Sicherheit in verteilten, objektorientierten Systemen

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Thomas Riechmann

Erlangen - 1999

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

Tag der Einreichung:	23. 4. 1999
Tag der Promotion:	12. 7. 1999
Dekan:	Prof. Dr. G. Herold
Berichterstatter:	Prof. Dr. F. Hofmann
	Prof. Dr. S. Jablonski

Kurzfassung

Für prozedurale und objektbasierte Programmierung von verteilten Systemen existieren heutzutage bereits viele ausgereifte, auf Capabilities und Zugriffslisten basierende Sicherheitsmodelle. Diese Modelle sind jedoch nicht für objektorientierte Programmierung von verteilten Anwendungen geeignet, da die hohe Abstraktionsebene und Dynamik der objektorientierten Programmierung eine Anpassung der Anwendungsklassen aller möglicherweise interagierender Objekte an die Sicherheitsanforderungen der Anwendung erfordert. Die Sicherheitskonfiguration wird dadurch hochkomplex und fehleranfällig.

Diese Arbeit stellt ein Sicherheitsmodell vor, das auf Sicherheitsmetaobjekten basiert. Sicherheitsmetaobjekte sind spezielle Objekte, die an eine Applikation gebunden werden können und eine für die Anwendung nötige Sicherheitsstrategie realisieren. Durch die Wahl entsprechender Sicherheitsmetaobjekte kann so eine Anwendung ohne Anpassung der Anwendungsklassen in verschiedenen Umgebungen mit unterschiedlichen Sicherheitsanforderungen eingesetzt werden.

Neben den klassischen Mechanismen wie Capabilities und Zugriffslisten können mit Sicherheitsmetaobjekten auch die mit klassischen Systemen nicht realisierbaren Sicherheitsstrategien "transitive Capabilities" und "rollenbasierte Identitäten" implementiert werden. Diese lösen die speziellen Sicherheitsprobleme objektorientierter Systeme, nämlich die fehlende Verbreitungskontrolle und das Trojanische-Pferd-Problem von Referenzen.

Sicherheitseigenschaften des Modells werden mittels einer Formalisierung von Teilaspekten bewiesen. Die Implementierbarkeit des Modells wird anhand der Beschreibung einer prototypischen Implementation gezeigt.

Inhaltsverzeichnis

Abbildungsverzeichnis . . . . . . *v*
Programmverzeichnis . . . . . . *vii*
Tabellenverzeichnis . . . . . . *ix*

1 Einleitung *1*
1.1 Objektorientierung und Sicherheit *2*
1.2 Ziele der Dissertation *3*
1.3 Gliederung der Arbeit *4*

2 Sicherheitsmodelle *5*
2.1 Bewertung von Sicherheitsmodellen *5*
2.2 Implementation von Sicherheitsmodellen *6*
2.3 Zugriffsmatrix-Modell *8*
2.3.1 Zugriffslisten *9*
2.3.2 Capabilities *9*
2.4 Bell - La Padula Modell *10*
2.5 Rollenbasierte Zugriffskontrolle *11*
2.6 Kryptographie *12*

3 Sicherheit in verteilten Systemen *17*
3.1 Betriebssysteme *17*
3.1.1 Hydra *17*
3.1.2 Amoeba *19*
3.1.3 Spring *20*
3.2 Verteilte Ablaufsysteme *21*
3.2.1 Corba *21*
3.2.2 DCOM *23*
3.2.3 DCE *25*
3.2.4 Java *25*
3.2.5 Andere Ablaufsysteme *28*
3.3 Vergleich *29*
3.4 Zusammenfassung *30*

4 Schutz durch Sicherheitsmetaobjekte 33
4.1 Grundlagen der Meta-Programmierung 33
4.2 Sicherheitsmetaobjekte 34
4.3 Konfigurierbare Capabilities 36
4.3.1 Einfacher Schutz 36
4.3.2 Einfache Transitivität 39
4.3.3 Komplette Transitivität 41
4.4 Subjekte / Zugriffslisten 43
4.4.1 Einfache Zugriffskontrolllisten 44
4.4.2 Virtuelle Domänen 46
4.4.3 Hierarchische Domänen und gesamtes System 50
4.4.4 Interne Referenzen 51
4.4.5 Realisierung 51
4.5 Rollenbasierte Identitäten 53
4.5.1 Druckverteiler mit domänenbasierten Identitäten 54
4.5.2 Druckverteiler mit threadbasierten Identitäten 55
4.5.3 Rollenbasierte disjunkte Identitäten 56
4.5.4 Hierarchische Domänen 58
4.5.5 Explizite Identitäten 60
4.5.6 Zusammenfassung: Rollenbasierte Identitäten 61
4.6 Elementare Wertobjekte 62
4.7 Reduktion von SMO Ketten 63
4.8 Meta-Hierarchien 67
4.9 Gesamtes System 68
4.10 Realisierung verschiedener Modelle 69
4.11 Vergleich 72
4.12 Problemlösungen 73
4.13 Zusammenfassung und Ausblick 74
5 Formales Modell 77
5.1 Reduktion von SMO Ketten 77
5.1.1 Identitätsmodifizierende SMOs 79
5.1.2 Identitätsbenutzende SMOs 79
5.1.3 Identitätsneutrale SMOs 80
5.1.4 Komposition und Inverse 80
5.1.5 Reduktion doppelter SMOs 81
5.1.6 Vertauschung von SMOs 82
5.1.7 Nicht vertauschbare SMOs 84
5.1.8 Grenzen des Modells 87
5.1.9 Zusammenfassung 87

5.2 Virtuelle Domänen und rollenbasierte Identitäten *88*
5.2.1 Basisdefinitionen *88*
5.2.2 Domänen *89*
5.2.3 Methodenaufrufe *91*
5.2.4 Beweis der Domänenerhaltung *94*
5.2.5 Innere Schleifen *97*
5.2.6 Identitäten und Startwerte *98*
5.2.7 Beispiel: Disjunkte Interaktion *99*
5.2.8 Beispiel: Hierarchische Domänen *103*
5.2.9 Grenzen des Modells *105*

5.3 Zusammenfassung *105*

6 Implementation / Prototyp *107*

6.1 Implementationsplattform *107*

6.2 Eigenschaften der Proxy-Implementation *108*
6.2.1 Proxy-Fähigkeit *108*
6.2.2 Transparenz der Proxies *110*
6.2.3 Objektschutz *110*
6.2.4 Auswahl des Zielsystems *111*

6.3 Realisierung: Lokales Modell *111*

6.4 Realisierung: Verteiltes Modell *113*
6.4.1 Vertrauensbeziehungen *114*
6.4.2 Einheiten des verteilten Systems und Darstellung *114*
6.4.3 Initiale Referenzen *117*
6.4.4 Implementierte Beispiele *118*

6.5 Effizienz *118*

6.6 Einschränkungen der Implementation *120*
6.6.1 Lokales Modell *120*
6.6.2 Verteiltes Modell *120*

6.7 Bewertung *120*

6.8 Zusammenfassung *121*

7 Zusammenfassung und Ausblick *123*

7.1 Zusammenfassung der Arbeit *123*

7.2 Weiterführende Arbeiten *125*

8 Literatur *127*

Abbildungsverzeichnis

Einleitung

Sicherheitsmodelle

Abbildung 2.1 Zugriffsmatrix 8
Abbildung 2.2 Ver- und Entschlüsselung von Daten 12
Abbildung 2.3 Verwendung von Verschlüsselung zur Geheimhaltung von Daten 12

Sicherheit in verteilten Systemen

Schutz durch Sicherheitsmetaobjekte

Abbildung 4.1 Eine mit einem Sicherheitsmetaobjekt geschützte Referenz 34
Abbildung 4.2 Quell- und zielorientiert angeheftete SMOs 35
Abbildung 4.3 Schutz von Referenzen 37
Abbildung 4.4 Einfache Transitivität 40
Abbildung 4.5 Mißbrauch von Referenzen als trojanisches Pferd 41
Abbildung 4.6 Komplette Tansitivität 42
Abbildung 4.7 Identitäts-SMO und Zugriffslisten-SMO 44
Abbildung 4.8 Versehentliche Weitergabe von identitätsbehafteten Referenzen 46
Abbildung 4.9 Virtuelle Domänen 47
Abbildung 4.10 Virtuelle Domäne mit geschütztem Rand 48
Abbildung 4.11 Ankommende externe Referenz 48
Abbildung 4.12 Ankommende interne Referenz mit Entfernung des Grenz-SMO 49
Abbildung 4.13 Ankommende interne Referenz mit Anfügen des Grenz-SMO 49
Abbildung 4.14 Abgehende interne Referenz über or1 / or2 49
Abbildung 4.15 Abgehende externe Referenz or3 über or1 / or2 50
Abbildung 4.16 Verschachtelte Domänen 50
Abbildung 4.17 Interne Referenz wird herausgegeben 51
Abbildung 4.18 Herausgabe von Referenz or1 53
Abbildung 4.19 Rollenbasierte Identitäten 54
Abbildung 4.20 Druckverteiler mit domänenbasierter Identität 55
Abbildung 4.21 Rollenbasierte disjunkte Identitäten 56
Abbildung 4.22 Versuch, eine Referenz unterzuschieben 57
Abbildung 4.23 Sicherheitsprobleme bei nicht-disjunkter Interaktion 58

Abbildung 4.24 Druckverteiler mit hierarchischen Domänen 59
Abbildung 4.25 Unterschieben von Referenzen bei hierarchischen Domänen 59
Abbildung 4.26 Die untergeschobene Referenz “ausgeklappt” 60
Abbildung 4.27 Explizite Identitäten 61
Abbildung 4.28 Beispiel: Entfernung mehrfach angehefteter SMOs 63
Abbildung 4.29 Grenz-SMO-Ketten: Es können nur äußere Schleifen entstehen 64
Abbildung 4.30 Beispiel: Referenz und inverse Referenz 65
Abbildung 4.31 Beispiel: Reduktion mehrfach angehängter SMOs 66
Abbildung 4.32 Denkbare Meta-Hierarchien 67
Abbildung 4.33 Nameserver 68
Abbildung 4.34 Schutz gegen Referenzspeicherung 70
Abbildung 4.35 Lösung des Einschluß-Problems 70

Formales Modell

Abbildung 5.1 Informationsfluß bei Aufrufen 78
Abbildung 5.2 Virtuelle Domänen 89
Abbildung 5.3 Beispiel für eine Objektreferenz 91
Abbildung 5.4 Abbildung von verschachtelten Aufrufen 92
Abbildung 5.5 Innere SMO Schleifen sind für die meisten Modelle unzulässig 98
Abbildung 5.6 Rollenbasierte disjunkte Identitäten 99
Abbildung 5.7 Druckverteiler mit hierarchischen Domänen 103

Implementation / Prototyp

Abbildung 6.1 Implementation von SMOs durch Proxies 108
Abbildung 6.2 Realisierung von Proxies in typisierten Sprachen 109
Abbildung 6.3 Symmetrische vs. asymmetrische Authentifizierung 112
Abbildung 6.4 Vertrauensbeziehungen 114
Abbildung 6.5 Methodenaufruf 115
Abbildung 6.6 Capability-SMO-behaftete, rechnerübergreifende Referenzen 116
Abbildung 6.7 Authentifizierung von rechnerübergreifenden Aufrufen 117

Zusammenfassung und Ausblick

Literatur

Programmverzeichnis

Einleitung

Sicherheitsmodelle

Sicherheit in verteilten Systemen

Schutz durch Sicherheitsmetaobjekte

Listing 4.1 Ein Sicherheitsmetaobjekt für zeitlich begrenzte Gültigkeit 36
Listing 4.2 Ein Sicherheitsmetaobjekt für Revokation von Referenzen 38
Listing 4.3 Ein Sicherheitsmetaobjekt für Restriktion von Referenzen 38
Listing 4.4 Schutz von abgehenden Referenzen 40
Listing 4.5 Die Such-Methode der Liste. 41
Listing 4.6 Komplette Transitivität 42
Listing 4.7 Ein Zugriffslisten-SMO 45
Listing 4.8 Ein Identitäts-SMO 45
Listing 4.9 Unterschiedliche Authentifizierung über die gleiche Referenz 46
Listing 4.10 Implementation eines Grenz-Identitäts-SMO 52
Listing 4.11 Implementation eines disjunkten Grenz-Identitäts-SMOs 58
Listing 4.12 Implementation eines Anonym-Grenz-Identitäts-SMOs 60

Formales Modell

Implementation / Prototyp

Listing 6.1 Binden von SMOs im Prototyp 111

Zusammenfassung und Ausblick

Literatur

Tabellenverzeichnis

Einleitung

Sicherheitsmodelle

Sicherheit in verteilten Systemen

Tabelle 3.1 Vergleich der Betriebs- und Ablaufsysteme 30

Schutz durch Sicherheitsmetaobjekte

Tabelle 4.1 Optimierung hintereinander hängender SMOs 65
Tabelle 4.2 Vertauschung von SMOs 66
Tabelle 4.3 Vergleich klassische Systeme – Sicherheitsmetaobjekte 72
Tabelle 4.4 Problemlösungen mit klassischen Systemen im Vergleich zu SMOs 73

Formales Modell

Implementation / Prototyp

Tabelle 6.1 Meßwerte für lokale Interaktion 118
Tabelle 6.2 Meßwerte für verteilte Interaktion 119

Zusammenfassung und Ausblick

Literatur

Einleitung

Vor etwas mehr als 10 Jahren begann die Entwicklung des Internets. Weltweite Rechnerkommunikation wurde möglich. Zunächst wurde das Internet nur für Forschungs- und Experimentalzwecke genutzt. Heute ist es ein nicht mehr hinwegzudenkender Teil der Rechnerwelt. Einkaufen, Kontoverwaltung, Informationsabruf, Übermitteln von Nachrichten sind Tätigkeiten, die über das Internet ausgeführt werden.

Durch die vielen Anbieter im Internet und die vielen Benutzer sind jedoch neue Probleme entstanden: Sicherheitsprobleme. Aufgrund der hohen Komplexität der heutigen Software enthält nahezu jede Anwendung, die über das Internet verwendet werden kann, Sicherheitslöcher, die zum Einbruch in einen Rechner genutzt werden können.

Bis vor ca. fünf Jahren spielte dieses Problem nur eine untergeordnete Rolle. Es gab nur einen kleinen Kreis von Personen, die das Wissen und den Willen zur Nutzung von Sicherheitslöchern besaßen. Dies hat sich in den letzten Jahren geändert. Im Internet sind Programme frei verfügbar, die Sicherheitslöcher in bestimmter Software ausnutzen und einen Einbruch in Rechner ermöglichen, auf denen diese Software läuft. Es gibt einen sehr großen Kreis von Personen, die diese Programme nutzen; die Rechner der Universität Erlangen werden beispielsweise mehrmals pro Tag von solchen Personen aus dem Internet auf Einbruchsmöglichkeiten untersucht. Mit der zunehmenden Kommerzialisierung des Internets (Kontoverwaltung, Zahlungen über das Internet) ist ein weiterer Anstieg von Einbruchsversuchen zu erwarten – für einen Kriminellen ist es dann möglich, durch einen erfolgreichen Einbruch Geld zu erlangen. Es ist daher nötig, die Software gegen Einbruchsmöglichkeiten besser abzusichern.

Die Programme und Betriebssysteme haben in den letzten Jahren stark an Komplexität und Größe zugenommen. Das in der prozeduralen Programmiersprache C implementierte Betriebssystem Solaris besteht beispielsweise aus mehr als drei Millionen Programmzeilen. Solche riesigen Softwarepakete lassen sich kaum noch warten: Für Erweiterungen oder Änderungen muß eine gewaltige Menge von Code auf Auswirkungen der Änderungen überprüft werden.

Bei neu entwickelten Programmen wird daher oft ein anderes Programmierparadigma angewandt, das eine modularere Vorgehensweise erlaubt: die Objektorientierung ([Boo94], [Mey90]). Man zerlegt die zu implementierende Anwendung in Objekte, die einzeln gewisse Funktionalität bieten und weitgehend unabhängig von anderen Objekten ihre Aufgabe verrich-

ten können. Die Objekte bzw. die Klassen, aus denen die Objekte instantiiert werden, können für verschiedene Applikationen wiederverwendet werden, man kann generische Klassenbibliotheken entwerfen. Es zeigt sich, daß durch die Objektorientierung einerseits die zu implementierende Codemenge insgesamt kleiner wird, andererseits die Anwendung leichter zu warten und zu erweitern ist.

Es stellt sich nun die Frage, ob sich durch die Modularisierung und Codereduktion der Objektorientierung auch Vorteile für Anwendungen ergeben, in denen Sicherheit eine Rolle spielt.

1.1 Objektorientierung und Sicherheit

In den letzten Jahren wurden verschiedene objektorientierte, verteilte Laufzeitsysteme entwikkelt, die objektorientierte, rechnerübergreifende Programmierung ermöglichen. Verschiedene Corba[1]-Implementationen, DCOM[2] und Java sind Beispiele dafür. Die Verteilung ist dabei meist weitgehend transparent, so daß man die gleichen Vorteile wie bei lokal ablaufenden Applikationen nutzen kann: hohe Abstraktion, Wiederverwendung und dadurch Codereduktion.

Wenn solche Applikationen jedoch beispielsweise im Internet eingesetzt werden sollen, stellt sich heraus, daß die Sicherheitsbetrachtungen bei objektorientierten Applikationen schwieriger als bei prozeduralen Applikationen sind. Prozedurale Applikationen haben wohldefinierte Interaktionsschnittstellen zwischen verschiedenen Applikationsteilen: die Prozeduren, die (möglicherweise mittels Fernaufruf über das Netzwerk) angesprochen werden können. Objektorientierte Applikationen haben dies nicht. Je nachdem, welche Objektreferenzen zwischen zwei Applikationsteilen ausgetauscht werden, verändern sich die Schnittstellen zwischen den Teilen und damit die aufrufbaren Methoden. Potentiell muß jedes Objekt darauf vorbereitet sein, mit anderen Applikationsteilen zu interagieren und entsprechende Sicherheitsmaßnahmen implementieren. Man erhält eine Komplexitätsanomalie: Die Applikationsprogrammierung wird durch Objektorientierung vereinfacht, die Implementation einer Sicherheitsstrategie wird komplexer. Dies führt dazu, daß zur Zeit viele verteilte Applikationen aus Sicherheitsüberlegungen heraus weiterhin prozedural implementiert werden.

Die klassischen Zugriffsschutzmechanismen allein (Zugriffslisten und Capabilities) reichen als Sicherheitsmechanismen für objektorientierte Systeme nicht aus. Zwar scheinen auf den ersten Blick Capabilities sogar besonders gut für objektorientierte Systeme geeignet zu sein, da eine Objektreferenz bereits eine Capability für die Schnittstelle eines Zielobjektes ist und damit das Konzept der Capability ein Bestandteil der objektorientierten Programmierung ist. Das Sicherheitsmodell der objektorientierten Plattform Java basiert aus diesem Grund beispielsweise primär auf Capabilities. Allerdings ist das Problem der Verbreitungskontrolle (ein bei Capabilities generell auftretendes Problem) in objektorientierten Systemen wegen der hohen Dynamik viel schwerwiegender als bei prozeduralen Systemen. Zugriffslisten wiederum sind problematisch,

1. Common Object Request Broker Architecture
2. Distributed Component Object Model

da sie schwer zu warten sind: Bei objektorientierten Applikationen mit vielen tausend Objekten muß möglicherweise für jedes Objekt überlegt werden, mit welchem Schutz es ausgestattet werden muß.

1.2 Ziele der Dissertation

Ziel dieser Arbeit ist die Entwicklung eines Sicherheitsmodells für objektorientierte, verteilte Systeme. Das Sicherheitsmodell soll einem Programmierer von objektorientierten Applikationen ermöglichen, die Applikationen möglichst ohne Anpassung seiner Applikationsklassen in Umgebungen laufen zu lassen, in denen Schutz zwischen den Objekten der Applikation wichtig ist. Die den Sicherheitsanforderungen entsprechende Konfiguration soll dabei möglichst getrennt von der Applikationsprogrammierung festgelegt werden können.

Im Rahmen der Arbeit werden die Sicherheitsmodelle von verschiedenen objektbasierten und objektorientierten Systemen betrachtet. Es wird sich herausstellen, daß diese Modelle der klassischen Systeme nur für objektbasierte und prozedurale Programmierung, nicht aber für objektorientierte Programmierung mit starker Interaktion (d.h. häufiger Austausch von Objektreferenzen) geeignet sind. In dieser Arbeit wird daher ein neues Sicherheitsmodell speziell für objektorientierte Programmierung entwickelt. Die Hauptziele sind:

- **Weitgehende Separation des Sicherheitscodes vom Anwendungscode**
  Der Anwendungscode sollte möglichst keine Sicherheitsmechanismen und keine Sicherheitsstrategie beinhalten. Dies wird durch Meta-Abstraktion erreicht: Die Anwendungsklassen müssen nichts über die Sicherheitsanforderungen wissen. Vielmehr wird auf einer anderen, übergeordneten Ebene, der Meta-Ebene, festgelegt, welche Sicherheitsanforderungen für die einzelnen Objekte und Objektreferenzen gelten.

- **Hohe Konfigurierbarkeit der Sicherheitsstrategie**
  Es sollte für alle Anforderungen die nötige Sicherheitsstrategie wählbar sein. Dazu wird besonderer Wert darauf gelegt, daß – neben den neuen, nicht mit klassischen Systemen realisierbaren Strategien, die für objektorientierte Programmierung nötig sind – auch alle wichtigen klassischen Modelle realisierbar sind.

- **Einfacher Mechanismus**
  Der Mechanismus, auf dem das Sicherheitsmodell basiert, sollte möglichst einfach sein. Dies hat mehrere Vorteile: Eine Formalisierung des Modells oder eines Teils des Modells ist weniger komplex und ermöglicht daher umfassendere Beweise die Sicherheit von bestimmten Situationen betreffend. Auch die Implementation ist leichter, da nur der (einfache) Mechanismus implementiert werden muß.

- **Einfache Realisierung einer Sicherheitsstrategie**
  Eine spezielle Sicherheitsstrategie sollte sich möglichst einfach implementieren lassen. Dazu wird in dieser Arbeit ein Programmiermodell für Sicherheitsmetaobjekte entwik-

kelt, das es ermöglicht, die schon bei der Anwendungsprogrammierung verwendete Vereinfachung durch Objektorientierung zu nutzen: Wiederverwendbarkeit von Klassen und hohes Abstraktionsniveau.

1.3 Gliederung der Arbeit

Die Arbeit ist wie folgt aufgebaut: Im folgenden Kapitel werden die Grundlagen von Sicherheitsmodellen erläutert. Die Designkriterien für klassische Sicherheitsmodelle werden herausgestellt, und die wesentlichen Sicherheitsmechanismen und Sicherheitsparadigmen klassischer Systeme werden erklärt und bewertet.

