusloff H.
Rechnergestützter Entwurf von Automatisierungssystemen
in Fachberichte Messen Steuern Regeln, Nr. 10
Springer Verlag Berlin 1983, pp 531 - 550

<Sto81> Stoughton A.
Access Flow - A Protection Model Which Integrates Access Control and Information Flow
Proc. 1981 IEEE Symp. on Security and Privacy
April 1981, pp 9 - 18

<Sto83> Stoyan H., Wedekind H.
Objektorientierte Software- und Hardwarearchitekturen
Berichte des German Chapter of the ACM Nr. 15,
Teubner Verlag Stuttgart 1983

<Ten76> Tennent R.D.
The Denotational Semantics of Programming Languages
CACM 19,8 (August 1976), pp 437 - 453

<Wal74> Walter K.G. et al.
Primitive Models for Computer Security
Dept. of Comp. and Information Sciences, Case Western Univ., NTIS REP AD-778467, Jan 1974

<Wal75a> Walter K.G. et al.
Structured Specification of A Security Kernel
ACM SIGPLAN Notices 10,6, Juni 1975, pp 285 - 293

<Wal75b> Walter K.G. et al.
Initial Structured Specifications for An Uncompromisable Computer Security System
ESD-TR-75-82, Bedford Mass., Juli 1975

<Wal80> Walker B.J., Kemmerer R.A., Popek G.J.
Specification and Verification of the UCLA UNIX Security Kernel
CACM 23,2 (Feb 1980), pp 118 - 131

<Web79> Weber G.
Methoden der Zuverlässigkeitsanalyse und -sicherung bei Hardware und Software
in Zuverlässigkeit von Rechenanlagen, Görke,
Oldenbourg Verlag München 1979, pp 72 - 96

<Web83> Weber K.
Modellierung von Fehlverhalten mit Berücksichtigung paralleler Abläufe
Dissertation Univ. Erlangen 1983

<Wei69> Weissmann C.
Security Controls in the ADEPT-50 Time Sharing System
Proc. 1969 AFIPS Nat. Computer Conf. vol. 35, AFIPS Press 1969, p 119 - 133

<Wu 81> Wu M.S.
Hierarchical Protection Systems
Proc. 1981 IEEE Symp. on Security and Privacy
April 1981, pp 113 - 123

<Wul76> Wulf W.A., London R.L., Shaw M.
Abstraction and Verification in ALPHARD: Introduction to Language and Methodology
Techn. Report Carnegie Mellon University, Dept. of Computer Sciences, Juni 1976

Lebenslauf

1952	Anton Friedl, geb. am 10.11.1952 in Nürnberg Eltern: Georg Friedl Emmi Friedl, geb. Strobl
1959 - 1963	Volksschule Odenberger Straße in Nürnberg
1963 - 1972	Willstätter Gymnasium Nürnberg Abitur im Juni 1972
1972 - 1978	Studium der Informatik an der Universität Erlangen - Nürnberg
1974 - 1977	Studentische Hilfskraft am Lehrstuhl III des Instituts für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen - Nürnberg
1978	Diplom Hauptprüfung im Fach Informatik an der Universität Erlangen - Nürnberg am 5.10.1978
1978 - 1982	Vertriebsingenieur bei der Fa. SIEMENS AG, Erlangen, Unternehmensbereich Energie- und Automatisierungstechnik
seit 1982	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Fertigungsautomatisierung und Produktionssystematik der Universität Erlangen - Nürnberg (Leiter: Prof. Dr.-Ing. K. Feldmann)