In Kapitel 3 werden heutige Betriebssysteme und verteilte Systeme betrachtet, die besondere Mechanismen für Sicherheit beinhalten. Die meisten der betrachteten Systeme erlauben objektbasierte Programmierung. Hauptaugenmerk liegt daher auf zwei Punkten: Einerseits wird untersucht, ob sich auch Sicherheitsmodelle für feingranulare, objektorientierte Programmierung mit häufigem Austausch von Objektreferenzen realisieren lassen, andererseits werden die von den Systemen unterstützten Sicherheitsmodelle herausgearbeitet, um für das in der Arbeit entwickelte Sicherheitsmodell einen Anhaltspunkt für die heute gebräuchlichen und daher zumindest in einigen Situationen nötigen Sicherheitsmechanismen zu bekommen.

Kapitel 4 enthält die Beschreibung des in dieser Arbeit entwickelten Sicherheitsmodells. Der grundlegende Mechanismus – Sicherheitsmetaobjekte – wird beschrieben und verschiedene Anwendungen werden erklärt. Es werden für die Objektorientierung benötigte Sicherheitsstrategien neu entwickelt; transitive Capabilities und rollenbasierte Identitäten sind die wichtigsten solchen Strategien. Es wird aber auch gezeigt, daß die klassischen Mechanismen, zu denen Zugriffslisten mit domänenbasierten Identitäten und einfache Capabilities zählen, implementierbar sind.

In Kapitel 5 werden zwei wichtige Teilaspekte des Sicherheitsmodells formalisiert: die Reduktion von redundanten Sicherheitsmetaobjekt-Ketten und virtuelle Domänen. Dies sind die komplexesten Teile des in Kapitel 4 eingeführten Sicherheitsmodells. Durch die Formalisierung werden die dort schon beschriebenen Sicherheitseigenschaften formal bewiesen.

In Kapitel 6 wird der im Rahmen dieser Arbeit implementierte Prototyp beschrieben. Es wird erläutert, inwieweit der Prototyp das Modell implementiert und wie eine volle Implementation aussehen könnte. Gleichzeitig zeigt die Prototypimplementation, an welchen Stellen bei einer vollen Implementation des Modells mit Performanz-Problemen zu rechnen ist und welche Optimierungen möglich sind.

2 Sicherheitsmodelle

In diesem Kapitel werden grundlegende Sicherheitsmechanismen und Sicherheitsmodelle dargestellt. Es wird untersucht, wie man Sicherheitsmodelle bewerten kann und welche Mechanismen es gibt, auf denen Sicherheitsmodelle basieren können.

2.1 Bewertung von Sicherheitsmodellen

Ein Sicherheitsmodell [Mcl94] beschreibt Eigenschaften eines Systems, die die Sicherheit, d.h. Dinge wie Vertraulichkeit, Verfügbarkeit oder Integrität des Systems betreffen. Sicherheitsmodelle können formal spezifiziert werden, für viele Systeme existiert allerdings nur eine informelle Definition.

Um ein Sicherheitsmodell oder einen Sicherheitsmechanismus bewerten zu können, benötigt man Kriterien, an denen sich seine Qualität ablesen läßt. Saltzer und Schroeder [SaS75] haben dazu einen Katalog von Design-Kriterien aufgestellt, mit dem Sicherheitsmodelle bewertet werden können, und der für die Entwicklung von Sicherheitsmodellen als Grundlage dienen kann.

- **Einfachheit des Designs**
  Das Design des Sicherheitsmodells sollte so klein und einfach wie möglich sein, um ungewollte Zugriffsmöglichkeiten, die durch schwer durchschaubare Komplexität des Modells entstehen können, auszuschließen.
- **Sichere Standardwerte**
  Das Modell sollte auf Erlaubnisbasis arbeiten, nicht auf Verbotsbasis. Das bedeutet alle Zugriffe sind zunächst verboten. Jeder Zugriff muß explizit gestattet werden, sonst schlägt er fehl.
- **Komplette Überwachung**
  Alle Zugriffe sollten überwacht werden, d.h. jeder Zugriff sollte geprüft werden und gemäß der Sicherheitsstrategie und dem Sicherheitsmodell zugelassen oder verboten werden.

- **Offenes Design**
  Die Sicherheit sollte nicht darauf basieren, daß die Sicherheitsmechanismen geheimgehalten werden.

- **Teilung der Rechte**
  Falls möglich sollten die Rechte geteilt werden. Es soll nicht eine Person oder ein Programmteil allein auf eine gewisse Ressource zugreifen können, sondern nur mehrere gemeinsam.

- **Kleinstmögliches Zugriffsrecht**
  Jeder Programmteil und jeder Benutzer sollte nur mit den Rechten ausgestattet sein, die er unbedingt benötigt, um die von ihm durchzuführenden Aufgaben erledigen zu können.

- **Kleinstmögliche Gemeinsamkeit**
  Unabhängige Programmteile sollten nur möglichst wenige Dinge gemeinsam nutzen, da eine gemeinsame Nutzung von Ressourcen immer die potentielle Gefahr birgt, daß ein Programmteil einen anderen böswillig beeinflussen kann.

- **Psychologische Akzeptanz**
  Das Modell sollte intuitiv sein, da sonst der Programmierer bzw. der Benutzer das Modell möglicherweise fehlerhaft verwendet.

2.2 Implementation von Sicherheitsmodellen

Um ein Sicherheitsmodell zu verwenden, benötigt man eine Implementation des Modells. Die Implementation muß genau das Modell realisieren, sonst können Sicherheitsprobleme auftreten, die in der Modellspezifikation nicht vorhanden sind. Dazu könnte man fordern, daß alle implementierten Komponenten formal verifiziert werden. Dies ist jedoch bei den meisten Systemen wegen der hohen Komplexität nicht möglich. Es stellt sich daher die Frage, ob dies tatsächlich nötig ist, oder ob man sich bei der Verifikation auf Kernbereiche beschränken kann. Das Orange Book [Dod83] definiert dazu eine sogenannte vertrauenswürdige Berechnungsbasis (Trusted Computing Base, TCB). Diese soll das gewünschte Sicherheitsmodell implementieren und unabhängig vom Rest des Systems die Realisierung des Modells garantieren. Die TCB wird zumindest einen Teil des Betriebssystems eines Rechners und möglicherweise einen Teil der administrativen Programme umfassen. Wenn die Implementation der TCB korrekt ist und sie sich durch nicht in der TCB befindliche Komponenten nicht verändern läßt (diese Eigenschaft muß Teil des Modells sein), realisiert sie das Modell.

Beide Forderungen sind in der Praxis schwierig zu realisieren:

- Da die Implementation der TCB oft sehr komplex ist, läßt sie sich meist nicht verifizieren. Damit kann die Korrektheit nicht sichergestellt werden.

- Die meisten Betriebssysteme erlauben zumindest administrativen Programmen und Administratoren die Modifikation des Betriebssystems und damit der TCB. Diese administrativen Programme müssen, um die obige Forderung der Nichtmodifizierbarkeit zu erfüllen, auch in die TCB aufgenommen werden, wodurch die TCB noch größer wird.

Um einen Eindruck von der Größe der TCB zu bekommen, wird im folgenden überlegt, wie die TCB bei einigen heute üblichen Betriebssystemen aussieht. Da es bei den meisten Betriebssystemen kein spezifiziertes Sicherheitsmodell gibt, muß überlegt werden, wie das jeweilige Sicherheitsmodell aussehen könnte und was die TCB dann umfassen muß.

- **Windows 95 und Windows 98**
  Windows 95/98 [Mon98] kennt keinen Schutz bei lokal ausgeführten Programmen. Jedoch bietet es z.B. Schutz gegen unbefugten Zugriff über Netzwerk auf Dateien auf der Festplatte. Dies könnte das Sicherheitsmodell von Windows 95/98 beinhalten. Da alle auf dem Rechner laufenden Programme vollen Schreibzugriff auf Betriebssystemdateien besitzen, muß die TCB mindestens das Betriebssystem und alle auf dem Rechner laufenden Programme umfassen.

- **Unix**
  Bei Unix [Hie93] könnte das Sicherheitsmodell Dinge, wie den Dateischutz (Schutz gegen Zugriff fremder Benutzer) umfassen. Bei Unix muß die TCB dann mindestens den Betriebssystemkern mit ladbaren Modulen, sowie alle Programme, die vom Administrator (root) aufgerufen werden oder mit seinen Rechten laufen (S-Bit), umfassen.

- **Java**
  Bei Java [Fla96] könnte das Sicherheitsmodell Schutz gegen unbefugten Objektzugriff und unbefugten Ressourcenzugriff (Dateien etc.) umfassen. Die TCB muß dann die virtuelle Java-Maschine umfassen sowie alle privilegierten Klassen (das sind bei Java alle lokalen Klassen wie Fensterklassen, Hilfsklassen, etc.), da diese Klassen Maschinenprogramme ausführen können und damit in der Lage sind, die virtuelle Maschine zu modifizieren. Bei einem Java-Laufzeitsystem, das auf einem anderen Betriebssystem aufsetzt (z.B. auf Unix oder Windows 95/98 im Gegensatz zu JavaOS [Mad96]), muß die TCB zusätzlich die entsprechenden Teile des Betriebssystems umfassen.

Bei allen vorgestellten Beispielen ist die TCB sehr groß. Dies ist nicht nur eine Eigenschaft der betrachteten Systeme, eine Ursache ist vielmehr auch das TCB-Konzept selbst [BlK97]. Außerdem realisiert die TCB in den Beispielen nur Schutz zwischen verschiedenen Benutzern oder Programmen. Falls ein Benutzer unbedacht Programme ausführt, kann es sein, daß diese seine Vorstellung vom Sicherheitsmodell verletzen, obwohl das Modell tatsächlich nicht verletzt wird. Solch ein Programm kann z.B. (bei Windows und Unix) alle Dateien des Benutzers lesen und an einen Dritten übermitteln oder sich in Dateien des Benutzers einnisten und später unerwünschte Aktivitäten einleiten, z.B. Dateien löschen, Dateien ändern oder Dateiinhalte an Dritte übermitteln. Solche Programme werden auch als Viren oder trojanische Pferde bezeichnet. Es kann auch vor dem Benutzer verborgen Dienste anbieten, die Dritten Zugriff auf die dem Benutzer zugänglichen Ressourcen erlauben. Solche Programme werden als Backdoor-Programme bezeichnet.

Falls die TCB fehlerhaft ist, nämlich in gewissen Fällen nicht das Modell realisiert, kann man meist keine Aussage mehr über das Verhalten des Systems machen. Dieser Fall tritt z.B. bei vielen Computerviren ein: Ein vireninfiziertes Programm wird vom Benutzer oder Administrator in die TCB aufgenommen (z.B. durch Starten des Programmes). Dieses Programm ist dann in der Lage, in beliebiger Weise das Verhalten der TCB zu modifizieren. Für Windows existiert beispielsweise ein solches Programm (Backorifice, [Luc98]), das, einmal gestartet, über Netzwerk einem Dritten dauerhaft vollen Zugriff auf alle Ressourcen (Dateien, Registratur, Bildschirminhalt, etc.) bietet.

Auf TCB basierende Sicherheitsmodelle sind also nicht unproblematisch: Die TCB ist oft groß und daher meist nicht verifizierbar [Mcl97]. Dennoch basieren die meisten Implementationen heutiger Sicherheitsmodelle auf einer TCB.

2.3 Zugriffsmatrix-Modell

Viele Sicherheitskonzepte heutiger Systeme verwenden das Zugriffsmatrix-Modell [SiS94]. Es gibt Objekte, das sind die Dinge im System, auf die zugegriffen werden kann (z.B. Dateien oder Drucker), und es gibt Subjekte (z.B. Benutzer, Programmteile), die auf diese Objekte möglicherweise zugreifen können. Die Zugriffsmatrix beschreibt nun, welche Subjekte in welcher Art auf welche Objekte zugreifen können. Mögliche Zugriffsarten in klassischen Systemen sind beispielsweise: Lesen, Schreiben, Löschen, Anhängen, Ausführen.

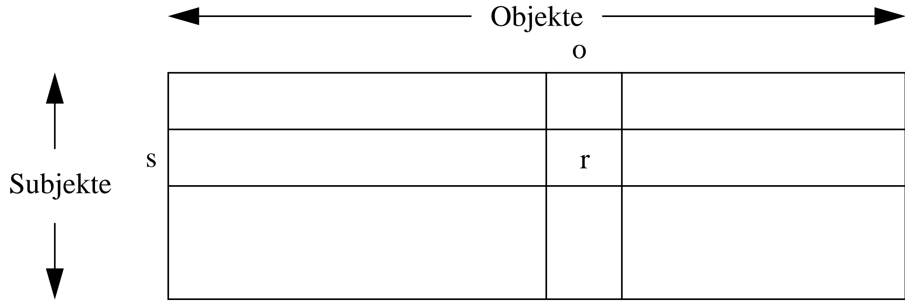

Abbildung 2.1 Zugriffsmatrix

In Abbildung 2.1 kann z.B. Subjekt `s` mit Recht `r` auf Objekt `o` zugreifen. Die Subjekte und Objekte müssen nicht disjunkt sein. Falls auch auf Subjekte zugegriffen werden kann, kann z.B. die Menge der Subjekte Teilmenge der Menge der Objekte sein.

Die Zugriffsmatrix ist in den meisten Systemen nicht statisch, sondern läßt sich zur Laufzeit modifizieren. Wenn ein Benutzer Zugriff auf ein gewisses Objekt hat, kann er beispielsweise (falls ihm das System dies erlaubt) einem anderen Benutzer ebenfalls Zugriff auf das Objekt gestatten. Dazu trägt er in die Matrix das entsprechende Recht ein.

In vielen Systemen möchte man gern Aussagen darüber treffen, ob ein bestimmtes Recht an ein gewisses Subjekt durchsickern kann. Man setzt dabei den Initialzustand und die möglichen Kommandos als bekannt voraus. Die Kommandos bestehen jeweils aus einer Bedingung (etwa: “Hat Subjekt `s` für Objekt `o` das Recht `r`?”) und einer Liste von Kommandos, die bei positivem

Entscheid ausgeführt werden (etwa: "Trage Recht `r` auf Objekt `o` für Subjekt `s` ein"). Man kann nun beweisen, daß das Sicherheitsproblem unentscheidbar ist ([HRU76], [Hof84]), d.h. es ist unentscheidbar, ob es ausgehend von einer Anfangskonfiguration eine Folge von Kommandos gibt, die ein gewisses Zugriffsrecht in die Zugriffsmatrix einträgt.

Dennoch basieren viele heutige Systeme auf dem Zugriffsmatrix-Modell. Da die Zugriffsmatrix im allgemeinen sehr groß und daher unhandlich ist, wird sie meist entweder spaltenweise oder zeilenweise zerlegt; die spaltenweise Zerlegung ergibt Zugriffslisten, die zeilenweise Zerlegung Capabilities.

2.3.1 Zugriffslisten

Zugriffslisten (access control list, acl) teilen die Zugriffsmatrix spaltenweise. Zu jedem Objekt wird gespeichert, welche Subjekte darauf zugreifen können. Es ist daher einfach, festzustellen, welche Subjekte Zugriff auf ein bestimmtes Objekt haben. Revokation von Rechten ist ebenfalls einfach: Um ein ausgesprochenes Zugriffsrecht ungültig zu machen, muß man lediglich in der entsprechenden Zugriffsliste das Subjekt austragen. Delegation von Rechten ist kompliziert, da dazu Kooperation mit der Zugriffsliste des Zielobjektes nötig ist. Um ein Zugriffsrecht an ein anderes Subjekt weiterzugeben, muß man dieses in die Zugriffsliste des Zieles eintragen lassen (welches die Zugriffsliste natürlich nur zulassen sollte, sofern man selbst Zugriff auf das Objekt hat).

Zugriffslisten sind daher besonders für statische Sicherheitsprobleme geeignet, bei denen man Delegation von Rechten nur selten benötigt.

2.3.2 Capabilities

Capabilities [DeH66] teilen die Zugriffsmatrix zeilenweise. Zu jedem Subjekt wird gespeichert, auf welche Objekte es zugreifen kann. Mit dieser Vorgehensweise ist es schwierig, festzustellen, welche Subjekte auf ein gewisses Objekt zugreifen können. Hierzu ist es nötig, bei allen Subjekten in der Capability-Liste nachzusehen. Revokation von Rechten ist – abhängig von der Implementation – ebenfalls schwierig. Falls die Capability-Liste bei dem Subjekt gespeichert ist, ist möglicherweise Kooperation mit dem Subjekt nötig, um ein Recht zu revozieren. Delegation ist bei dieser Vorgehensweise einfach: Um ein Recht weiterzugeben, ist nur Kooperation der Subjekte, zwischen denen das Recht ausgetauscht werden soll, nötig. Problematisch erscheint dabei, daß das Objekt selbst keine Kenntnis von dieser Weitergabe erhält.

Capabilities werden oft ähnlich betrachtet wie Objektreferenzen: Bei der Erzeugung eines Objektes erhält das erzeugende Subjekt automatisch eine Capability, die ihm Zugriff auf das Objekt erlaubt. Einem anderen Subjekt kann nur durch Weitergabe der Capability Zugriff auf das Objekt gewährt werden.

In der objektorientierten Sichtweise können dann die Objekte des Systems gleichzeitig als Subjekte betrachtet werden. Die Capabilities, über die sie verfügen, sind die Referenzen in den Instanzvariablen des Objektes und die lokalen Variablen aller gerade aktiven Methoden des Objektes (das sind die Methoden, in denen sich ein Aktivitätsträger befindet).

2.4 Bell - La Padula Modell

Das Bell-La Padula Modell (Beschreibung in [Mcl85]) baut auf dem Zugriffsmatrix-Modell auf. Es gibt genau festgelegte Zugriffsrechte, die in die Zugriffsmatrix eingetragen werden können:

- **read**
  Dieses Attribut erlaubt das Lesen des Objektes.
- **append**
  Dieses Attribut ermöglicht das Schreiben des Objektes, ohne den Inhalt zu lesen.
- **read-write**
  Dieses Attribut gestattet das Lesen und Schreiben des Objektes.
- **execute**
  Dieses Attribut erlaubt das Ausführen des Objektes ohne den Inhalt zu lesen oder zu schreiben.

Das Bell-La Padula Modell versucht, den Informationsfluß zwischen Objekten zu beschränken. Die Objekte und die Subjekte des Systems haben dazu eine zusätzliche Eigenschaft: das Sicherheitsniveau, das bei Objekten deren Geheimhaltungsstufe angibt und bei Subjekten deren Rechte beinhaltet. Die Subjekte haben zu dem Sicherheitsniveau noch ein momentanes Sicherheitsniveau, auf dem sie aktuell arbeiten, das niedriger als ihr Sicherheitsniveau sein muß oder diesem entsprechen muß. Um den Informationsfluß einzuschränken, werden zwei Systemeigenschaften definiert: die einfache Sicherheitseigenschaft und die *-Eigenschaft.

- Ein System erfüllt die einfache Sicherheitseigenschaft, wenn kein Subjekt im System existiert, das auf Objekte mit höherem Sicherheitsniveau mit den Attributen “read” oder “read-write” zugreifen kann.
- Ein System erfüllt die *-Eigenschaft, wenn Subjekte im System
  - mit dem Attribut “append” nur auf Objekte mit höherem oder gleichem Sicherheitsniveau
  - mit Attribut “read-write” nur auf Objekte mit gleichem Sicherheitsniveau und
  - mit Attribut “read” nur auf Objekte mit niedrigerem oder gleichem Sicherheitsniveau wie ihr aktuelles zugreifen können.

Die *-Eigenschaft verhindert Informationsfluß von Objekten, die ein hohes Sicherheitsniveau besitzen an Objekte und Subjekte mit niedrigerem Sicherheitsniveau. Durch die Einführung des aktuellen Sicherheitsniveaus eines Subjekts kann ein Subjekt in verschiedenen Sicherheitsni-

veaus arbeiten, ohne der *-Eigenschaft widersprechenden Informationsfluß zwischen diesen zu erzeugen. Voraussetzung ist dabei allerdings, daß das Subjekt selbst keine Informationen speichert. Wenn Subjekte mit Personen assoziiert werden, darf die Person beispielsweise keine Informationen, die sie bei Arbeiten mit hohem Sicherheitsniveau erhalten hat, in Arbeit bei niedrigem Sicherheitsniveau einfließen lassen. Da die *-Eigenschaft für viele reale Anwendungen zu restriktiv ist und zumindest an einigen Stellen auch der Datenfluß von Objekten mit hohem an Objekte mit niedrigem Sicherheitsniveau nötig ist, werden sogenannte vertrauenswürdige Subjekte definiert, die gegen die *-Eigenschaft verstoßen dürfen.

Abgesehen von dieser Ausnahme handelt es sich bei der *-Eigenschaft um eine erzwungene Sicherheitsstrategie: Ein Subjekt, das gewisse Informationen lesen kann, hat keine Möglichkeit (egal ob versehentlich oder mit Absicht), diese Daten an Subjekte mit niedrigerem Sicherheitsniveau zu übermitteln.

2.5 Rollenbasierte Zugriffskontrolle

In vielen Organisationen wird als Basis für Berechtigungen das Zugriffsmatrix-Modell verwendet, wobei Personen mit Subjekten der Zugriffsmatrix identifiziert werden. Die Zugriffsrechte jeder einzelnen Person werden gemäß ihren Aufgaben und daher benötigten Ressourcen direkt in die Zugriffsmatrix eingetragen. Wenn Veränderungen an den Berechtigungen vorgenommen werden müssen, beispielsweise wenn eine neue Person in die Organisation aufgenommen wird, muß eine neue Zeile in die Zugriffsmatrix eingetragen werden. Wenn die Zugriffsmatrix in Form von Zugriffslisten verwaltet wird, ist dies nicht einfach: Die Zugriffsliste jedes einzelnen Objektes muß informiert werden. Bei Verwendung der anderen Speicherungsform der Zugriffsmatrix, Capabilities, tritt ein ähnliches Problem beim Hinzufügen neuer Objekte auf.

Rollenbasierte Zugriffskontrolle [FeK92] löst dieses Problem. Die Subjekte der Zugriffsmatrix werden nicht direkt mit Personen assoziiert, sondern es werden Rollen eingeführt. Die Rollen stellen die Subjekte des klassischen Zugriffsmatrix-Modells dar. Jeder Person werden nun Rollen, in denen sie arbeiten darf, zugeordnet. Für eine gewisse Tätigkeit wählt eine Person dann eine oder mehrere von ihren Rollen aus. Diese werden zu den aktiven Rollen der Person; die Person darf dann auf die Objekte zugreifen, die für die Rolle zulässig sind. Rollenbasierte Zugriffskontrolle regelt dabei den Zugriff nicht auf Objektzugriffsbasis, sondern auf Operationsbasis. Für eine bestimmte Rolle sind gewisse Operationen zulässig. Wenn neue Personen in die Organisation aufgenommen werden, muß die Zugriffsmatrix nicht geändert werden; den neuen Personen werden lediglich Rollen zugewiesen.

Zusätzlich erlaubt rollenbasierte Zugriffskontrolle die Begrenzung des Informationsflusses [FCK95]. Es können Rollen als unverträglich definiert werden. Dann kann eine Person diese Rollen nicht gleichzeitig aktivieren. So kann man verhindern, daß zwischen Objekten, die die Person unter verschiedenen Rollen ansprechen kann, Information fließt.

2.6 Kryptographie

Für die Implementation von verteilten Sicherheitsmodellen benötigt man, sofern man nicht allen Komponenten des Systems vertrauen kann, Kryptographie. In diesem Abschnitt wird auf die verschiedenen Arten der Kryptographie und die damit erreichbaren Semantiken eingegangen, Vorteile und Nachteile werden diskutiert. Die mathematischen Grundlagen und die Implementation von Kryptographie-Algorithmen werden hier nicht erläutert, sie sind detailliert in [KPS95] dargestellt; dieses Buch diente auch als Grundlage für diesen Abschnitt.

In der Literatur wird Kryptographie oft als die Disziplin innerhalb der Informatik bezeichnet, die sich mit Verschlüsselungsverfahren zum Schutz geheimer Daten vor unbefugten Zugriffen befaßt [Bro89]. Dies ist auch die ursprüngliche Bedeutung: Das Wort Kryptographie stammt aus dem griechischen und bedeutet: “geheimes schreiben”. Tatsächlich ist dies jedoch nur ein Teil der heutigen Kryptographie. Kryptographie beinhaltet zusätzlich Schutz vor Manipulation von Daten und Mechanismen zur Authentifizierung von Daten.

Als Basis für diese Dienste werden Verschlüsselungsalgorithmen benötigt, die in ihrer Ursprungsform dazu dienen, Daten geheim über einen nicht-abhörsicheren Kanal zu transferieren. Diese Algorithmen basieren darauf, Quelldaten (im weiteren Klartextdaten genannt) in Kanaldaten (im weiteren verschlüsselte Daten genannt) umzurechnen. Für diese Umrechnung (Verschlüsselung der Daten) wird ein Schlüssel benötigt. Die Rückrechnung der verschlüsselten Daten in Klartextdaten erfolgt ebenfalls durch einen Schlüssel (Abbildung 2.2). Wenn es sich dabei um den gleichen Schlüssel handelt (d.h. Schlüssel A=Schlüssel B), nennt man das Verfahren *symmetrisches Verschlüsselungsverfahren*, sonst *asymmetrisches Verschlüsselungsverfahren*.

Abbildung 2.2 Ver- und Entschlüsselung von Daten

Die Grundidee bei der Verschlüsselung zum Schutz vor unbefugten Zugriffen ist, daß man aus den verschlüsselten Daten ohne Kenntnis des Schlüssels keine Rückschlüsse auf die Klartextdaten ziehen kann (Abbildung 2.3). Die verschlüsselten Daten kann man über einen nicht-ver-

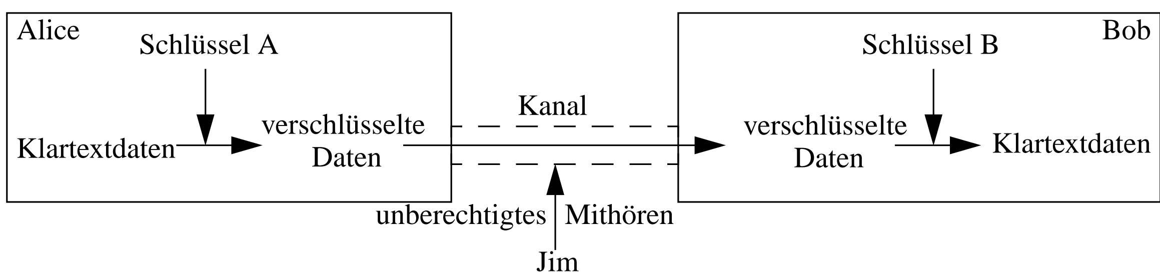

Abbildung 2.3 Verwendung von Verschlüsselung zur Geheimhaltung von Daten

trauenswürdigen Kanal senden, ohne daß jemand, der am Kanal mithört, die über den Kanal gesendeten Daten verstehen kann. Dazu muß man allerdings sicherstellen, daß der Schlüssel vom Sender und Empfänger geheimgehalten wird.

Ein generelles Problem bei der Verschlüsselung ist die Wahl der Schlüssel: Die Schlüssel muß man so wählen, daß jemand, der am Kanal mithört, diese nicht erraten kann. Wenn man beispielsweise eine Zufallszahl als Schlüssel verwendet, muß man besondere Anforderungen an den Zufallszahlengenerator stellen. Viele Zufallszahlengeneratoren nehmen die aktuelle Uhrzeit als Basis und erzeugen dann Pseudozufallszahlen durch Bitpermutation. Diese Generatoren sind für die Schlüsselerzeugung ungeeignet, da ein potentieller Angreifer den Zeitpunkt der Schlüsselerzeugung kennen könnte und damit dann den Schlüssel ebenfalls kennt. Statt dessen kann man nicht-ratbare Ereignisse, wie beispielsweise der zeitliche Abstand von Tastendrücken eines Benutzers in sehr hoher Auflösung (Mikrosekunden) nehmen.

Bei Verschlüsselungsalgorithmen treten eine Reihe von weiteren Problemen auf:

- **Sicherheit**
  Wenn jemand nur die verschlüsselten Daten kennt, sollte er daraus nicht (oder nur mit sehr hohem Aufwand) die Klartextdaten ermitteln können. Dieses Kriterium sollte jeder Verschlüsselungsalgorithmus erfüllen.

- **Folgeschluß**
  Wenn jemand einen Teil der Klartextdaten kennt und den Kanal abhört, so daß er alle verschlüsselten Daten kennt, kann er möglicherweise den Schlüssel ermitteln oder Rückschlüsse auf den Rest der Klartextdaten ziehen. Dies sollte ein Verschlüsselungsalgorithmus verhindern oder zumindest erschweren. Wenn ein Angreifer eine sehr große Menge von Klartextdaten und zugehörigen verschlüsselten Daten besitzt, kann er beispielsweise durch Ausprobieren aller Schlüsselmöglichkeiten den passenden Schlüssel ermitteln und damit andere Daten dekodieren. Dies sollte nicht effizient durchführbar sein, die Menge von Schlüsseln, die durchprobiert werden müßte, sollte also groß sein.

- **Rückschluß von der Menge der übersandten Daten auf die Menge der Klartextdaten**
  Auch durch Kommunikation mittels verschlüsselter Daten verrät man einem am Kanal Lauschenden, mit welchen Kommunikationspartnern man kommuniziert und in welcher Größenordnung man mit welchem Kommunikationspartner Daten austauscht. Dieses Problem wird von den meisten Algorithmen nicht berücksichtigt, es gibt aber Lösungen dafür [Tim97].

Zunächst soll hier die symmetrische Verschlüsselung betrachtet werden. Da zum Ver- und Entschlüsseln der gleiche Schlüssel verwendet wird, muß dieser sowohl dem Sender als auch dem Empfänger bekannt sein. Dazu muß der Schlüssel zwischen beiden übertragen werden. Dies muß so geschehen, daß kein Dritter diesen erhält. Mit symmetrischer Verschlüsselung ist es möglich, zwischen zwei Teilnehmern, die den gleichen Schlüssel besitzen, in beide Richtungen vertraulich Daten auszutauschen. Es existieren sehr effiziente symmetrische Verschlüsselungsalgorithmen. Daher finden diese besonders bei Verschlüsselung von großen Datenmengen Anwendung. Der Nachteil von symmetrischer Verschlüsselung ist die Schlüsselverwaltung und

Verteilung bei Systemen mit vielen potentiellen Kommunikationspartnern: Wenn man sicherstellen möchte, daß die Kommunikation zwischen zwei Partnern jeweils nur von diesen beiden mitverfolgt werden kann, benötigt man für jede Kommunikationskonstellation einen separaten geheimen Schlüssel. Wenn also bei `n` Kommunikationspartnern potentiell jeder mit jedem geheim kommunizieren möchte, muß jeder Teilnehmer `n-1` Schlüssel besitzen, insgesamt existieren dann `n*(n-1)/2` verschiedene Schlüssel im System. Bei großen Systemen oder dynamischen Systemen ist solch eine Lösung nicht praktikabel. Bekannte symmetrische Verschlüsselungsalgorithmen sind: DES, triple DES und IDEA.

Bei asymmetrischer Verschlüsselung wird zur Verschlüsselung der Daten ein anderer Schlüssel als zur Entschlüsselung verwendet. Besonders interessant sind hierbei die Verfahren, die mit öffentlichem Schlüssel arbeiten. Bei diesen Verfahren läßt sich aus dem Schlüssel zur Verschlüsselung (öffentlicher Schlüssel) nur mit sehr hohem Aufwand der Schlüssel zur Entschlüsselung (privater Schlüssel) berechnen. Man kann daher den öffentlichen Schlüssel allen Teilnehmern zugänglich machen. Jeder ist dann in der Lage, Daten zu verschlüsseln, jedoch nur derjenige, der den zugehörigen privaten Schlüssel kennt, kann die Daten entschlüsseln. Bei `n` Kommunikationspartnern benötigt dann jeder nur einen privaten Schlüssel zum Entschlüsseln und `n-1` öffentliche Schlüssel zum Verschlüsseln. Es gibt dann insgesamt `n` private und die dazu passenden `n` öffentlichen Schlüssel im System. Asymmetrische Verfahren sind relativ ineffizient, daher werden sie meist nur zur Verschlüsselung kleiner Datenmengen eingesetzt. Der bekannteste asymmetrische Verschlüsselungsalgorithmus ist RSA.

Oft werden beide Arten von Verfahren kombiniert eingesetzt. Um Daten mit einem Teilnehmer auszutauschen wird mithilfe eines asymmetrischen Verfahrens Kontakt aufgenommen und ein in dem Moment generierter symmetrischer, sogenannter Sitzungsschlüssel ausgetauscht. Mit Hilfe dieses Sitzungsschlüssels wird dann kommuniziert. Auf diese Weise kann man die Vorteile der asymmetrischen Verschlüsselung (einfache Schlüsselverwaltung) und der symmetrischen Verschlüsselung (hohe Effizienz) vereinigen.

Eine weitere Möglichkeit, asymmetrische Verschlüsselung einzusetzen, ist das Signieren von Daten zur Authentifizierung. Ein Teilnehmer besitzt dazu einen privaten Schlüssel, alle anderen kennen den zugehörigen öffentlichen Schlüssel. Er erzeugt nun über die zu authentifizierenden Daten einen sicheren Hash, das ist eine Prüfsumme über die Daten, die nicht reversibel ist. Das bedeutet, daß sich zu beliebigen Daten zwar eine Prüfsumme mit geringem Aufwand berechnen läßt, sich jedoch zu einer Prüfsumme nur mit sehr großem Aufwand passende Daten berechnen lassen. MD5 ist beispielsweise eine solche sichere Hash-Funktion. Diesen sicheren Hash verschlüsselt der Teilnehmer mit seinem privaten Schlüssel und übermittelt dann das Verschlüsselungsergebnis und die Daten an andere Teilnehmer. Diese können nun verifizieren, daß die Quelle der Daten den privaten Schlüssel kennt: Sie können das Verschlüsselungsergebnis mit dem öffentlichen Schlüssel des Teilnehmers entschlüsseln und den sicheren Hash von den Daten selbst bilden. Wenn beide Ergebnisse übereinstimmen, sind die Empfänger der Daten sicher, daß die Quelle der Daten Kenntnis des privaten Schlüssels hat. Wenn nun beispielsweise ein privater Schlüssel nur einem bestimmten Benutzer bekannt ist, kann dieser Benutzer mit dem Schlüssel nachweisen, daß die übermittelten Informationen von ihm stammen.

Bei der Verwendung von asymmetrischer Verschlüsselung erscheint die Schlüsselverteilung zunächst unproblematisch. Nur die öffentlichen Schlüssel müssen verteilt werden, daher treten dabei keine Probleme der Geheimhaltung auf; die Schlüsselverteilung kann unverschlüsselt erfolgen. Ein nicht generell lösbares Problem ist jedoch die Authentifizierung der Schlüssel. Ein Teilnehmer, der neu in ein System eintritt und keinen öffentlichen Schlüssel anderer Teilnehmer kennt, kann keinen sicheren Kontakt zu einem anderen Teilnehmer herstellen. Er kann sich zwar den dazu nötigen öffentlichen Schlüssel geben lassen, er kann jedoch nicht prüfen, ob der Schlüssel, den er erhält, wirklich von diesem Teilnehmer stammt, ob also der Schlüssel wirklich korrekt ist.

Zur Lösung dieses Problems gibt es verschiedene Ansätze.

X509 [Sum97] verwendet den zentralen Ansatz. Es gibt einige wenige, zentrale, vertrauenswürdige Zertifizierungsstellen. Die öffentlichen Schlüssel der Zertifizierungsstellen sind allen Teilnehmern bekannt. Bei diesen Zertifizierungsstellen kann ein Teilnehmer seinen öffentlichen Schlüssel zusammen mit weiteren Informationen über sich signieren lassen. Die Zertifizierungsstelle prüft nach gewissen Kriterien (die abhängig von der Güteklasse des Schlüssels sind) die Informationen, bevor sie sie mit ihrem privaten Schlüssel signiert. Wenn man nun solch einen öffentlichen Schlüssel mit den zugehörigen Informationen und der Signatur bekommt, kann man überprüfen, ob er tatsächlich von einer der Zertifizierungsstellen signiert wurde. In diesem Fall akzeptiert man den Schlüssel.

PGP [Gar95] verwendet den verteilten Ansatz. Jeder Teilnehmer kann öffentliche Schlüssel anderer Teilnehmer signieren und weitergeben. Jeder muß dann selbst entscheiden, welchen anderen Teilnehmern und welchen Schlüsseln er traut.

Eine komplett andere Lösung verwendet Kerberos (Kerberos V4) [KoN93]. Hier hat jeder Teilnehmer einen geheimen Schlüssel, und es gibt eine zentrale Schlüsselverwaltungsstelle, die alle geheimen Schlüssel kennt. Sie kann also mit jedem Teilnehmer verschlüsselt kommunizieren. Wenn nun zwei Teilnehmer kommunizieren möchten, erzeugt sie einen neuen geheimen Schlüssel und übermittelt diesen verschlüsselt an beide Partner. Obwohl nur symmetrische Verschlüsselung verwendet wird, sind relativ wenig Schlüssel nötig. Ein Nachteil ist aber die zentrale Koordinierungsinstanz.

Bei allen Lösungen muß man initial gewisse Schlüssel kennen oder bereits verteilt haben. Lediglich die Art, wie weitere Vertrauensbeziehungen bzw. die weitere Schlüsselverteilung vorgenommen wird, ist verschieden. Während Kerberos zur Kontaktaufnahme von Teilnehmern jeweils mit einer zentralen Komponente Verbindung aufnehmen muß (und daher schlecht skaliert), muß dies bei X509 nur zur Erzeugung von neuen Schlüsseln geschehen (z.B. wenn neue Teilnehmer erstmalig in das System kommen). Bei hochgradig dynamischen Systemen oder sehr großen Systemen entstehen jedoch auch bei dieser Lösung Engpässe. Ein dezentrales System wie bei PGP skaliert am besten, da es überhaupt keine zentralen Komponenten benötigt. Die Vertrauensbeziehungen müssen aber durch jeden Teilnehmer einzeln festgelegt werden, was oft nicht einfach ist.

3 Sicherheit in verteilten Systemen

In diesem Kapitel werden Systeme untersucht, die spezielle Mechanismen für Sicherheit bieten. Dabei wird besonders auf die angebotenen Sicherheitsmechanismen und -semantiken eingegangen. Neben Experimentalsystemen werden insbesondere auch Systeme untersucht, die tatsächlich im Einsatz sind. Die von solchen Systemen zur Verfügung gestellten Mechanismen können als Anhaltspunkt für die Überlegung verwendet werden, welche Sicherheitsmechanismen für Anwendungen nötig sind.

3.1 Betriebssysteme

3.1.1 Hydra

Hydra [WCC+74],[Lev84] ist eins der ältesten Betriebssysteme, das eine Vielzahl von Sicherheitsproblemen behandelt. Das von Hydra verwendete Sicherheitsmodell basiert auf Capabilities. Es gibt Objekte im System, die aus einem Datenbereich und Capabilities bestehen. Ein Objekt kann nur angesprochen werden, wenn man eine entsprechende Capability besitzt. Eine solche Capability besteht aus einer Referenz auf das Zielobjekt und einem Bitfeld, das erlaubte Operationen, die über die Capability ausgeführt werden dürfen, festlegt.

Die festlegbaren Operationen sind:

- Lesen, Schreiben der Daten, Anhängen an die Daten
- die Capability-Liste des Objektes lesen, schreiben, anfügen, löschen
- Zielobjekt kopieren, löschen
- Capability löschen, Rechte reduzieren
- benutzerdefinierbare Rechte

In Hydra gibt es auch Capabilities für Funktionen. Wenn man in Hydra eine Funktion über eine solche Capability aufruft, kann man an diese als Parameter weitere Capabilities übergeben. Falls man der Funktion nicht vollständig vertraut, kann man beim Aufruf die übergebenen Capabilities beschränken. Beispielsweise kann man das Schreibrecht vor der Übergabe entfernen. Zusätzlich können weitere Rechte festgelegt werden:

- Modifikation des Zielobjektes erlaubt bzw. nicht erlaubt
- Weitergabe, Speichern der Capability erlaubt bzw. nicht erlaubt

Beim Funktionsaufruf kann zusätzlich festgelegt werden, ob überhaupt die Modifikation von Objekten erlaubt ist. Wenn dieses Recht entzogen wird, kann der Aufruf keine Daten modifizieren. Damit kann man verhindern, daß ein Aufruf z.B. Informationen über die Aufrufparameter in einem Objekt ablegen und diese einem anderen Aufrufer später zugänglich machen kann.

Um objektbasierte Programmierung zuzulassen, erlaubt Hydra Rechteverstärkung. Wenn man eine Referenz auf ein Objekt besitzt, sollte, um die Objektkapselung zu wahren, kein direkter Zugriff auf die Objektdaten zulässig sein. Die Rechte zum Lesen und Schreiben der Objektdaten sollten also in der Capability gelöscht sein. Man ruft nun eine Funktion auf (die die Rolle einer Methode des Objektes übernimmt) und übergibt ihr eine Referenz (Capability) auf dieses Objekt. Die Funktion muß in der Lage sein, die Objektdaten zu modifizieren, d.h. die Capability muß das Recht zum Lesen und Schreiben der Daten besitzen. Dazu kann die Funktion eine Rechteverstärkungsvorlage enthalten, die die übergebene Parametercapability mit zusätzlichen Rechten versieht. Mit diesen Verstärkungsvorlagen muß natürlich vorsichtig umgegangen werden, da sie eine vorgenommene Rechteeinschränkung außer Kraft setzen. Über Verstärkungsvorlagen sollten daher nur Funktionen aus dem Modul des Objektes (bzw. im objektorientierten Sinne aus der Klasse des Objektes) verfügen.

Capabilities sind bei Hydra im Betriebssystem implementiert: Die Capability selbst befindet sich im geschützten Kernadreßraum. Das Benutzerprogramm muß Betriebssystemfunktionen aufrufen, um mit einer Capability zu arbeiten und gibt dabei einen Index in die Capabilitytabelle an. So wird sichergestellt, daß Programme nicht unkontrolliert Capabilities modifizieren können.

Die Mechanismen von Hydra erlauben die Lösung spezieller Sicherheitsprobleme:

- **Gegenseitiges Mißtrauen**
  Beim Aufruf einer Funktion muß weder der Aufrufer der Funktion trauen, noch umgekehrt. Es kann genau konfiguriert werden, in welcher Weise eine Funktion auf welche Capabilities zugreifen kann. Diese Konfiguration kann allerdings durch Rechteverstärkungsvorlagen umgangen werden.
- **Konstanten-Problem**
  Bei einer Capability kann festgelegt werden, daß über sie keine Modifikationen vorgenommen werden können.

- **Weitergabe-Problem**
  Bei einer Capability kann festgelegt werden, ob sie weitergegeben werden darf. Dadurch kann die Benutzung von Capabilities auf die Programmteile beschränkt werden, denen man sie direkt übergeben hat.
- **Speicherungs-Problem**
  Eine Capability besitzt ein Attribut, das festlegt, ob sie gespeichert werden kann. Durch Verwendung des Attributes ist man bei der Übergabe der Capability an eine Funktion sicher, daß die aufgerufene Funktion diese Capability nicht speichert. Man kann damit garantieren, daß die Funktion nach Beendigung des Aufrufs keine Möglichkeit mehr hat, weiterhin auf die übergebenen Capabilities zuzugreifen.
- **Einschluß (Confinement)** (siehe auch [Lam73])
  Bei einem Funktionsaufruf kann festgelegt werden, daß nur in einigen, genau spezifizierten Objekten Informationen abgelegt werden dürfen. Der Aufrufer hat dadurch die Möglichkeit, die Informationen, die er an die Funktion übergibt, einzuschließen. Sie können dann keinesfalls nach außen dringen.
- **Initialisierungsproblem**
  Wenn man ein neues Objekt generiert, kann man festlegen, daß man nach der Initialisierung als einziger eine Referenz auf dieses besitzt.
- **Revokation**
  Durch Indirektion ist es in Hydra möglich, Capabilities nachträglich ungültig zu machen.

Hydra bietet viele Konzepte, die auch für heutige objektorientierte Systeme und verteilte Systeme interessant sind. Obwohl Hydra auf Capabilities basiert, hat der Benutzer Kontrolle über Weitergabe und Nutzung der Capabilities, so daß die Nachteile der Capabilities teilweise ausgeglichen werden.

Die Capabilities werden bei Hydra nicht feingranular verwendet. Da für alle Aktionen mit Capabilities Kern-Interaktion nötig ist, wäre dies zu ineffizient. Außerdem muß der Benutzer selbst für jede seiner Capabilities festlegen, mit welchen Einschränkungen er sie versehen möchte – Sicherheitsstrategie und Applikationscode werden gemischt. Hydra besitzt auch kein verteiltes Sicherheitsmodell, sondern implementiert nur lokale Sicherheit.

3.1.2 Amoeba

Amoeba [TMR86] ist ein verteiltes, objektorientiertes Betriebssystem, das auf Capabilities basiert. Eine Amoeba-Capability ist eine Objektreferenz auf ein gewisses Serverobjekt. Interaktion zwischen Amoeba-Prozessen erfolgt durch Fernaufrufe an solchen Capabilities.

Ein Server-Prozeß kann Objekte anbieten, indem er Capabilities für diese erstellt. Dazu belegt er für jedes Objekt einen Port und eine Objektnummer, unter der er das Objekt zur Verfügung stellt. Zusätzlich erzeugt er zu jedem angebotenen Objekt eine Zufallszahl und speichert diese.

Die Zufallszahl dient als Schutz vor unbefugtem Zugriff. Wenn ein Klient auf ein Objekt eines Servers zugreifen möchte, benötigt er eine Capability für dieses Objekt, die aus Port, Objektnummer sowie der Zufallszahl zu dem Objekt besteht.

Im Gegensatz zu anderen Betriebssystemen wie Hydra muß die Capability nicht gegen böswillige Manipulation geschützt werden. Der Klient kann zwar eine beliebige Capability mit Portnummer und Objektnummer erzeugen, aber er kennt die zu dem Objekt passende Zufallszahl nicht und kann daher nicht auf das Objekt zugreifen. Capabilities sind in Amoeba somit einfache Zahlen, die ohne spezielle Maßnahmen des Betriebssystems gespeichert und weitergegeben werden können – auch an Klienten, die auf anderen Rechnern laufen.

Restriktion von Capabilities, also die Einschränkung der aktivierbaren Operationen einer Capability, ist auf zwei Arten möglich: Einerseits kann Kontakt mit dem Server aufgenommen werden. Dieser erzeugt dann eine neue Capability, die nur über eingeschränkte Rechte verfügt. Andererseits kann aber auch die Capability ohne Interaktion mit dem Server beschränkt werden. Die Capability verfügt über ein Feld für Zugriffsrechte, in das die eingeschränkten Rechte eingetragen werden können. Um zu verhindern, daß das Zugriffsrechte-Feld von einem Klienten einfach wieder auf volle Rechte gesetzt wird, wird die Zufallszahl mit einer speziellen, mit den Rechten parametrierten Einweg-Funktion so modifiziert, daß diese die niedrigeren Rechte widerspiegelt. Ein Rückrechnen der ursprünglichen Zufallszahl ist aufgrund der Einweg-Funktion nur mit extrem hohem Rechenaufwand möglich. Revokation und Expiration von Capabilities sind in Amoeba nicht möglich.

In Amoeba werden die Capabilities für viele Betriebssystemdienste wie Dateisysteme, Speicherverwaltung und Festplattenverwaltung eingesetzt. Im Gegensatz zu anderen Betriebssystemen werden für diese Dienste reine Capabilities (keine Zugriffslisten) verwendet.

Besonders interessant ist bei Amoeba die Realisierung des verteilten Sicherheitsmodells. Capabilities können dezentral erzeugt werden und auch von Klienten mit Einschränkungen versehen werden. Allerdings erscheint es problematisch, das Sicherheitsmodell auf Capabilities basieren zu lassen, ohne Revokation und Expiration zu ermöglichen. Trennung von Sicherheitsstrategie und Applikationssemantik wird von Amoeba nicht unterstützt, die Sicherheitsstrategie muß von den Applikationen selbst implementiert werden.

3.1.3 Spring

Spring [MGH+94] ist ein objektorientiertes Betriebssystem, das auf Capabilities basiert. Eine Spring-Capability ist eine transparente Objektreferenz (Fat Pointer): Über eine solche Referenz können Methoden des Zielobjektes aufgerufen werden, unabhängig davon, wo sich das Zielobjekt befindet. Dabei können als Parameter der Methode sowohl Wertobjekte, als auch Objektreferenzen (d.h. Capabilities) übergeben werden. Die Capabilities werden im Betriebssystem verwaltet; sie sind daher gegen böswillige Benutzermanipulation geschützt.

Zusätzlich zu Capabilities können auch Zugriffslisten verwendet werden. An der Zielseite einer Capability können dazu Zugriffslisten eingesetzt werden, die, abhängig vom Aufrufer, nur gewisse Methodenaufrufe zulassen. Die Zugriffslisten kommen insbesondere bei Betriebssystemdiensten wie dem Dateisystem zum Einsatz.

Dabei tritt das Delegationsproblem auf: Ein Benutzer, der beispielsweise eine seiner Dateien drucken möchte, besorgt sich zunächst über das Betriebssystem eine Referenz auf diese Datei, die mit einer Zugriffsliste geschützt ist und nur ihm Zugriff gestattet. Wenn er nun diese Referenz an einen Druckserver übergibt, kann dieser nicht auf die Datei zugreifen, es sei denn, er hat (wie beispielsweise in Unix) besondere Rechte, die ihm uneingeschränkten Zugriff auf alle Dateien des Benutzers zusichern. Da diese Vorgehensweise dem Prinzip der kleinstmöglichen Rechte widerspricht, wurde in Spring dieses Problem anders gelöst: Der Benutzer kann aus der mit einer Zugriffsliste versehenen Capability eine (möglicherweise eingeschränkte) reine Capability erzeugen. Wenn er diese an den Druckserver übergibt, kann der Druckserver nur genau auf diese eine Datei zugreifen.

Obwohl Spring Capabilities wie bei Hydra im Kernadreßraum verwaltet, ist die Erstellung und Modifikation der Capabilities hocheffizient. Durch Ausnutzung spezieller Prozessoreigenschaften der Zielplattform ist sogar die Benutzung, d.h. der Methodenaufruf über eine solche Referenz an einem Objekt, das sich in einem anderen Adreßraum befindet, extrem effizient. Die Capabilities können daher feingranular eingesetzt werden. Bei der Betrachtung der Spring-Implementation selbst ([Sun95a]) zeigte sich allerdings, daß dies nicht geschah: Obwohl das Spring-Betriebssystem ebenfalls mit der Capability-Technik arbeitet, wurde bei der Implementation nur grobgranulare Objektorientierung verwendet. Statt Objektreferenzen auf Einzelobjekte herauszugeben, liefert der Spring-Kern oft nur eine Referenz auf ein Verwaltungsobjekt, das dann eine große Schnittstelle besitzt und Methodenaufrufe an Einzelobjekte weiterleitet. Ein Grund hierfür ist die Zentralisierung der Sicherheitsstrategie: Um die Konfiguration der Sicherheitsstrategie an wenigen zentralen Stellen durchführen zu können, werden die Interfaces zwischen verschiedenen Komponenten so weit wie möglich reduziert. Es gibt nur wenige solche Schnittstellen, die aber viele Methoden – die zum Teil eigentlich anderen Schnittstellen zugehören müßten – enthalten, obwohl dies der Objektorientierung widerspricht.

3.2 Verteilte Ablaufsysteme

3.2.1 Corba

Corba (Common Object Request Broker Architecture, [Red96], [OHE96]) ist eine Architektur für verteilte Objektinteraktion. Corba beinhaltet eine spezielle Sprache (IDL, Interface Definition Language) zur Beschreibung der Schnittstellen von Objekten. Die Implementierung dieser Objekte und die Verwendung der Objekte über die Schnittstellen kann in einer nahezu beliebigen Programmiersprache erfolgen: Es gibt für alle gängigen Programmiersprachen (C, C++, Java, Smalltalk, Cobol, Ada, Perl) Umsetzungen der IDL-Schnittstellen auf Schnittstellen der Programmiersprache. Wenn ein Programm eine Corba-Objektreferenz besitzt, kann es also nicht

nur vom Ort des Objektes, sondern auch von der Programmiersprache, in der das Objekt realisiert ist, abstrahieren. Corba spezifiziert neben der Objektinteraktion mit Methodenaufrufen auch verschiedene Dienste wie Namensdienst, Koordinierungsdienst und Sicherheitsdienst.

Die Spezifikation des Sicherheitsdienstes ist sehr umfangreich [Omg97]. Da aufgrund des hohen Implementationsaufwands nicht alle Corba-Implementierungen den kompletten Sicherheitsdienst realisieren können, wird der Sicherheitsdienst in zwei Ebenen unterteilt: Die Realisierung des Ebene 1 Sicherheitsdienstes umfaßt Sicherheitsfunktionen und Strategien, die auf Systemebene konfiguriert werden, Ebene 2 erlaubt den Applikationen, die Sicherheitsfunktionen und Strategien auch selbst zu realisieren. Ebene 1 wird beispielsweise von Orbix (Iona Technologies) realisiert, Ebene 1 und 2 wird vom VisiBroker (Visigenic) und IntraVerse (eine Erweiterung des IONA-Orbix der Firma DASCOM) implementiert.

In Corba gibt es Identitäten, die als Basis für Sicherheitsstrategien dienen. Diese umfassen mehrere, möglicherweise verschiedene Identifikationen, die für verschiedene Dienste verwendet werden. Beispielsweise kann eine Identität aus einer Identifikation für Zugriffserlaubnisprüfung, einer weiteren Identifikation für Mitprotokollierung der Ereignisse (auditing) und einer dritten Identifikation für die Abrechnung von Operationen (accounting) bestehen. Die Identifikationen können auch Rollen oder Gruppen umfassen, denen die Identität angehört.

Corba unterstützt frei konfigurierbare Sicherheitsstrategien (z.B. durch Zugriffslisten und Capabilities) auf Objektebene, Threadebene und Domänenebene. Die Strategien auf Objektebene müssen von der Applikation selbst realisiert werden. Der Klient, der einen Methodenaufruf durchführen möchte, kann festlegen, welche seiner Privilegien er für den Aufruf verwenden möchte und ob er die Delegation seiner Rechte an das Zielobjekt wünscht – in diesem Fall kann das Zielobjekt unter seiner Identität arbeiten. Dies kann er beim Aufruf selbst konfigurieren oder für den aktuellen Thread festlegen. Die Delegation von Rechten läßt sich in Corba feingranular konfigurieren. Man kann festlegen, daß man keine Rechte delegieren möchte, man kann eingeschränkte Rechte delegieren, die nur für ein bestimmtes Objekt gelten, oder seine Rechte komplett an das Zielobjekt delegieren. Bei verketteten Aufrufen kann man auch festlegen, daß alle Identitäten aller Aufrufer gesammelt werden, so daß als Aufruferidentität die Summe der Identitätenspur des Aufrufs verwendet wird.

Das Zielobjekt kann nun aufgrund der vom Klienten bereitgestellten Informationen Sicherheitsüberprüfungen durchführen. Es muß dazu bei jedem erfolgten Aufruf ein separates Sicherheitsstrategieobjekt aktivieren, das die entsprechende Strategie realisiert. Die implementierte Strategie kann als Entscheidungsparameter den Aufrufer, den Zustand der Applikation und die aufzurufende Zielmethode verwenden und so insbesondere Zugriffslisten und Capabilities realisieren.

Sicherheitsstrategien lassen sich in Corba auch unabhängig von der Applikation festlegen. Dazu definiert Corba Sicherheitsstrategiedomänen. Eine Sicherheitsstrategiedomäne besteht aus einer Menge von Objekten. Die Domänen können sich überschneiden und auch hierarchisch verschachtelt sein. Für jede solche Domäne kann der Systemadministrator Sicherheitsstrategien

festlegen, die beispielsweise die Delegation von Rechten, die Erlaubnis für Aufrufe an und von anderen Domänen, die Mitprotokollierung von Aufrufen und die Verschlüsselung von Aufrufinformationen umfassen kann.

Unabhängig von den Sicherheitsstrategiedomänen gibt es in Corba Sicherheitsumgebungsdomänen. Objekte in einer solchen Domäne haben eine spezielle Beziehung zueinander, die es erlaubt, einen Teil der Sicherheitsstrategie bei diesen Objekten zu ignorieren. Wenn mehrere Objekte sich auf demselben Rechner befinden, kann man diese in einer Sicherheitsumgebungsdomäne unterbringen, die für Kommunikation zwischen diesen Objekten die Datenverschlüsselung außer Kraft setzt, da Datenverschlüsselung lokal nicht notwendig ist. Wenn mehrere Objekte mit denselben Rechten ausgestattet sind und die Objekte einander vertrauen, kann man diese in einer Sicherheitsumgebungsdomäne unterbringen, die keinen Schutz zwischen den Objekten realisiert. Bei C++-Corbaobjekten kann beispielsweise ohne (relativ aufwendige) Trennung durch virtuelle Speicherverwaltung kein Schutz zwischen verschiedenen Objekten realisiert werden. Wenn man zwei C++-Corbaobjekte in einer solchen Sicherheitsumgebungsdomäne unterbringt, kann man erreichen, daß trotz fehlendem Schutz die Objekte in eine virtuelle Speicherverwaltungsdomäne gepackt werden und damit das System effizienter wird.

Zusätzlich gibt es noch Sicherheitstechnikdomänen, die Objekte umfassen, die mit denselben Algorithmen und Protokollen (Verschlüsselung, Authentifizierung) arbeiten.

Das Konzept von Corba ist stark domänenorientiert. Die von Corba favorisierte Eigenschaft der Trennung von Sicherheitsstrategie und Applikationsrealisierung läßt sich nur bei Definition der Sicherheitsstrategie auf Domänenebene realisieren. Hier lassen sich jedoch keine feingranularen Strategien mehr definieren. Bei feingranularer Interaktion zwischen verschiedenen Domänen können viele Objektreferenzen zwischen diesen Domänen etabliert werden. Auf Domänenebene sind diese nur mit großem Aufwand unterscheidbar: Es ist schwer, für verschiedene Quellreferenzen oder Zielobjekte verschiedene Strategien zu implementieren. Bei grobgranularer Interaktion, d.h. wenn sich nur wenige Objekte in jeder Domäne befinden, die dann alle als gleichwertig angesehen werden, ist dieses Konzept hingegen gut geeignet. Die Sicherheitsstrategie läßt sich dann an einer Stelle global und unabhängig von der Applikation definieren, und es lassen sich sogar Applikationen, die ohne Berücksichtigung von Sicherheitsanforderungen implementiert wurden, nachträglich mit einer Sicherheitsstrategie ausstatten.

3.2.2 DCOM

Für Interaktion zwischen Komponenten in Windows NT/95/98 ([Ive96], [Cus93]) hat Microsoft das sogenannte Component Object Model (COM) definiert. Dieses erlaubt Objekten transparente Kommunikation, unabhängig davon, ob die Objekte sich im selben Adreßraum (dann ist die Interaktion hocheffizient) oder in verschiedenen Adreßräumen befinden (dann wird der Aufruf über Stellvertreterobjekte vom Betriebssystem abgehandelt). Um auch transparente Kommunikation zwischen Objekten auf verschiedenen Rechnern zu ermöglichen, wurde COM um eine Verteilungskomponente auf DCOM (distributed COM, [Mic96]) erweitert. Damit können Objekte auch kommunizieren, wenn sie sich auf verschiedenen Rechnern befinden. In DCOM wurde versucht, die verschiedenen Sichtweisen auf die Objekte so orthogonal wie möglich zu hal-

ten: Der Programmierer sieht nur die Objekte mit ihren Schnittstellen. Für ihn ist transparent, auf welchem Rechner und in welchem Adreßraum sich die Objekte befinden. Der Administrator, der eine Applikation starten möchte, kann nun wählen, auf welchem Rechner die Applikation laufen soll und auf welchen Rechnern gewisse Objekte instantiiert werden sollen. Damit ist es sogar möglich, Applikationen zu verteilen, die geschrieben wurden, bevor es DCOM gab: Die Applikation möchte ein COM-Objekt instantiieren; das Laufzeitsystem entscheidet daraufhin anhand der Konfiguration des Administrators, das Objekt auf einem anderen Rechner zu erzeugen. Die Applikation erhält dann eine rechnerübergreifende Referenz auf dieses Objekt.

Während bei Windows 95/98 lokal keine Sicherheitsüberlegungen nötig sind (es gibt in Windows 95/98 lokal keinen Schutz), sind diese im verteilten Fall und bei Windows NT wichtig. DCOM verwendet dazu Zugriffslisten, mit denen man Objekte schützen kann [Mic98]. Die Zugriffslisten können einerseits direkt im Programm angegeben werden oder unabhängig vom Programm in der Windows-Registratur konfiguriert werden. Dabei kann einzeln Schutz für jede aufrufbare Methode festgelegt werden.

Für die Realisierung von Zugriffslisten sind Identitäten nötig, mit denen Aufrufe authentifiziert werden können. DCOM verwendet dazu die bereits im Windows verfügbare Benutzerkennung. Ein Aufrufer kann bei jedem Aufruf wählen, ob zur Authentifizierung seine Benutzerkennung verwendet wird, oder ob der Aufruf anonym durchgeführt wird. Zusätzlich kann er bestimmen, ob und wie weit er seine Rechte delegieren möchte, d.h. ob das Zielobjekt des Aufrufs unter seiner Kennung arbeiten darf und ob das Zielobjekt auch Aufrufe an andere COM-Objekte unter seiner Kennung durchführen darf. Dadurch sind auch Aufrufe an nicht-vertrauenswürdige Objekte möglich, die zwar die Benutzerkennung zur Authentifizierung übermittelt bekommen, diese aber nicht selbst verwenden dürfen.

Bei jedem Aufruf und jedem Zielobjekt kann festgelegt werden, ob die Aufrufdaten verschlüsselt werden sollen und ob die Datenintegrität sichergestellt werden soll. Für alle Einstellungen können in der Windows-Registratur Vorgabewerte festgelegt werden. Außerdem können dort globale Einstellungen vorgenommen werden wie beispielsweise die Festlegung, wer auf welchem Rechner welche Objekte erzeugen darf.

DCOM besitzt seine Stärken hauptsächlich bei grobgranularer Objektorientierung. Eine Anwendung stellt wenige Objekte zur Verfügung, für die gemeinsam in der Registratur über die Klassen der Objekte eine Sicherheitsstrategie angegeben wird (Zugriffsliste). Bei feingranularer Objektorientierung kann man die Sicherheitsstrategie nicht mehr separat in der Registratur festlegen, da man in dem Fall meist verschiedene Sicherheitsstrategien für die verschiedenen Objekte benötigt. Man müßte die Strategie daher in die Anwendungsobjekte hineinprogrammieren, was der Separation von Sicherheitsstrategie und Applikationssemantik widerspricht.

3.2.3 DCE

DCE (Distributed Computing Environment, [Sin97]) ist ein von der Open Group festgelegter Standard für Interaktion zwischen verteilten Komponenten. DCE-Implementationen sind inzwischen für die meisten Rechnerarchitekturen verfügbar. Verteilte Kommunikation in DCE basiert auf dem Fernaufruf (RPC). Verschiedene Dienste wie Namensdienst und verteilter Dateidienst sind in DCE bereits enthalten.

Das Sicherheitsmodell für verteilte Kommunikation in DCE [Osf92] basiert auf Kerberos [KoN93]. Eine zentrale Einheit, das sogenannte Schlüssel-Verteilungs-Zentrum (Key Distribution Center, KDC), verwaltet Identitäten und Schlüssel. Wenn zwei Kommunikationspartner miteinander kommunizieren möchten, kontaktiert einer von beiden das KDC. Dieses prüft dann die Authentizität des Initiators und vermittelt die Kommunikation, indem es ein sogenanntes Ticket ausstellt. Ein Ticket enthält zertifizierte Benutzerinformationen und einen Schlüssel zum Verschlüsseln der Verbindung zwischen den beiden Partnern. Die Kommunikationspartner erhalten jeweils die zertifizierten Benutzerinformationen und können daraus ablesen, wer der Kommunikationspartner genau ist und mit Zugriffslisten festlegen, welche Identitäten einen gewissen Aufruf durchführen dürfen. Die zertifizierten Benutzerinformationen sind nur begrenzte Zeit gültig und müssen dann erneuert werden. Kerberos basiert auf symmetrischer Verschlüsselung: Jeder Kommunikationspartner und jede Identität im System besitzt einen geheimen Schlüssel, den das KDC ebenfalls kennt. Durch geschickte Verwendung von symmetrischer Verschlüsselung wird sowohl die Abhörsicherheit von allen Verbindungen, als auch die Datenintegrität sichergestellt. Kerberos erlaubt auch Delegation. Eine Identität kann ein Ticket erstellen, das einen Teil ihrer Rechte beinhaltet und kann dies an einen anderen Partner weitergeben, der dann mit den delegierten Rechten arbeiten kann.

Das KDC stellt bei großen Systemen einen Engpaß dar. Daher können redundante KDCs verwendet werden, oder das System kann geteilt und mit verschiedenen KDCs ausgestattet werden. In diesem Fall wird allerdings die Kommunikation zwischen Partnern mit verschiedenen KDCs komplexer. DCE unterstützt Zugriffslisten, basierend auf Individuen und Gruppen, die allerdings nur im Wirkungsbereich des zugehörigen KDCs gültig sind.

Das Hauptproblem bei Verwendung des Kerberos-Authentifizierungsdienstes ist das zentrale KDC: Es muß gut gegen böswillige Zugriffe geschützt sein, da es alle geheimen Schlüssel aller Identitäten besitzt. Andererseits stellt es einen Engpaß im System dar, da es bei jeder Kommunikationsinitiierung involviert werden muß. Bei Verwendung mehrerer redundanter KDCs zur Umgehung dieses Engpasses multipliziert sich aber das Risiko, daß eins der Zentren nicht gut genug gegen unbefugte Zugriffe geschützt ist.

3.2.4 Java

Java [Fla96] besteht aus zwei Komponenten: Der Programmiersprache Java und dem Java-Ablaufsystem. Die Programmiersprache Java ist eine objektorientierte Sprache mit Einfachvererbung bei Klassen und Mehrfachvererbung bei Schnittstellen. Sie erlaubt – im Gegensatz zu anderen objektorientierten Sprachen wie C++ – nur objektorientierte Programmierung. So kennt

sie beispielsweise keine Funktionen und erlaubt keine Typumwandlungen, die im objektorientierten Sinne unzulässig sind. Diese Überprüfungen werden teilweise zur Übersetzungszeit vorgenommen, zum Teil müssen sie aber auch durch das Java-Ablaufsystem vorgenommen werden. Die verschiedenen Abläufe beim Starten eines Programms sind folgende [Sun95]:

- Übersetzen des Java-Quelltextes in einen Zwischencode, den sogenannten Bytecode
  Der Übersetzer prüft neben Syntaxfehlern auch die statisch prüfbaren Eigenschaften zur Einhaltung der Objektorientierung, wie beispielsweise, ob ein gewisser Zugriff auf eine Methode erlaubt ist (Einhaltung der Attribute “private”, “protected”).
- Laden des Bytecodes in die Ablaufumgebung
  Das Lademodul der Java-Ablaufumgebung prüft noch einmal die gleichen Dinge wie der Java-Übersetzer, da es sich gegen böswilliges Unterschieben von fehlerhaft übersetzten Klassen schützen muß. Da der Bytecode sehr stark an der Java-Programmiersprache orientiert ist, können die Überprüfungen auch auf Bytecodeebene noch einmal erfolgen.
- Ausführung der Methoden je nach Programm
  Während der Programmausführung werden Laufzeitprüfungen vorgenommen. Typumwandlungen werden überwacht, Zugriffe auf nicht belegten Speicher werden verhindert (Feldzugriffe mit falschen Indizes) und Zugriffe auf Systemressourcen (Dateien) werden überprüft.

Das Laufzeit-Sicherheitsmodell besteht aus drei Konzepten:

- Capabilities
  Objektreferenzen in Java sind Capabilities. Nur wer eine Objektreferenz besitzt, kann auf das Zielobjekt zugreifen.
- Schutz zwischen Klassen auf Vererbungs- und Paketebene
  Die Attribute private, protected und public, die anderen Klassen Zugriff auf Methoden und Instanzvariablen verweigern oder gestatten können, lassen sich nicht umgehen, wie dies beispielsweise bei C++ möglich ist. Sie werden daher als Basis für Schutz bestimmter Objekte vor unerlaubten Zugriffen verwendet.
- Zugriffslisten auf Herkunftsebene (Java-1.0) oder auf Privilegienebene (Java-1.2)
  In eine Methode kann eine Zugriffsliste hineinprogrammiert werden. Die Methode prüft dann, ob der Aufrufer bestimmte Privilegien hat und weist eventuell den Aufruf ab. Um die Sicherheitskonfiguration zentral festlegen zu können, erfolgt in vielen Fällen ein Aufruf an den globalen, sogenannten Sicherheitsverwalter (Security-Manager), der dann die Zugriffsliste implementiert. Diese Vorgehensweise wird beispielsweise bei Dateizugriffen angewandt.

Wir wollen nun die Zugriffslisten genauer betrachten. Wir betrachten hier nur Java-1.2 [Gon98], da Java-1.0 ein schwächeres Modell implementiert, das sich mit Java-1.2 darstellen läßt. Die Zugriffslisten müssen in die zu schützende Klasse hineinprogrammiert werden, wobei dies nicht auf Klassenebene, sondern auf Methodenebene erfolgen muß – um alle Methoden einer Klasse

zu schützen muß somit die Zugriffsliste in jede Methode hineinimplementiert werden. Die Zugriffslistenimplementation kann dann abfragen, mit welchen Privilegien die Methode aufgerufen wurde.

Die Privilegien werden folgendermaßen bestimmt: Das System wird in Domänen unterteilt. Jede Klasse ist genau einer Domäne zugeordnet. Es gibt beispielsweise eine Domäne, die die lokal (von Festplatte) geladenen Klassen umfaßt. Es kann nun weitere Domänen geben, die vom Netzwerk geladene Klassen umfassen. Dabei bilden jeweils Klassen (bzw. deren Objekte), die von einem bestimmten Ort geladen wurden, eine Domäne. Den Domänen können vom Administrator Privilegien zugewiesen werden. Bei den vom (möglicherweise nicht-vertrauenswürdigen) Netz geladenen Klassen kann man dabei durch Prüfung von Signaturen sicherstellen, daß die Klassen tatsächlich von einer gewissen Institution stammen. Die festlegbaren Privilegien können beispielsweise Schreib- oder Lesezugriff auf bestimmte Dateien oder die Erlaubnis, Netzwerkverbindungen zu erstellen, umfassen.

Ein Aktivitätsträger in Java hat aktuelle Privilegien, aufgrund derer die Überprüfungen der Zugriffslisten durchgeführt werden. Diese Privilegien berechnen sich aus der Schnittmenge aller Domänenprivilegien aller Aufrufer. Ein Aktivitätsträger, der in Domäne A gestartet wurde und dann in Domäne B springt, hat also dann nur die Privilegien, die sowohl Domäne A als auch Domäne B besitzen. Um nun Domäne B zu ermöglichen, explizit ihre Privilegien zu verwenden, kann sie diese Schnittmengenbildung außer Kraft setzen. Sie kann festlegen, daß höhere Aufrufer nicht bei der Schnittbildung berücksichtigt werden.

Wir wollen nun diese Strategie bei verschiedenen Fällen betrachten, bei denen privilegierter Code (eine Methode in einer privilegierten Domäne) und nicht-privilegierter Code interagieren. Diese Betrachtung findet man mit Alternativrealisierungen auch in [WBD+97].

- Sprung (durch Methodenaufruf) von nicht-privilegiertem Code in privilegierten Code
  Der privilegierte Code hat dabei implizit keine besonderen Rechte, sondern muß die Rechte explizit anfordern. Dies erfüllt die Eigenschaft der sicheren Vorgabewerte, hat aber den Nachteil, daß die Sicherheitsstrategie und der Anwendungscode gemischt werden müssen, um Aktionen mit hohen Privilegien auszuführen.

- Sprung (durch Methodenaufruf) von privilegiertem Code in nicht-privilegierten Code
  Der nicht-privilegierte Code erbt dabei keine Rechte und hat auch keine Möglichkeit, die Rechte zu erlangen. Es entsteht kein Sicherheitsproblem.

- Trojanisches-Pferd-Problem
  Der nicht-privilegierte Code kann versuchen, dem privilegierten Code Referenzen unterzuschieben, über die dann Aufrufe mit hohen Privilegien durchgeführt werden. Hier zeigt sich, daß die Bindung der Privilegien an einen Aktivitätsträger ungünstig ist: Wenn der privilegierte Code die Referenz bei sich (in einem privilegierten Objekt) abspeichert und ein anderer, interner (und damit privilegierter) Aktivitätsträger später auf die Referenz zugreift, werden ungewollt die hohen Privilegien verwendet.

Das Hauptproblem der Privilegienstrategie sind Trojanische-Pferd-Referenzen. Dies erscheint besonders problematisch, da sehr viele privilegierte Klassen existieren (Verwaltungsklassen wie "Vector" sind privilegiert) und das Java-Ablaufsystem keine Verbreitungskontrolle von Referenzen implementiert. Für die Privilegienvergabe und -benutzung gibt es schon Ansätze, durch ein formales Modell Aussagen über die Sicherheit eines Systems beweisen zu können [KaG98]. Durch die relativ komplexe Art, wie Privilegien verwaltet und an Threads gebunden werden, sind allerdings Sicherheitsaussagen nur mit hohem Aufwand zu beweisen. Das Privilegienmatrix-Problem, d.h. die Entscheidung, ob in einem Ablauf, bei dem der Initialzustand und der Zielzustand bekannt sind, ein Thread in einer bestimmten Domäne mit gewissen Privilegien gearbeitet hat, ist NP-vollständig.

Java beinhaltet auch einen Mechanismus für Fernaufrufe (RMI, [Sun96]). Man kann Objektreferenzen erzeugen, die auf ein Objekt auf einem anderen Rechner verweisen. Java implementiert diesbezüglich aber kein Sicherheitsmodell. Aufrufe an andere Rechner werden weder authentifiziert, noch wird in irgendeiner Weise die Datenintegrität oder der Schutz vor unbefugten Aufrufen sichergestellt.

3.2.5 Andere Ablaufsysteme

Wir wollen noch einige weitere Ablaufsysteme betrachten, bei denen besondere Sicherheitskonzepte eingesetzt werden.

Active Capabilities [CQL+96] ist ein Konzept, das konfigurierbare Capabilities realisiert. Eine Capability stellt eine Referenz auf ein Zielobjekt dar und besteht aus einem Capabilityobjekt. Wenn man einen Methodenaufruf an dem Zielobjekt durchführen möchte, teilt man dies dem Capabilityobjekt mit, und dieses sorgt dann dafür, daß der Aufruf ausgeführt werden kann. Es kann dazu die Art der Authentifizierung und andere Sicherheitsparameter festlegen. Diese Vorgehensweise trennt Applikationssemantik und Sicherheitsstrategie.

Das objektbasierte Experimentalsystem Legion [WWK96] und die Corba-Implementation distributed SOM (DSOM, Distributed System Object Model, [BBN96]) realisieren konfigurierbare Zugriffslisten. Statt eine vom System vorgegebene Zugriffsliste zu benutzen, kann man für ein Objekt auch selbst eine Methode angeben, die vor jedem Aufruf an dem Zielobjekt automatisch aktiviert wird und prüft, ob der Aufruf zulässig ist. Als Entscheidungsparameter können dann neben der Aufruferidentität und der aufzurufenden Methode auch der Programmzustand und andere im Programm verfügbare Daten verwendet werden.

Javascript ist eine typlose, objektbasierte Programmiersprache, die zunächst nur für die Programmierung von dynamischen HTML-Seiten verwendet wurde [Ste96]. Inzwischen wird Javascript jedoch auch für Skript-Programmierung von Diensten und für die Entwicklung von kleinen Anwendungen eingesetzt [RiR99]. Javascript-Programme in HTML-Seiten müssen speziellen Sicherheitsanforderungen unterliegen. Der Javascript-Interpreter überwacht daher die korrekte Verwendung von Objektreferenzen und Variablen. Je nach Quelle des Programms kann festgelegt werden, welche Rechte ein Javascript-Programm hat. Ein Programm, das von einer unbekannten Quelle aus dem Netzwerk stammt, kann beispielsweise zwar Inhalte der lo-

kalen Festplatte des Rechners laden und in einem Fenster darstellen lassen, kann jedoch nicht auf den dargestellten Inhalt zugreifen. Dies erreicht Javascript durch Unterteilung in Domänen. Eine Javascript-Domäne wird durch die Menge aller Programme und HTML-Seiten aus einer Quelle gebildet. Im Falle eines mit einer gewissen Identität signierten Programms ist dies die Menge aller mit dieser Identität signierten Programme. Im Falle einer URL als Quelle sind dies URLs mit dem gleichen Verzeichnispfad. Eine Unterdomäne besteht aus URLs mit dem Verzeichnis als Präfix, aber einem längeren Pfad. Bei der Quelle http://www.fau.de/t/test.html wäre also http://www.fau.de/t/x/test2.html in einer Unterdomäne, http://www.fau.de/test3.html jedoch nicht. Ein Programm kann nun zwar in anderen Fenstern Seiten darstellen, die nicht zu seiner Domäne oder einer Unterdomäne gehören, kann aber nur auf Fenster zugreifen, die zur gleichen Domäne oder einer Unterdomäne gehören. Die Implementation von Javascript bei aktuellen Webbrowsern wie Netscape Communicator oder Internet Explorer ist nicht im Quelltext verfügbar, aus den zahlreichen auftretenden Sicherheitslöchern läßt sich aber schließen, daß es keine zentrale Festlegung dieser Strategie gibt, sondern daß in jede Javascript-Methode (beispielsweise in jede Methode der Fenster-Klassen) die Strategie separat fest hineinimplementiert wurde.

Ein systemübergreifend betrachtetes Konzept ist das Konzept von Proxies [Neu93]. Ein Proxy stellt eine Objektreferenz dar, die nur gewissen Identitäten Aufrufe erlaubt. Falls eine solche Identität einen Aufruf durchführt, wird jedoch der Aufruf tatsächlich unter einer anderen Identität an dem Zielobjekt durchgeführt. Damit läßt sich Delegation von Rechten trotz Verwendung von Zugriffslisten erreichen. Ein Benutzer kann beispielsweise, um eine nur für ihn zugreifbare Datei zu drucken, eine Referenz auf die Datei erzeugen, die nur vom Druckverteiler verwendet werden kann, dann aber die Aktionen an der Datei unter der Benutzer-Identität durchführt.

3.3 Vergleich

Für den Vergleich der verschiedenen betrachteten Systeme werden die wichtigsten Kriterien aus Abschnitt 2.1 verwendet.

Das erste untersuchte Kriterium ist die Einfachheit des Designs. Während bei den meisten untersuchten Systemen das Design klar und überschaubar ist, da es nur auf wenigen Grundmechanismen basiert, bietet Corba eine so große Menge von Sicherheitsmechanismen, daß es verhältnismäßig schwer zu überblicken ist.

Das Kriterium der sicheren Standardwerte erfüllen zunächst einmal alle Systeme: Auf ein Zielobjekt kann man nur zugreifen, wenn man eine Capability besitzt bzw. wenn man in der Zugriffsliste des Ziels eingetragen ist; sonst (d.h. per Vorgabe) ist der Zugriff verboten. Bei objektorientierter Programmierung erfüllt jedoch keins der Systeme mehr das Kriterium: im Zuge von Methodenaufrufen übergebene Objektreferenzen stellen neue Capabilities dar, die per Vorgabe nicht mit Einschränkungen versehen sind. Sie müßten explizit geschützt werden. Wenn man die Parameterübergabe als explizite Erlaubnis des Zugriffs ansieht, würde das Kriterium erfüllt. Dies ist jedoch bei objektorientierter Programmierung sehr fragwürdig, daher wird das Kriterium in dem Fall als nicht erfüllt angesehen.

Aus den gleichen Gründen kann auch das Kriterium der kompletten Überwachung bei allen Systemen nur bei nicht-objektorientierter Programmierung als erfüllt angesehen werden. Lediglich Corba und DCOM erlauben durch Domänenstrategien komplette Überwachung von Zugriffen zwischen verschiedenen Domänen, womit das Kriterium bei objektorientierter Programmierung als teilweise erfüllt betrachtet werden kann. Ähnliches gilt für die Realisierung des kleinstmöglichen Zugriffsrechtes.

	Hydra	Amoeba	Spring	Corba	DCOM	Java
Einfachheit des Designs	+	+	+	-	+	+
Sichere Standardwerte	+	+	+	+	+	+
Sichere Standardwerte (bei objektorientierter Programmierung)	-	-	-	-	-	-
Komplette Überwachung	+	+	+	+	+	+
Komplette Überwachung (bei objektorientierter Programmierung)	-	-	-	(+) nur bei Domänen	(+) nur bei Domänen	-
Teilung der Rechte	-	-	-	-	-	-
Kleinstmögliches Zugriffsrecht	+	+	-	+	+	-
Kleinstmögliches Zugriffsrecht (bei objektorientierter Programmierung)	-	-	-	-	-	-

Tabelle 3.1 Vergleich der Betriebs- und Ablaufsysteme

3.4 Zusammenfassung

Die meisten objektbasierten Systeme verwenden als primären Sicherheitsmechanismus Capabilities, da diese auf Objektreferenzen abgebildet werden können und sich so am besten in objektbasierte Systeme integrieren lassen – es sind weder zusätzliche Abstraktionen noch Aufwand des Anwendungsprogrammierers nötig. Die Probleme der Verbreitungskontrolle von Capabilities und der Einschränkung des Zugriffs werden jedoch nicht zufriedenstellend gelöst; statt dessen werden meist als Zusatzmechanismus Zugriffslisten angeboten. Diese sind jedoch wegen der unvollkommenen Eingliederung in ein objektbasiertes System nur für grobgranulare Objektorientierung geeignet – für jede mögliche Konstellation muß überlegt werden, ob Probleme wie das Trojanische-Pferd-Problem existieren, was nur bei überschaubarer Menge von sicher-

heitsrelevanten Objektreferenzen möglich ist. Außerdem müssen die Zugriffslisten vom Anwendungsprogrammierer oder Systemadministrator konfiguriert werden, was ebenfalls nur bei geringer Menge von Zugriffslisten möglich ist und oft nicht von der Anwendungsprogrammierung getrennt werden kann. Eine Lösung, die auch bei feingranularer Objektorientierung erlaubt, Sicherheit auf hohem Abstraktionsniveau zu spezifizieren und von der Anwendungsprogrammierung weitgehend getrennt festzulegen, gibt es in den bisherigen Systemen nicht.

4 Schutz durch Sicherheitsmetaobjekte

In diesem Kapitel wird das im Rahmen dieser Arbeit entwickelte Schutzkonzept vorgestellt. Es basiert auf sogenannten Sicherheitsmetaobjekten, die die Basismechanismen für Schutz bereitstellen. Es wird gezeigt, daß man mit Sicherheitsmetaobjekten herkömmliche Mechanismen wie Capabilities und Zugriffslisten realisieren kann. Zusätzlich können Sicherheitsmetaobjekte Strategien wie rollenbasierte Identitäten, transitive Capabilities und transitive Zugriffslisten implementieren – Strategien, die mit klassischen Systemen nicht realisierbar sind. Diese Sicherheitsstrategien wurden im Rahmen dieser Arbeit entwickelt und berücksichtigen speziell die Sicherheitsanforderungen von objektorientierten Systemen. Bei dem Entwurf dieses Sicherheitsmodells wird neben den allgemeinen Kriterien für Sicherheitsmodelle aus Abschnitt 2.1 auf die Trennung von Programmierung der Sicherheitsstrategie und Anwendungsprogrammierung besonderen Wert gelegt.

4.1 Grundlagen der Meta-Programmierung

Ein Ablaufsystem kann man aus verschiedenen Sichtweisen betrachten. Der Anwendungsprogrammierer betrachtet das Ablaufsystem als Plattform, die die für ihn nötigen Abstraktionen bereitstellt. Der Programmierer, der das Ablaufsystem selbst programmiert oder verändert, muß eine andere Sicht haben. Er betrachtet die Implementation des Ablaufsystems und programmiert die Objekte, aus denen das Ablaufsystem besteht. Dies nennt man Meta-Programmierung [Mae87]. Ein Beispiel für Meta-Programmierung ist die Implementation eines Java-Interpreters in der Programmiersprache C. Der Anwendungsprogrammierer entwickelt Programme in Java, die auf diesem Java-Interpreter lauffähig sind. Er kann den Java-Interpreter benutzen, hat aber keinen Zugriff auf die internen Strukturen des Interpreters. Der Meta-Programmierer implementiert die Java-Maschine selbst.

Man kann auch mehrere Meta-Ebenen hierarchisch aufeinander aufbauen. Dies ist bei der PM-Systemarchitektur der Fall [Kle92]. Aspekte wie beispielsweise die Speicherverwaltung werden in der PM-Systemarchitektur durch Meta-Programmierung realisiert.

In vielen Fällen ist es sinnvoll, die strenge Trennung von Anwendungsebene und Metaebene aufzuweichen. Auch der Anwendungsprogrammierer möchte Zugriff auf die Implementation des eigenen Laufzeitsystems haben. Dies nennt man reflexive Programmierung. Man unterscheidet strukturelle Reflexion und Verhaltensreflexion. Strukturelle Reflexion umfaßt den lesenden Zugriff auf Eigenschaften des Ablaufsystems wie Klassen, Methoden und Speicherverwaltung. Verhaltensreflexion erlaubt zusätzlich das Verhalten des Ablaufsystems zu modifizieren [Bro98]. Metaxa[3] [KlG96] ist beispielsweise ein System, das Verhaltensreflexion gestattet, indem es die Modifikation der Semantik von Methodenaufrufen ermöglicht.

4.2 Sicherheitsmetaobjekte

Sicherheitsmetaobjekte ([RiH97], [RHK98]) sind Metaobjekte, die die Semantik von Objektreferenzen modifizieren. Sie können sowohl durch einen Meta-Programmierer realisiert werden, der für Anwendungen die benötigte Sicherheitsstrategie festlegt, als auch durch den Anwendungsprogrammierer selbst, wenn dieser eine spezielle Sicherheitsstrategie wünscht. Dies ist Verhaltensreflexion: Der Programmierer kann die Semantik der von ihm verwendeten Objektreferenzen modifizieren.

Bei dem hier vorgestellten Sicherheitskonzept wird von einem objektbasierten System ausgegangen: Als Dateneinheiten gibt es im System nur Objekte. Ein Objekt läßt sich nur ansprechen, wenn man eine Objektreferenz auf dieses besitzt.

Prinzipiell gibt es zwei verschiedene Zugriffsarten auf ein Objekt: Methodenaufrufe und Instanzvariablenzugriffe. Hier werden im weiteren nur Methodenaufrufe betrachtet, da sich Instanzvariablenzugriffe als Methodenaufrufe modellieren lassen: Eine Instanzvariable kann durch zwei Methoden modelliert werden, nämlich eine Methode zum Lesen der Variablen und eine zum Schreiben der Variablen. Instanzvariablenzugriffe stellen sich dann als Aufruf einer der beiden Methoden dar.

Dieses objektorientierte Modell wird durch Sicherheitsmetaobjekte (SMOs) erweitert; das sind spezielle Objekte, die auf Objektzugriffe Einfluß nehmen können. Ein SMO läßt sich an eine Objektreferenz anheften und besitzt dann die Kontrolle über alle Aktionen, die diese Objektreferenz betreffen.

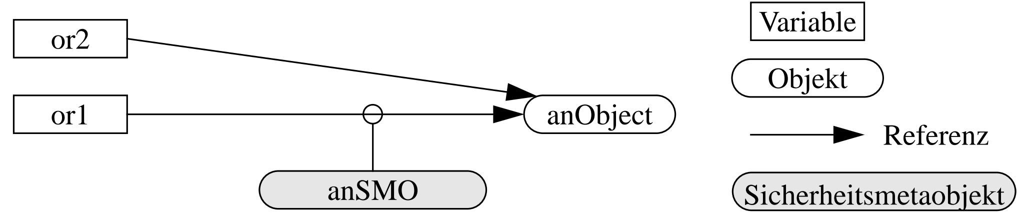

Abbildung 4.1 Eine mit einem Sicherheitsmetaobjekt geschützte Referenz

3. Das in der Literaturreferenz als MetaJava bezeichnete System wurde inzwischen in Metaxa umbenannt.

Abbildung 4.1 zeigt ein Beispiel. Objektreferenz `or1` referenziert das Objekt `anObject`. Sicherheitsmetaobjekt `anSMO` ist an diese Referenz angeheftet. Bei allen Zugriffen über die Objektreferenz `or1` wird automatisch vom Laufzeitsystem das Sicherheitsmetaobjekt `anSMO` involviert. Dieses kann Zugriffsschutz implementieren, kann aber auch Informationen über den Aufruf oder Aufrufer für andere Sicherheitsmetaobjekte bereitstellen. Es kann weitere Referenzen auf dasselbe Objekt im System geben, die ungeschützt sind (im Beispiel `or2`) oder mit anderen Sicherheitsmetaobjekten geschützt sind.

Sicherheitsmetaobjekte können vom Benutzer, von anderen Sicherheitsmetaobjekten oder vom Laufzeitsystem implementiert und an Objektreferenzen geheftet werden. Dieser Vorgang beeinflußt nur die Referenz, an die das SMO angeheftet wird. Ein Sicherheitsmetaobjekt kann an mehrere Referenzen angeheftet sein. Es können auch mehrere Sicherheitsmetaobjekte an eine Referenz geheftet sein. In diesem Fall werden die Sicherheitsmetaobjekte nacheinander involviert. Es ist nicht möglich, von außen ein Sicherheitsmetaobjekt von einer Referenz zu entfernen. Dies kann nur das Sicherheitsmetaobjekt selbst.

Falls die Referenz dupliziert wird, sind automatisch an das Referenzduplikat dieselben Sicherheitsmetaobjekte angeheftet. Referenzen können z.B. im Zuge von Variablenzuweisungen, Parameterübergabe und Rückgabe von Ergebnissen dupliziert werden.

Es gibt zwei Möglichkeiten, ein SMO an eine Referenz zu heften: quellorientiert und zielorientiert. Dies beeinflußt die Sicht des SMOs auf die Referenz. Ein an eine Referenz quellorientiert angeheftetes SMO betrachtet die Referenz als ausgehend, d.h. die Referenz zeigt von dem Applikationsteil, in dem sich das SMO befindet, nach außen, also in einen anderen Applikationsteil. Ein zielorientiert angeheftetes SMO betrachtet die Referenz als eingehend, also von außen nach innen. Da die Art, ein SMO anzuheften, lediglich dessen Sichtweise auf die entsprechende Referenz beeinflußt, kann ein SMO quellorientiert an eine Referenz und gleichzeitig zielorientiert an eine andere Referenz geheftet sein. In Abbildung 4.2 ist das SMO `anSMO` zielorientiert an die Referenz `or1` angeheftet. Aufrufe über diese Referenz sind aus Sicht des SMOs ankommende Aufrufe. Gleichzeitig ist es quellorientiert an Referenz `or2` angeheftet und betrachtet Aufrufe über diese Referenz als abgehende Aufrufe.

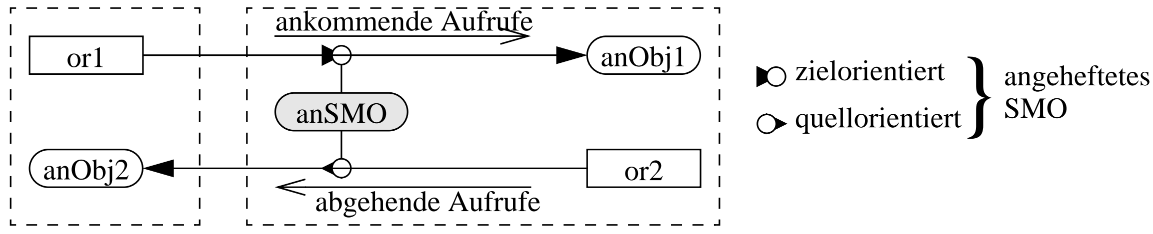

Abbildung 4.2 Quell- und zielorientiert angeheftete SMOs

SMOs können verschiedene Ereignisse behandeln. Sie können Aufrufe abfangen oder übergebene Parameter und Rückgabewerte prüfen oder schützen, bevor diese tatsächlich übergeben werden. Die dazu nötigen Mechanismen werden in den folgenden Abschnitten vorgestellt. Die Beispiele werden in Java-ähnlicher Syntax dargestellt. Sie entsprechen im wesentlichen der

Syntax des im Rahmen dieser Arbeit implementierten Prototyps, der genauer im Kapitel 6 beschrieben wird, sind jedoch zum Teil etwas vereinfacht (nicht zum Verständnis nötige Details wurden weggelassen).

4.3 Konfigurierbare Capabilities

4.3.1 Einfacher Schutz

Eine Objektreferenz ist in dem hier entwickelten Modell zunächst eine Capability für das gesamte Interface des Zielobjektes. Um konfigurierbare Capabilities (revozierbare, zeitlich begrenzt gültige oder einschränkende) zu implementieren, benötigt man ein Sicherheitsmetaobjekt, das die entsprechende Einschränkung implementiert.

In Listing 4.1 soll eine Objektreferenz auf eine Liste mit zeitlich begrenzter Gültigkeit versehen werden. Der erste Teil des Programmes zeigt die Benutzung eines entsprechenden Sicherheitsmetaobjektes. Das Sicherheitsmetaobjekt `s` wird an die Listenreferenz `l` geheftet. Es wird hier mit zielorientierter Sicht an die Referenz geheftet (mit `dstAttachTo`), das bedeutet, daß es Aufrufe über die geschützte Referenz als ankommende Aufrufe betrachtet. Durch das Anheften

```
List l = ..... ;                                // eine Referenz auf eine Liste
SecurityMeta s =
        new MetaExpire(new Date(1,7,1997));     // Metaobjekt erzeugen
l = s.dstAttachTo ( l );                        // Metaobjekt an Referenz
                                                // heften
...                                             // die Referenz 'l' ist nun
                                                // geschützt
x.untrustedMethod(l);                           // l an nicht-vertrauenswürdigen
                                                // Programmteil übergeben

class MetaExpire extends SecurityMeta {         // die MetaExpire-Klasse
     final Date d;                              // Gültigkeitsende

     MetaExpire(Date d) { this.d = d; }         // Gültigkeitsende in d speichern

     void incomingCall (ObjectRef o,            // diese Methode prüft
                        Method m,               // Aufrufe über geschützte
                        ParameterList p,        // Referenzen
                        PrincipalList pl) {
          if (d.isBefore(getCurrentDate()))     // Referenz schon ungültig?
               throw (new SecException(...));   // dann ist der Aufruf unzulässig
     }                                          // sonst Aufruf zulassen
}

class X { untrustedMethod(List lst) {lst.Get();...}}     // Aufruf wird geprüft
```

Listing 4.1 Ein Sicherheitsmetaobjekt für zeitlich begrenzte Gültigkeit

entsteht eine neue, nunmehr geschützte Referenz. Damit wird die Variable `l` überschrieben, so daß in diesem Beispiel die einzige ungeschützte Referenz auf die Liste verschwindet. Bei jedem Aufruf über die geschützte Referenz wird nun automatisch durch das Laufzeitsystem die `incomingCall`-Methode des Sicherheitsmetaobjektes aktiviert, die prüft, ob die Gültigkeitsdauer bereits überschritten ist und in diesem Fall den Aufruf durch Erzeugen einer Ausnahme verhindert. Die geschützte Referenz wird an einen nicht-vertrauenswürdigen Programmteil übergeben. Dieser hat nun keine Möglichkeit mehr, nach dem Ende der Gültigkeitsdauer auf die Referenz zuzugreifen. Die `incomingCall`-Methode des SMOs erhält als Parameter weitere Informationen über den Aufruf: die Objektreferenz, an der der Aufruf ausgeführt werden soll (bei Aufrufen über diese Referenz wird das SMO nicht mehr involviert), die aufzurufende Methode, die Parameter und Informationen über den Aufrufer (diese werden in einem späteren Kapitel genauer betrachtet).

Abbildung 4.3 zeigt das Ergebnis: Bei der Übergabe der geschützten Referenz `l` an einen anderen Programmteil wird die Referenz dupliziert. Das Sicherheitsmetaobjekt ist zielorientiert an die Referenz geheftet und betrachtet daher Aufrufe als ankommende Aufrufe; die SMO-Methode `incomingCall` wird bei jedem Aufruf automatisch aktiviert und prüft die Zulässigkeit des Aufrufs.

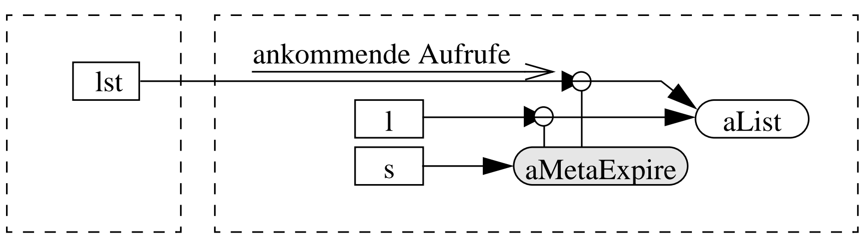

Abbildung 4.3 Schutz von Referenzen

Die `MetaExpire`-Klasse ist generisch implementiert, sie kann alle Arten von Objekten schützen. Die im Beispiel zu schützende Liste enthält keine Unterstützung für Schutz. Implementation der Applikationssemantik und der Sicherheitskonfiguration sind getrennt.

Es gibt noch zwei weitere Funktionalitäten, die Capabilities im allgemeinen zur Verfügung stellen: Revokation von Capabilities und Einschränkung der Zugriffe.

Die Implementation für ein Revokations-SMO (Listing 4.2) sieht ähnlich wie die Implementation zeitlich begrenzter Gültigkeit aus. Die `MetaRevocation`-Klasse enthält eine Statusvariable, die die Information enthält, ob Aufrufe zugelassen werden sollen oder nicht. Zunächst werden Aufrufe zugelassen. Durch Aufruf der `revoke`-Methode des SMOs können alle mit diesem SMO geschützten Referenzen ungültig gemacht und dadurch revoziert werden.

```
class MetaRevocation extends SecurityMeta {
     boolean expired=false;
     void revoke() { expired=true; }

     void incomingCall (...) {
          if (expired) throw (new SecurityException(...));
     }
}
```

Listing 4.2 Ein Sicherheitsmetaobjekt für Revokation von Referenzen

Einschränkung von aufrufbaren Methoden kann ebenfalls durch ein entsprechendes SMO implementiert werden. Hierzu muß die Metainformation über den Aufruf, die der `incomingCall`-Methode vom Laufzeitsystem übergeben wird, genutzt werden. In Listing 4.3 bekommt das SMO im Konstruktor ein Interface bzw. eine Klasse übergeben, auf deren Methoden die möglichen Aufrufe beschränkt werden. Im anschließenden Beispiel wird eine Listenreferenz geschützt: Es wird eine Referenz erzeugt, mit der man nur Methodenaufrufe tätigen kann, die durch das Interface `ReadOnlyList` implementiert werden, d.h. es können nur lesende Methoden der Liste aufgerufen werden.

```
class MetaRestriction extends SecurityMeta {     // die MetaRestriction-Klasse
     Class allowedInterface;
     MetaRestriction (Class allowedInterface) {
          this.allowedInterface=allowedInterface;
     }

     void incomingCall (ObjectRef o,
                        Method m,
                        ParameterList p,
                        PrincipalList pl) {
          if (!allowedInterface.implements(m))       // erlaubte Methode?
               throw (new SecurityException(...));   // nein
     }
}

// Benutzungsbeispiel:
List l=....;
l = new MetaRestriction (ReadOnlyList.class) . dstAttachTo (l);
```

Listing 4.3 Ein Sicherheitsmetaobjekt für Restriktion von Referenzen

4.3.2 Einfache Transitivität

In den Beispielen im vorigen Abschnitt wurde nur der manuelle Schutz von Objektreferenzen durch den Benutzer betrachtet. In objektorientierten Systemen werden jedoch oft Objektreferenzen zwischen sich nicht vertrauenden Programmteilen ausgetauscht; diese Objektreferenzen muß man in vielen Fällen gegen unbefugte Aufrufe des anderen Programmteils schützen. Der Programmierer müßte an allen Stellen wo Objektreferenzen an einen anderen Programmteil übergeben werden, SMOs an diese heften. Dies würde zu einer Vermischung von Sicherheitsstrategie und Semantikimplementation führen. Ferner würde es dem in Abschnitt 2.1 erwähnten Prinzip der kompletten Überwachung widersprechen: Wenn der Programmierer auch nur an einer Stelle im Programm vergißt, eine Referenz zu schützen, kann dies zu Sicherheitsproblemen führen.

SMOs erlauben daher die Überwachung von ausgetauschten Referenzen. Durch diese Überwachung ist es möglich, transitive Sicherheitsstrategien zu implementieren.

Zunächst soll einfache Transitivität betrachtet werden: Referenzen, die im Zuge von Methodenaufrufen zurückgegeben werden und dadurch als neue Referenz zwischen den Programmteilen etabliert werden, werden mit dem gleichen Schutz versehen, wie die Referenz, über die der Aufruf getätigt wird. Danach wird gezeigt, daß diese einfache Transitivität nicht alle Fälle abdecken kann, und es wird komplette Transitivität eingeführt, die alle Fälle umfaßt.

Einfache Transitivität umfaßt Rückgabewerte über geschützte Referenzen. Betrachten wir beispielsweise die geschützte Referenz auf eine Liste in Abschnitt 4.3.1. Die Liste könnte über eine `get`-Methode verfügen, die dem Aufrufer eine Referenz auf ein Eintragsobjekt zurückliefert. Diese Referenz wäre bei der dort vorgestellten Implementation nicht geschützt. Die Referenz auf die Liste kann der Aufrufer nach Ablauf der Gültigkeit nicht mehr verwenden, jedoch könnte er Referenzen auf Elemente über die `get`-Methode erhalten und gespeichert haben.

Um auch solche Referenzen mit Schutz zu versehen, werden SMOs über zu transferierende Referenzen informiert, bevor sie tatsächlich übergeben werden. Sie können dazu eine `outgoingRef`-Methode implementieren, die vom Laufzeitsystem für jede Referenz aktiviert wird, die zurückgegeben werden soll.

Der Ablauf sieht dann folgendermaßen aus:

- Über eine mit einem SMO geschützte Listen-Referenz wird die `get`-Methode aufgerufen.
- Das Laufzeitsystem aktiviert nun die `incomingCall`-Methode des SMOs. Diese kann entscheiden, ob der Aufruf zulässig ist.
- Falls der Aufruf nicht vom SMO abgewiesen wurde, wird er nun ausgeführt.

- Nach Beendigung des Aufrufs wird die `incomingCallTerminated`-Methode des SMOs aufgerufen. Diese dient einerseits zu Informationszwecken für das SMO, andererseits kann das SMO jedoch auch den Aufruf z.B. durch eine Fehlermeldung abbrechen. Die Ausführung des Aufrufs wird dabei nicht rückgängig gemacht, aber dem Aufrufer wird kein Ergebnis des Aufrufes mehr übermittelt.
- Nun müßte die Referenz auf das Ergebnisobjekt an den Aufrufer zurückgeliefert werden. Diese wird jedoch zunächst an die `outgoingRef`-Methode des SMOs gegeben. Das SMO kann nun entscheiden, wie verfahren werden soll: Es kann einen Fehler auslösen und damit verhindern, daß die Ergebnisreferenz zurückgegeben wird, es kann die Referenz schützen, indem es ein anderes SMO oder sich selbst an diese anheftet, oder es kann die Referenz ungeschützt zurückgeben.

Listing 4.4 zeigt die entsprechende Implementation. Die `outgoingRef`-Methode bekommt ein `ObjectRef`-Objekt übergeben, das Objektreferenzen kapselt. Der Aufruf `dstAttachToRef` bewirkt, daß an die gekapselte Referenz ein SMO (in diesem Fall das SMO selbst) geheftet wird. Gekapselte Objektreferenzen wurden eingeführt, um den Umgang mit Objektreferenzen zu vereinfachen: Ohne “Übergabe per Referenz” würden Referenzmodifikationen sonst unnötig kompliziert. In Abbildung 4.4 ist das Ergebnis zu sehen: Ein Aufrufer der `get`-Methode erhält als Ergebnis eine ebenfalls geschützte Referenz auf das Eintragsobjekt. Im Fall der Implementation zeitlich begrenzter Gültigkeit wird dann nach Ablauf der Gültigkeitsdauer die erhaltene Eintragsreferenz ebenfalls ungültig.

```
class MetaExpire extends SecurityMeta {

     void incomingCall(ObjectRef o, Method m, ...) { ... }

     void outgoingRef(ObjectRef o) {          // Objektreferenz o wird zurückgegeben
          this.dstAttachToRef(o);              // o mit dem Metaobjekt selbst schützen
     }
}
```

Listing 4.4 Schutz von abgehenden Referenzen

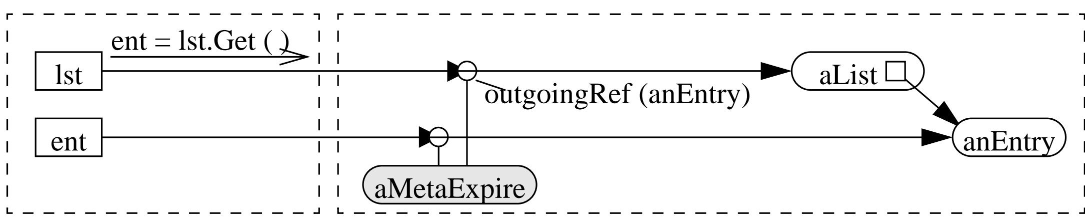

Abbildung 4.4 Einfache Transitivität

4.3.3 Komplette Transitivität

Die Implementation im vorigen Abschnitt realisiert Transitivität nur für Referenzen, die als Rückgabewerte übergeben werden. Referenzen, die als Parameter übergeben werden, scheinen zunächst unkritisch, da diese in die andere Richtung zeigen. Unsere Liste könnte z.B. eine Such-Methode implementieren, die als Parameter eine Referenz auf ein Elementobjekt bekommt und prüft, ob sich das Element schon in der Liste befindet. Da die Referenz nun in die andere Richtung zeigt, nämlich von der Liste nach außen, benötigt man für sie keinen Schutz gegen unbefugte Aufrufe. Ganz ohne Schutz kommt man hier jedoch auch nicht aus, sonst kann diese Referenz als trojanisches Pferd wirken; über sie können unbemerkt Referenzen nach außen dringen. In Abbildung 4.5 ist diese Situation dargestellt. Über die geschützte Referenz `lst` wird die Such-Methode der Liste mit einer Referenz (`trojH`), die später als trojanisches Pferd benutzt wird, aufgerufen. Die Such-Methode iteriert durch die Liste und testet für jeden Eintrag, ob dieser mit dem übergebenen Objekt übereinstimmt (Listing 4.5). Dies erfolgt durch Aufruf der `equals`-Methode an der übergebenen Objektreferenz (`e`). Die `equals`-Methode erhält nun wiederum ein Listenelement als Parameter. Das Objekt `aTrojH` besitzt nun eine Referenz auf ein Listenelement, die nicht geschützt ist. Die Sicherheitsstrategie ist noch nicht komplett transitiv.

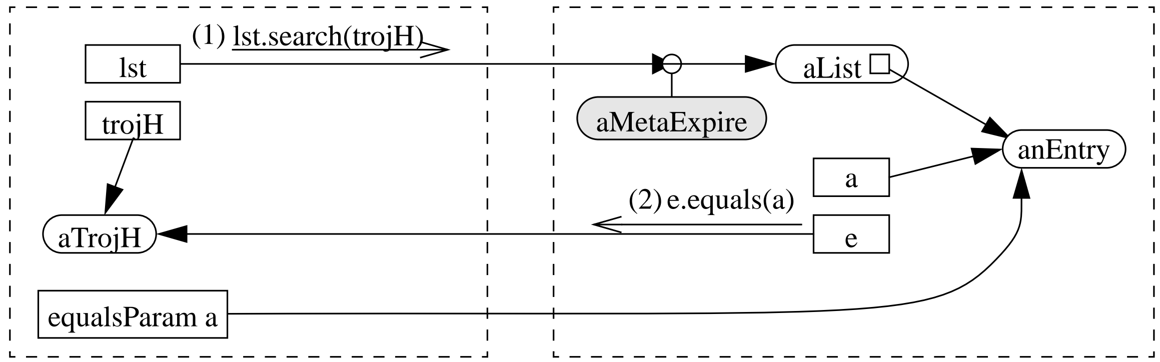

Abbildung 4.5 Mißbrauch von Referenzen als trojanisches Pferd

```
class List { .....
     boolean search(Entry e) {                // Such-Methode der Liste
          for (Entry a=first; ....) {         // Durch die Liste iterieren
               if (e.equals(a)) {return true;} // Eintrag gefunden
          }
          return false;                       // Eintrag nicht gefunden
}    }
```

Listing 4.5 Die Such-Methode der Liste.

Das Problem ist die übergebene Referenz: Wenn eine Referenz über eine geschützte Referenz übergeben wird, muß sie ebenfalls geschützt werden, und zwar nicht gegen unbefugte Aufrufe, sondern vielmehr nur gegen Weitergabe ungeschützter Referenzen. Solche Referenzen zeigen in die andere Richtung: von den geschützten Objekten aus nach außen. Daher benötigt ein Si-

cherheitsmetaobjekt, das eine solche Referenz überwachen soll, eine andere Sicht auf die Referenz: quellorientierte Sicht. Es betrachtet dann Aufrufe über die Referenz als abgehende Aufrufe, übergebene Parameter als herausgegebene Referenzen, Rückgabewerte als ankommende Referenzen. Um eine Referenz quellorientiert zu schützen, kann man ein Sicherheitsmetaobjekt quellorientiert (durch die Methode `srcAttachTo` des Metaobjektes) an diese anheften (Listing 4.6).

```
class MetaExpire extends SecurityMeta {
     void incomingCall(ObjectRef o, Method m, ParameterList p) { ... }
     void outgoingCall(...) {}         // abgehende Aufrufe unbeschränkt zulassen

     void outgoingRef(ObjectRef o) {   // Objektreferenz o verläßt geschützten Bereich
          this.dstAttachToRef(o);      // 'o' zielorientiert schützen
     }
     void incomingRef(ObjectRef o) {   // Objektreferenz o betritt geschützten Bereich
          this.srcAttachToRef(o);      // 'o' quellorientiert schützen
     }
}
```

Listing 4.6 Komplette Transitivität

Abbildung 4.6 zeigt das Ergebnis: Die initiale Referenz `lst` ist zielorientiert geschützt. An den bei dem Methodenaufruf (1) übergebenen Parameter `trojH` heftet sich das Sicherheitsmetaobjekt selbst quellorientiert (2). Wenn nun über diesen Parameter wiederum eine Methode aktiviert wird (3), kann das Sicherheitsmetaobjekt Parameter zielorientiert schützen (4), so daß der ursprüngliche Aufrufer keine ungeschützten Referenzen bekommt.

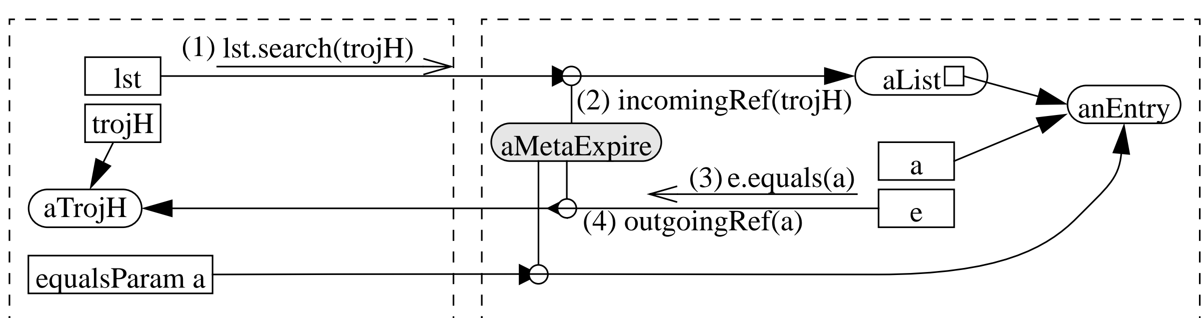

Abbildung 4.6 Komplette Tansitivität

Diese Sicherheitsmetaobjekt-Implementation ist vollständig transitiv und kann für alle Arten von Schutzstrategien verwendet werden, wie Zugriffsbeschränkungen, Revokation von Referenzen, zeitlich beschränkte Gültigkeit. Beispielsweise könnte man ein Sicherheitsmetaobjekt implementieren, das das Listen-Objekt gegen Schreibzugriffe schützt. Der Schutz wirkt dann nicht nur auf die Liste, sondern auch auf die Elemente der Liste, und dies, ohne daß die Listen-Implementation verändert werden muß.

4.4 Subjekte / Zugriffslisten

Obwohl man viele Probleme, die in bisherigen Systemen wegen Verbreitungs-Problemen der Capabilities mit Zugriffslisten gelöst wurden, nun mit transitiven Capabilities lösen kann, gibt es mehrere Klassen von Problemen, für die man tatsächlich Zugriffslisten benötigt:

- Initiale Kontaktaufnahme
  Wenn ein Applikationsteil mit einem anderen zum ersten Mal interagiert (die Vermittlung kann z.B. über einen Nameserver erfolgen), möchte der Zielapplikationsteil in der Lage sein, den Zugriff auf bestimmte andere Programmteile oder Benutzer einzuschränken. Da vorher keine Verbindung zwischen den Programmteilen besteht, kann dies nicht durch Weitergabe von Capabilities erfolgen.

- Rechteverstärkung
  Man benötigt oft Capabilities mit eingeschränkten Rechten, deren Rechte von höher privilegierten Applikationen verstärkt werden können, d.h. diese Applikationen können dann z.B. ohne Einschränkung zugreifen. Ein Beispiel wäre eine Referenz auf einen Drucker. Eine Benutzerapplikation könnte eine Referenz auf ein Druckerobjekt besitzen, die lediglich das Drucken ermöglicht. Wenn das Drucken aus irgendeinem Grund nicht funktioniert, kann sie die Druckerreferenz an ein Administrationsobjekt übergeben, das nun mit erhöhten Rechten über die Druckerreferenz zugreifen kann und den Drucker über die Referenz administrieren kann.

Die Zugriffslisten selbst kann man durch eine kleine Konzepterweiterung intuitiv mit den SMOs aus dem vorigen Abschnitt implementieren [RiK98]: Die entsprechenden Methoden des SMOs (`incomingCall` etc.) müssen dazu lediglich Informationen über den Aufrufer erhalten. Dies könnten Informationen über den aufrufenden Benutzer oder Programmteil sein. Anhand dieser Informationen kann von Zugriffslisten-SMOs die Zulässigkeit des Aufrufes überprüft werden. Wie schon bei den Capabilities vorgeführt, lassen sich dann auch transitive Zugriffslisten implementieren.

Statt nun die für die Zugriffslisten nötige Identitäts-Information statisch vom System vorzugeben und zum Beispiel an Programmteile, Aktivitätsträger oder Benutzer zu binden, wird diese Information ebenfalls mit SMOs realisiert, um freie Konfigurierbarkeit von Identitäten für Aufrufe zu ermöglichen.

Dazu werden folgende Arten von SMOs unterschieden:

- **Zugriffslisten-SMOs**
  Diese können Aufrufe prüfen und anhand der aufgerufenen Methode, der Parameter und des Aufrufers entscheiden, ob ein Aufruf zulässig ist. Ein Capability-SMO ist daher ein Spezialfall des Zugriffslisten-SMOs, da es nur die Methode, nicht aber den Aufrufer prüft.

- **Identitäts-SMOs**
  Diese führen keine Zugriffskontrollen durch. Sie stellen lediglich Information über die Identität des Aufrufers (das aufrufende Subjekt) zur Verfügung, die von den Zugriffslisten-SMOs dann ausgewertet werden kann.

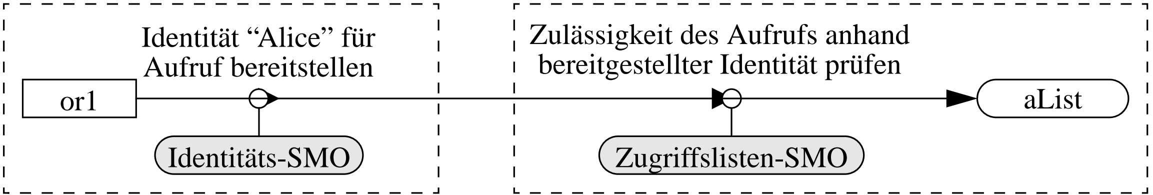

Abbildung 4.7 Identitäts-SMO und Zugriffslisten-SMO

Abbildung 4.7 zeigt ein Beispiel für die Verwendung der beiden Arten von SMOs. Die Liste `aList` wird mit einem Zugriffslisten-SMO geschützt, das Aufrufe nur bestimmten Benutzern erlaubt. Der Benutzer, der über diese Referenz zugreifen möchte, muß nun ein Identitäts-SMO an diese Referenz anheften, das bei Aufrufen seine Identität zur Verfügung stellt. Objektreferenz `or1` ist solch eine Referenz: Wenn man über `or1` eine Methode aufruft, stellt das Identitäts-SMO für den Aufruf zusätzliche Metainformationen zur Verfügung, in unserem Beispiel die Information, daß der Aufruf von “Alice” getätigt wird. Diese Information wird der weiteren Kette von SMOs mit übergeben; sie ist jedoch nur SMOs (im Beispiel dem Zugriffslisten-SMO) zugänglich. Das Zielobjekt (`aList`) erhält diese Informationen nicht; dort wird (sofern die SMOs dem Aufruf zugestimmt haben) nur die Zielmethode aufgerufen.

Falls an eine Objektreferenz kein Identitäts-SMO angeheftet ist, wird keine Identität für den Aufruf bereitgestellt, d.h. der Aufruf wird anonym durchgeführt. Wenn in dem Beispiel kein Identitäts-SMO an `or1` geheftet wäre, könnte man über die Referenz nicht zugreifen, da das Zugriffslisten-SMO nur bestimmten Identitäten Zugriff gewährt und keine Identität bereitgestellt würde.

Die Identitäten selbst werden nicht durch das Modell festgelegt. Nach Bedarf kann man Subjekte des Systems mit Identitäten assoziieren. In vielen Systemen werden z.B. Benutzer als Subjekte angesehen. Hier könnte ein Identitäts-SMO dann die Information bereitstellen, daß ein gewisser Benutzer (im Beispiel “Alice”) einen Aufruf durchführt. Da das Identitäts-SMO benutzerimplementierbar ist, muß man natürlich dafür sorgen, daß die bereitgestellte Information von dem Identitäts-SMO bewiesen wird, z.B. indem das SMO die Kenntnis von Loginnamen und Paßword des Benutzers oder die Kenntnis eines privaten Schlüssels des Benutzers nachweist. Genauer wird auf die Möglichkeiten in der Beschreibung der Prototyp-Implementation (Abschnitt 6) eingegangen.

4.4.1 Einfache Zugriffskontrolllisten

Ein Zugriffslisten-SMO ist ähnlich wie ein Capability-SMO implementiert (Listing 4.7). In der `incomingCall`-Methode wird getestet, ob der momentan durchzuführende Aufruf erlaubt ist. Dazu wird die Liste von Aufruferidentitäten (`pList`) verwendet. In dieser Liste befinden sich alle Aufruferidentitäten, die von Identitäts-SMOs als Verantwortliche für diesen Aufruf genannt wurden (in Abbildung 4.7 ist dies nur eine Identität, nämlich “Alice”). Unsere Implementation erlaubt nur Aufrufe, die von der Identität `principalAlice` autorisiert wurden.

```
class AclSMO extends SecurityMeta {                // Zugriffslisten-SMO
....
     void incomingCall (ObjRef targetObj,
                        Method m,
                        ParameterList param,
                        PrincipalList pList)       // diese Identität(en) führt den
     {                                             // Aufruf durch
          if (pList.contains(principalAlice))      // Nur eine bestimmte Identität
               return;                             // darf aufrufen
          else
               throw new SecurityException();
     }
}
```

Listing 4.7   Ein Zugriffslisten-SMO

Um nun diese Information zur Verfügung zu stellen, muß man ein entsprechendes Identitäts-SMO implementieren und an die zu authentifizierende Referenz anheften. Da der Aufrufer das SMO anheftet, muß das SMO Quellsicht auf die Referenz haben, und es muß daher quellorientiert (mit `srcAttachTo`) angeheftet werden. Aufrufe betrachtet es dann als abgehende Aufrufe.

```
class PrincipalSMO extends SecurityMeta {          // Identitäts-SMO
....
     void outgoingCall (ObjRef targetObj,           // abgehender Aufruf:
                         Method m,
                         ParameterList param,
                         PrincipalList pList)      // diese Identität(en) führt den
     {                                             // Aufruf durch
          pList.add(principalAlice);               // Identität Alice mit in die Liste
     }                                             // aufnehmen
}
```

Listing 4.8   Ein Identitäts-SMO

Bei der Implementation (Listing 4.8) muß die Methode `outgoingCall` implementiert werden: Sie fängt abgehende Aufrufe über eine Referenz ab. Im Beispiel fügt sie der Liste von für den Aufruf verantwortlichen Identitäten die Identität `principalAlice` hinzu.

Die vorgestellte Art, Zugriffslisten mit Information über die Aufruferidentität zu versorgen, hat den Vorteil, daß für jede Referenz separat bestimmt werden kann, mit welcher Identität Aufrufe authentifiziert werden sollen. Durch Duplizierung einer Referenz kann man sogar über eine Referenz unter verschiedenen Identitäten Methoden aufrufen. In Listing 4.9 werden die Referenzen `l1` und `l2` erzeugt, die auf das gleiche Objekt zeigen, aber unterschiedliche Identitäts-SMOs benutzen.

```
List l=.....                                     // geschützte Listenreferenz
List l1=new PrincipalSMO1().srcAttachTo(l);
List l2=new PrincipalSMO2().srcAttachTo(l);
```

Listing 4.9 Unterschiedliche Authentifizierung über die gleiche Referenz

Zwei Probleme entstehen bei diesem Verfahren allerdings, die in den folgenden Abschnitten gelöst werden:

- Explizites Anbinden
  In objektorientierten Applikationen werden viele Referenzen zwischen Applikationsteilen ausgetauscht. Um Authentifizierung über alle solche Referenzen zu implementieren, muß an vielen Stellen in der Applikation Code eingefügt werden, der Referenzen mit Identitäts-SMOs versieht. Dadurch werden Applikationssemantik und Sicherheitsstrategie gemischt.
- Capability-Eigenschaft
  Eine Referenz mit Identitäts-SMO (wie `or1` in Abbildung 4.8) ist eine Capability für Aufrufe unter dieser Identität an der Referenz. Wenn man solch eine Referenz unbedacht an einen anderen Applikationsteil gibt, kann dieser unter der fremden Identität Methodenaufrufe durchführen. Bei dem Methodenaufruf `or2.Pass(or1)` würde in Abbildung 4.8 die Referenz `or1` inklusive des Identitäts-SMOs an den oberen Applikationsteil weitergegeben. Dieser kann dann unter der von dem Identitäts-SMO bereitgestellten Identität auf das Objekt `aList` zugreifen.

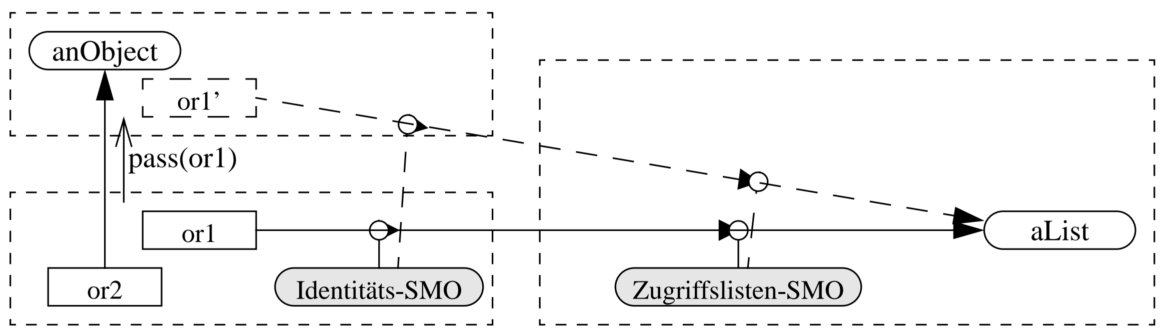

Abbildung 4.8 Versehentliche Weitergabe von identitätsbehafteten Referenzen

4.4.2 Virtuelle Domänen

Zur Lösung oben genannter Probleme definieren wir sogenannte virtuelle Domänen:

> Eine *virtuelle Domäne* besteht aus einer Menge von Objekten und einer Menge von Objektreferenzen. Die Objekte können lediglich auf Objektreferenzen zugreifen, die zur gleichen virtuellen Domäne gehören.

Wenn ein Objekt zu einer virtuellen Domäne gehört, müssen seine Instanzvariablen und alle lokalen Variablen von in dem Objekt laufenden Aktivitätsträgern daher automatisch zur gleichen virtuellen Domäne gehören. Eine Objektreferenz auf ein Objekt kann in mehreren Domänen vorhanden sein. (Solch eine Referenz ist zumindest in der Domäne enthalten, die das Objekt selbst enthält.)

Virtuelle Domänen kann man in beliebigen Granularitätsstufen bilden: Eine virtuelle Domäne kann eine gesamte Applikation umfassen, sie kann aber auch einen Applikationsteil beinhalten. Abbildung 4.9 zeigt ein Beispiel: Das Objekt `anObject` liegt in Domäne `d1`, demzufolge müssen auch seine Instanzvariablen `or1` und `or2` zu `d1` gehören. Die Ziele dieser Instanzvariablen (`o1`, `o2`) gehören zu einer anderen Domäne (`d2`).

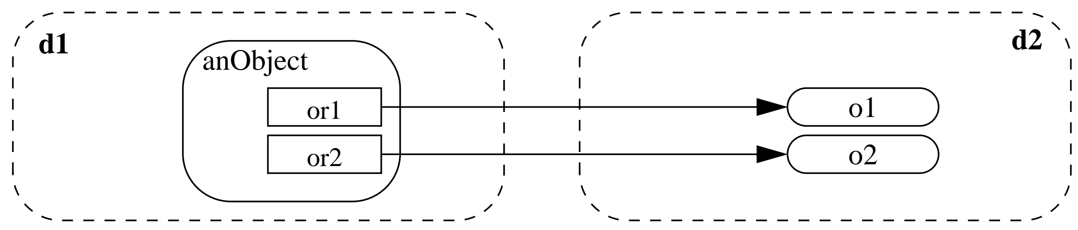

Abbildung 4.9 Virtuelle Domänen

Eine virtuelle Domäne sollte Objekte und Objektreferenzen beinhalten, die sich gegenseitig vertrauen. Die Objekte einer Domäne vertrauen einander, daß sie mit Rechten, die sie erhalten (z.B. Objektreferenzen, die sie bekommen), sinnvoll umgehen, d.h. keine böswilligen Aktionen mit diesen Rechten durchführen.

Objekte in verschiedenen Domänen vertrauen einander nicht oder nur begrenzt. Sie sollten daher keine Objektreferenzen mit angeheftetem Identitäts-SMO über Domänengrenzen hinweg übergeben. Um dies sicherzustellen, muß bei der Übergabe von identitätsbehafteten Referenzen über Domänengrenzen die Identität (bzw. das Identitäts-SMO) entfernt werden. Falls Anwendungen starke Interaktion zwischen verschiedenen Domänen benötigen, tritt dieser Fall oft auf – dieses Problem durch die Anwendung selbst lösen zu lassen, scheidet daher aus. Der Anwendungsprogrammierer müßte einen Teil der Sicherheitsstrategie selbst implementieren und dürfte an keiner Stelle die Entfernung des Identitäts-SMOs vergessen, da sonst Sicherheitsprobleme entstehen.

Man könnte nun die virtuellen Domänen als zusätzlichen Mechanismus zum Sicherheitsmodell hinzufügen. Dadurch würde der Mechanismus jedoch komplizierter und, wie später gezeigt wird, inflexibel. Daher wird hier ein anderer Weg beschritten: Die virtuellen Domänen werden ohne Zusatzmechanismen des Sicherheitsmodells realisiert. Sie werden nur durch spezielle SMOs implementiert.

Um zu erreichen, daß die Weitergabe von Referenzen an andere Domänen der Kontrolle von SMOs obliegt, müssen Referenzen von anderen Domänen und in andere Domänen durch spezielle SMOs vor unkontrollierter Referenzübergabe geschützt werden. Da Aufrufe über domänen-

überschreitende Objektreferenzen die einzige Möglichkeit darstellen, Objektreferenzen in andere Domänen zu transferieren, genügt es, solche Referenzen zu schützen, um virtuelle Domänen zu etablieren.

Eine virtuelle Domäne wird durch ihre Grenze definiert: Alle Referenzen aus der Domäne und in die Domäne sind mit sogenannten Grenz-SMOs geschützt. Der Rand der Domäne bildet sich also aus den angehefteten Grenz-SMOs (Abbildung 4.10).

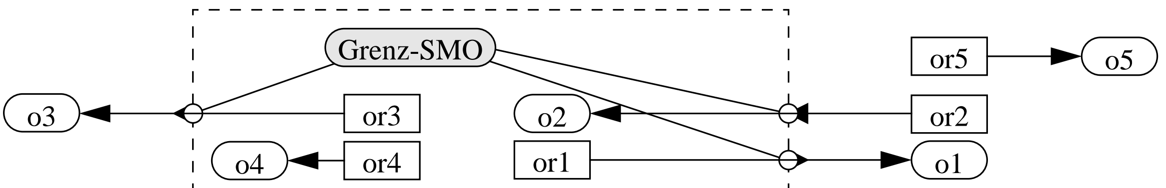

Abbildung 4.10 Virtuelle Domäne mit geschütztem Rand

Das Grenz-SMO erfüllt zwei Aufgaben: Es muß die virtuelle Domäneneigenschaft erhalten und zusätzlich bei Referenzen, an die es quell-angeheftet ist (`or1` und `or3` im Beispiel), Identitätsinformation zur Verfügung stellen.

Initial (bei der Erzeugung einer virtuellen Domäne) vorhandene Referenzen müssen die Domänen-Eigenschaft erfüllen, alle neu entstehenden Referenzen sollten jedoch automatisch (d.h. durch SMOs) die Eigenschaft erfüllen.

Grenzübergreifende Referenzen können nur entstehen bzw. sich ändern, wenn Methodenaufrufe über die Grenze erfolgen. Wir betrachten daher nun die verschiedenen auftretenden Fälle, um zu verdeutlichen, wie die Grenz-SMOs die Domäneneigenschaft erhalten:

- ankommende externe Referenz
  Der Fall, daß eine externe Referenz die Domäne betritt (ankommende externe Referenz), tritt bei Parameterübergabe über eingehende Referenzen (`or2.method(or5)`) und Ergebnisrückgabe über ausgehende Referenzen auf (Über `or1` wird eine Methode aufgerufen, diese liefert `or5` als Rückgabewert zurück). Der ankommenden externen Referenz muß dann ein Grenz-SMO (quellorientierte Sicht) angeheftet werden (`or5'` in Abbildung 4.11).

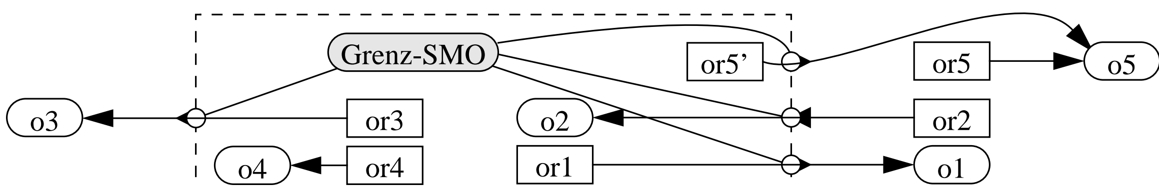

Abbildung 4.11 Ankommende externe Referenz

- ankommende interne Referenz
  Der Fall, daß eine interne Referenz, also eine Referenz auf ein internes Objekt, die Domäne betritt (ankommende interne Referenz), tritt auf, wenn beispielsweise `or2` über die Re-

ferenz `or1` zurückgegeben wird oder über `or2` als Parameter übergeben wird. Für diese Situationen gibt es zwei Möglichkeiten, die Domäneneigenschaft zu erhalten. Entweder man entfernt das Grenz-SMO von der Referenz (Abbildung 4.12) oder man fügt es noch einmal an, so daß die Referenz zweimal die Grenze überquert (Abbildung 4.13). Die letztere Vorgehensweise hat den Vorteil, daß einmal ausgegebene Referenzen, also Referenzen, die in die Hände anderer, nicht-vertrauenswürdiger Domänen gekommen sind, auch wenn sie wieder zurückkommen, nicht mehr als interne Referenzen behandelt werden. Dadurch kann man, wie wir später sehen werden, böswilliges Unterschieben von Referenzen verhindern. Wir verwenden daher die zweite Vorgehensweise, d.h. das nochmalige Anfügen eines Grenz-SMOs.

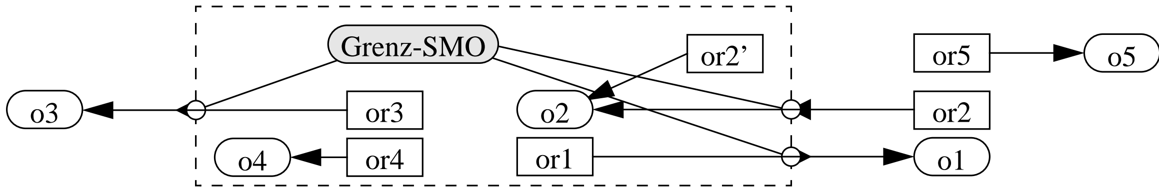

Abbildung 4.12 Ankommende interne Referenz mit Entfernung des Grenz-SMO

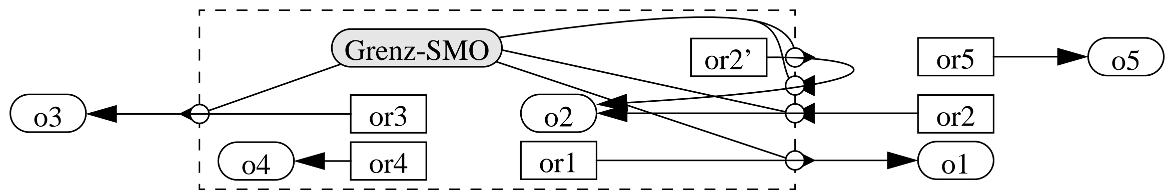

Abbildung 4.13 Ankommende interne Referenz mit Anfügen des Grenz-SMO

- abgehende interne Referenz
  Dieser Fall tritt auf, wenn eine interne Referenz (wie `or4`) die Domäne verläßt. Dies kann beispielsweise über `or1` als Parameter oder über `or2` als Rückgabewert geschehen. An diese Referenz wird das Grenz-SMO mit Quellsicht geheftet. (Abbildung 4.14)

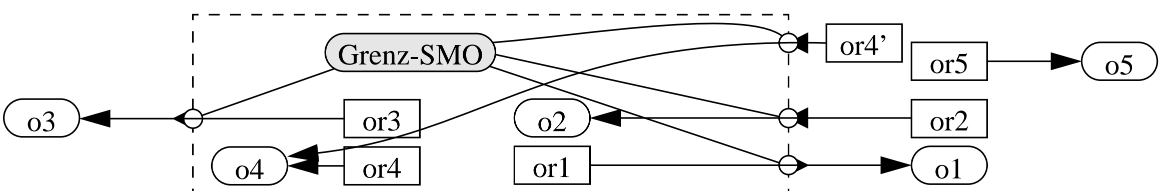

Abbildung 4.14 Abgehende interne Referenz über or1 / or2

- abgehende externe Referenz
  Bei externen Referenzen (also Referenzen auf externe Objekte), die die Domäne verlassen, muß eine versehentliche Weitergabe von Rechten verhindert werden. Eine externe

Referenz besitzt ein angeheftetes Grenz-SMO, das Aufrufe authentifiziert. Dieses Grenz-SMO muß entfernt werden, bevor die Referenz an andere Domänen übergeben wird. Beispiel: `or3` wird über `or1` oder `or2` nach außen gegeben. (Abbildung 4.15)

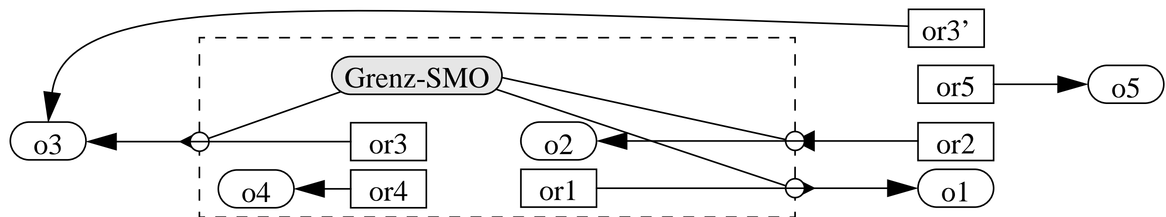

Abbildung 4.15 Abgehende externe Referenz or3 über or1 / or2

4.4.3 Hierarchische Domänen und gesamtes System

Virtuelle Domänen können auch hierarchisch verschachtelt sein (Abbildung 4.16). Dann muß eine Domäne vollständig in einer anderen Domäne liegen. Sich nur teilweise überlappende Domänen werden nicht zugelassen. Die Grenz-SMOs liegen genau in der Domäne, die sie schützen (auch nicht in einer weiter innen liegenden). Das bedeutet, daß verschiedene Domänen nicht dasselbe Grenz-SMO verwenden können.

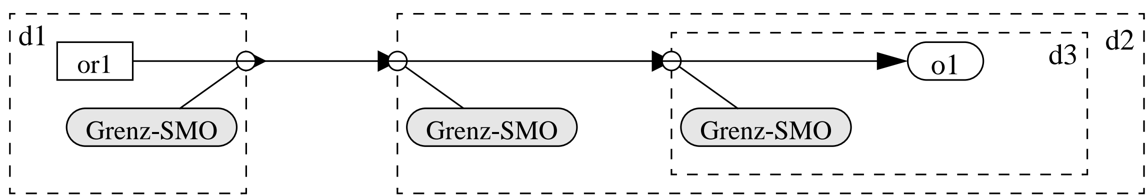

Abbildung 4.16 Verschachtelte Domänen

Die mit solchen Grenz-SMOs in den bisherigen Beispielen implementierte Semantik ist (bis auf die Hierarchisierung) die klassische Strategie “domänenbasierte Identitäten”. Das Grenz-SMO von Domäne `d1` bestimmt in Abbildung 4.16, wie Aufrufe über `or1` authentifiziert werden. Sobald `or1` die Domäne verläßt und z.B. Domäne `d2` betritt, bestimmt deren Grenz-SMO die Authentifizierung. Bei der Verwendung von einem einzigen Grenz-SMO pro Domäne erhält man also die Strategie domänenbasierte Identitäten – eine Strategie, die sich durch Aufnahme der Domänen in das Sicherheitsmodell auch einfacher hätte realisieren lassen.

Durch Verwendung verschiedener Grenz-SMOs innerhalb einer Domäne und durch zusätzliche Verwendung von reinen Identitäts-SMOs lassen sich jedoch auch andere Strategien implementieren, wie beispielsweise rollenbasierte Identitäten, die in Abschnitt 4.5 genauer betrachtet werden.

4.4.4 Interne Referenzen

Ein bisher noch nicht betrachtetes Problem sind interne Referenzen. Falls an eine Objektreferenz ein Zugriffskontroll-SMO angeheftet ist, kann es nötig sein, Identitätsinformation zur Verfügung zu stellen, um auf die Referenz zuzugreifen. Es gibt nun zwei Möglichkeiten, dieses Problem zu lösen:

- Man definiert Identitätsinformation, die bei internen Referenzen automatisch verwendet wird.
- Man muß zusätzlich zum Zugriffskontroll-SMO ein Identitäts-SMO an die Referenz heften. Dieses Identitäts-SMO wird automatisch entfernt, wenn die Referenz eine Domänengrenze überschreitet.

Die zweite Alternative scheint die bessere, da sie freier konfigurierbar ist. Abbildung 4.17 zeigt ein Beispiel: Objektreferenz `or2` referenziert `o2` und ist durch ein Zugriffslisten-SMO geschützt. Um lokal in der Domäne zugreifen zu können, muß ein Identitäts-SMO angeheftet werden (`Id-SMO1`). Falls diese Referenz aus der Domäne (über `or1`) herausgegeben wird, verschwindet das Identitäts-SMO, und `or2'` ist die resultierende Referenz.

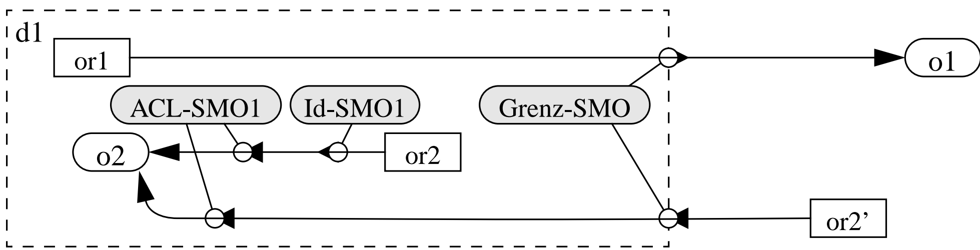

Abbildung 4.17 Interne Referenz wird herausgegeben

4.4.5 Realisierung

Für die Implementation von Grenz-SMOs benötigen wir einen zusätzlichen Mechanismus: SMOs müssen von Referenzen entfernt werden können. Dies darf natürlich nicht unkontrolliert geschehen, sondern das Entfernen kann nur durch das angeheftete SMO selbst erfolgen. Wenn wir nicht wie in den vorigen Beispielen nur ein einziges Grenz-SMO in einer Domäne haben, sondern die Grenze durch mehrere verschiedene Grenz-SMOs aufgespannt wird, benötigen wir zusätzlich Metainteraktion zwischen SMOs. Die verschiedenen Grenz-SMOs müssen bei einem Methodenaufruf miteinander kommunizieren, um die Domäneneigenschaft zu erhalten.

Listing 4.10 zeigt das Implementationsgerüst. Die entscheidenden Methoden sind `incomingRef` und `outgoingRef`. Gemäß unserer obigen Überlegung muß sich bei `incomingRef` das Grenz-SMO in jedem Fall selbst an die ankommende Referenz anheften. Die `outgoingRef`-Methode ist etwas komplizierter. Sie muß testen, ob die herauszugebende Referenz Identitäts-SMOs quellorientiert angeheftet hat und diese entfernen. Dabei werden diese Identitäts-SMOs nur einschließlich des ersten Grenz-SMOs entfernt, um auch hierarchische Domänen realisieren zu können. Metainteraktion findet in dieser Methode wie folgt statt: Zu-

```
class BorderPrincipalSMO extends SecurityMeta {
     void outgoingCall (...) {pList.add(somePrincipal);} // Ident. Info. liefern
     void outgoingRef (ObjectRef outRef) {
          Object[] metaObjs=outRef.getSrcSMOs();      // quell-orientierte SMOs

          for (int n=0; n<metaObjs.length; n++)       // Ident.-SMOs entfernen
               if (metaObjs[n] instanceof PrincipalSMORemovable) {
                    metaObjs[n].removeFrom(outRef);
                    if (metaObjs[n].isBorderObject())       // Grenz-SMO?
                         return;                            // dann Ende
               }
          this.dstAttachTo(outRef);                         // sonst selbst anheften
     }
     void incomingRef (ObjectRef inRef) { this.srcAttachTo(inRef); }

     Object metaInteraction() {              // Rückgabe von 'this' geschützt (nur
          return ..... (this);               // einige Meth. ansprechbar:
     }                                       // removeFrom, isBorderObject)

     void removeFrom (ObjectRef ref) {
          if (this.isSrcAttachedTo(ref)) ...// dann 'this' von ref entfernen
     }
     boolean isBorderObject() {return true;}
}
```

Listing 4.10 Implementation eines Grenz-Identitäts-SMO

nächst besorgt sich das SMO Referenzen auf alle quellorientiert angehefteten SMOs (mit `getSrcSMOs`). Die von dieser vom Laufzeitsystem bereitgestellten Methode zurückgegebenen SMO-Referenzen werden durch das Laufzeitsystem gesammelt, indem bei allen SMOs die Methode `metaInteraction()` aufgerufen wird. Der Rückgabewert dieser Methode wird dem Aufrufer von `getSrcSMOs` in die Hand gegeben. Normalerweise wird diese Methode entweder nichts zurückliefern, um Metainteraktion nicht zuzulassen, oder eine geschützte Referenz auf sich selbst zurückliefern, um den Aufrufer nur bestimmte Methoden aufrufen zu lassen wie in diesem Fall.

Nachdem die `outgoingRef`-Methode nun alle Referenzen gesammelt hat, probiert sie der Reihe nach, alle SMOs zu entfernen. Das gelingt nur bei Identitäts-SMOs. Bis einschließlich der Grenze werden solche SMOs entfernt.

In Abbildung 4.18 ist ein komplizierter Fall dargestellt: Über Referenz `or2` wird die Referenz `or1` herausgegeben. Dabei entsteht die Referenz `or1'`. Die Zugriffskontrolllisten-SMOs bleiben an der Referenz, egal ob sie quell- oder zielorientiert angeheftet sind. Die quellangehefteten Identitäts-SMOs (hier nur `Id-SMO1`) werden bis zur Grenze entfernt, das Grenz-SMO selbst (`Grenz-SMO2`) wird auch entfernt. Alles andere bleibt. Die Beziehung zwischen den Objektreferenzen und den angehefteten SMOs ist keine Referenzbeziehung, daher dürfen solche Beziehungen auch virtuelle Domänengrenzen überschreiten, wie z.B. bei der Referenz `or1'` das angeheftete `ACL-SMO1`.

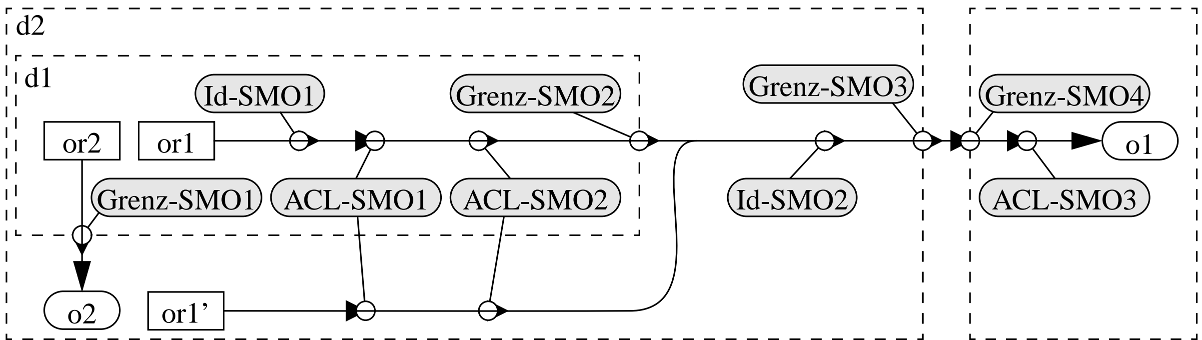

Abbildung 4.18 Herausgabe von Referenz or1

4.5 Rollenbasierte Identitäten

Wie wir in den vorigen Abschnitten bereits gesehen haben, kann eine SMO-Grenze auch durch verschiedene Grenz-SMOs gebildet werden. Durch geschickte Wahl der Grenz-SMOs an den verschiedenen ausgehenden Referenzen kann man die im Rahmen dieser Arbeit entwickelte Strategie *rollenbasierte Identitäten* implementieren.

Aufrufe, die von einer Domäne ausgehen, werden mit gewissen Identitätsinformationen authentifiziert. Bei rollenbasierten Identitäten hängt die Identitätsinformation von der Referenz ab. Die Idee dabei ist, daß Programmteile oft in verschiedenen Rollen mit anderen Programmteilen interagieren. Beispielsweise könnte eine Systemadministrationsapplikation mit Benutzerapplikationen in der Rolle “Administrator” interagieren. Diese Rolle beinhaltet erhöhte Rechte, wie z.B. das Recht zum Terminieren von Benutzerapplikationen. Wenn die Systemadministrationsapplikation aber beispielsweise Ergebnisse auf einem Drucker ausdrucken möchte, benötigt sie diese Rechte nicht, und die Rolle “Administrator” paßt auch semantisch nicht zu der Drucktätigkeit. Das Drucken kann sie in der Rolle “Benutzer” durchführen. Beispielsweise könnte dies die Rolle der Person sein, die die Systemadministrationsapplikation gestartet hat. Diese Vorgehensweise entspricht dem in Abschnitt 2.1 vorgestellten Prinzip des kleinstmöglichen Zugriffsrechts.

Die Realisierung dieses Konzeptes erfolgt durch Referenzen, an die verschiedene Grenz-Identitäts-SMOs angeheftet sind. Die Referenz zu anderen Applikationen muß im Beispiel dann ein SMO angeheftet haben, das die Administrationsidentität zur Authentifizierung benutzt. Die Referenz zum Drucker hat ein SMO angeheftet, das normale Benutzeridentität verwendet. Die Administrationsapplikation kann sogar zusätzlich eine zweite Referenz auf den Drucker besitzen, die zur Administration des Druckers verwendet wird und daher mit der Administrationsidentität arbeitet. Für die jeweils nötige Aktion werden immer nur die minimal nötigen Rechte verwendet, so daß die Gefahr eines versehentlichen Mißbrauchs der Rechte verringert wird (Abbildung 4.19).

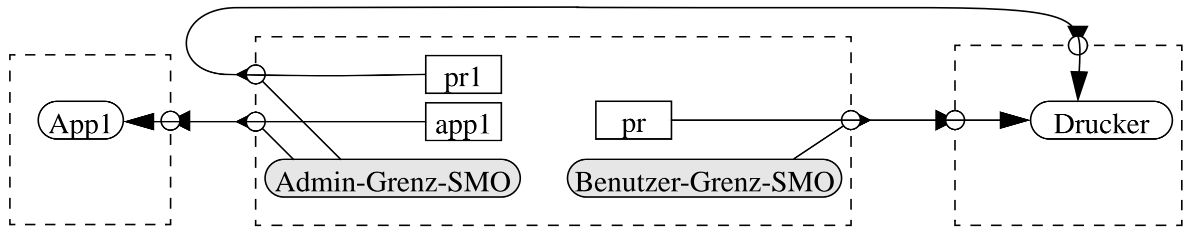

Abbildung 4.19 Rollenbasierte Identitäten

Bei statischen Konfigurationen wie in dem Beispiel kann man rollenbasierte Identitäten intuitiv verwenden. Falls jedoch dynamisch Objektreferenzen über solche rollenbasierte Referenzen ausgetauscht werden, muß man zusätzliche Überlegungen anstellen. In diesem Fall gibt es verschiedene Einschränkungen, die man berücksichtigen und nach denen man die Konfiguration wählen muß.

Wir werden drei solche Konfigurationen betrachten und deren jeweilige Eigenschaften untersuchen:

- Rollenbasierte disjunkte Identitäten
- Hierarchische Domänen
- Explizite Identitäten

Wir werden diese drei Konfigurationen anhand der Implementation eines Druckverteilers (Printer-Spoolers) evaluieren. Ein Druckverteiler muß einerseits mit Benutzerapplikationen, die drucken möchten, interagieren. Andererseits muß er mit Druckern und temporären Zwischendateien (Spool-Dateien) interagieren, wozu er spezielle Authentifizierung benötigt. Dies ist wichtig, um zu verhindern, daß beispielsweise Applikationen unbefugt auf Drucker und Zwischendateien zugreifen können. Drucker und Zwischendateien müssen also mit Zugriffskontrolllisten geschützt sein, die nur bestimmten Identitäten Zugriff erlauben.

Zunächst wird gezeigt, daß bei Verwendung domänenbasierter Identitäten und threadbasierter Identitäten Sicherheitsprobleme entstehen. Danach wenden wir die drei obigen Konfigurationen auf diese Situation an. Alle drei Konfigurationen lösen die Probleme, haben jedoch unterschiedliche Eigenschaften.

4.5.1 Druckverteiler mit domänenbasierten Identitäten

Um einen Druckverteiler mit domänenbasierten Identitäten zu implementieren, muß man zunächst die Domänen festlegen. Dafür gibt es zwei Möglichkeiten:

- Das Drucksystem, bestehend aus Druckverteilern und Druckern, bildet eine Domäne oder
- der Druckverteiler bildet eine Domäne, und jeder Drucker bildet eine Domäne.

Das im folgenden betrachtete Trojanische-Pferd-Referenz-Problem tritt bei beiden Alternativen exakt an der gleichen Stelle auf, es genügt daher, nur eine der beiden Alternativen zu betrachten. Wir beschränken uns daher auf die zweite.

Temporäre Druckdateien und Drucker sollen gemäß der Anforderung speziell geschützt sein. Die Domäne, in der sich der Druckverteiler befindet, muß daher entsprechende Authentifizierung implementieren, die zum Ansprechen von temporären Druckdateien und den Druckern nötig ist. Benutzer haben kein Recht, auf temporäre Druckdateien zuzugreifen.

Ein böswilliger Benutzer, der eine temporäre Druckdatei eines anderen Benutzers lesen möchte, kann zwar nun nicht direkt auf diese zugreifen, er kann jedoch versuchen, sie dem Druckverteiler unterzuschieben, indem er versucht, diese zu drucken.

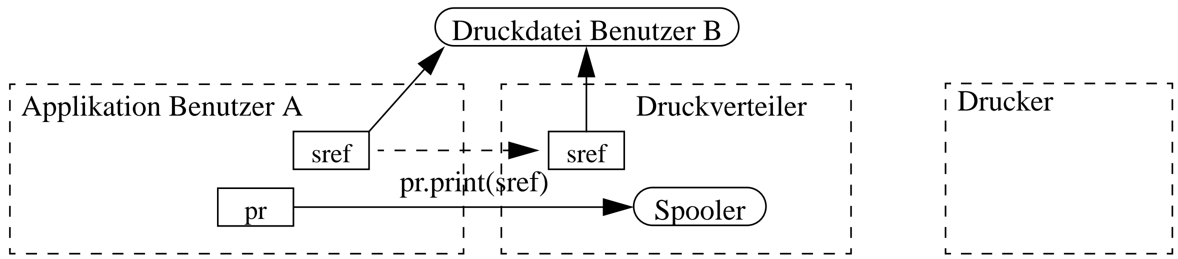

Abbildung 4.20 Druckverteiler mit domänenbasierter Identität

Abbildung 4.20 zeigt ein Beispiel. Die Applikation von Benutzer A hat sich eine Referenz auf eine temporäre Druck-Datei von Benutzer B besorgt (`sref`). Wegen mangelnder Zugriffsrechte (nur der Druckverteiler kann darauf zugreifen) kann sie jedoch nicht von der Druckdatei lesen. Nun ruft sie die Druck-Methode des Druckverteilers auf und übergibt ihr die Datei-Referenz als Parameter. Der Druckverteiler greift nun auf die Datei zu, um sie für Benutzer A zu drucken. Dabei verwendet er automatisch (und unbeabsichtigt) seine Rechte als Druckverteiler. Benutzer A kann so durch einen Trick trotzdem an den Inhalt dieser Datei gelangen. Die einzige Lösung dieses Problems besteht darin, die Domänenrechte pro Aufruf ein- und ausschalten zu können, was jedoch wiederum zu einer Mischung von Sicherheitsstrategie und Applikationssemantik führt.

4.5.2 Druckverteiler mit threadbasierten Identitäten

Bei threadbasierten Identitäten ist die Konfiguration nicht ganz einfach. Threadbasiert bedeutet, daß man einem Thread bestimmte Privilegien geben kann, die über Authentifizierung von Aufrufen entscheidet. Beispiel Druckverteiler: Ein Benutzer ruft die `print`-Methode des Druckverteilers auf. Der Thread, der diesen Aufruf durchführt, hat zunächst keine besonderen Privilegien. Der Druckverteiler muß nun mit dem Drucker oder temporären Druckdateien arbeiten, wozu er Privilegien benötigt. Er bindet dazu diese Privilegien (“Authentifizierung als Druckverteiler”) an den aktuellen Thread und führt die entsprechenden Aufrufe durch. Dabei muß er darauf achten, daß er vor der Interaktion mit Benutzerobjekten diese Thread-Rechte wieder entfernt, da sonst der Benutzer die Rechte des Druckverteilers durch trojanische-Pferd-Objekte

stehlen kann. Hierbei ist daher die Vermischung von Applikationssemantik und Sicherheitsstrategie unausweichlich. Innerhalb der Applikation (des Druckverteilers) müssen mehrfach die Thread-Rechte geändert werden. Auch das Prinzip der sicheren Standardwerte ist nicht eingehalten, da nach dem Binden von Privilegien an den Thread dieser automatisch die Rechte durch die gesamte Aufrufkette hindurch verwendet.

4.5.3 Rollenbasierte disjunkte Identitäten

Eine Lösungsmöglichkeit für das Druckverteilerproblem ist die Strategie *rollenbasierte disjunkte Identitäten*. Bei rollenbasierten disjunkten Identitäten wird davon ausgegangen, daß ein Applikationsteil (in diesem Fall der Druckverteiler) mit verschiedenen Parteien exklusiv kommuniziert. D.h. er kann zwar mit jeder Partei interagieren und Objektreferenzen austauschen, darf aber die dort erhaltenen Referenzen nicht an andere Parteien weitergeben. Beim Beispiel Druckverteiler bedeutet dies, daß der Druckverteiler zwar von den Benutzerapplikationen eine Objektreferenz auf den zu druckenden Text übergeben bekommt, diese aber nicht an andere Parteien, wie z.B. den Drucker, weitergeben darf. Er könnte den Text beispielsweise vorverarbeiten, ein neues Text-Objekt (z.B. Postscript-Objekt) daraus erzeugen und dieses an den Drucker übergeben.

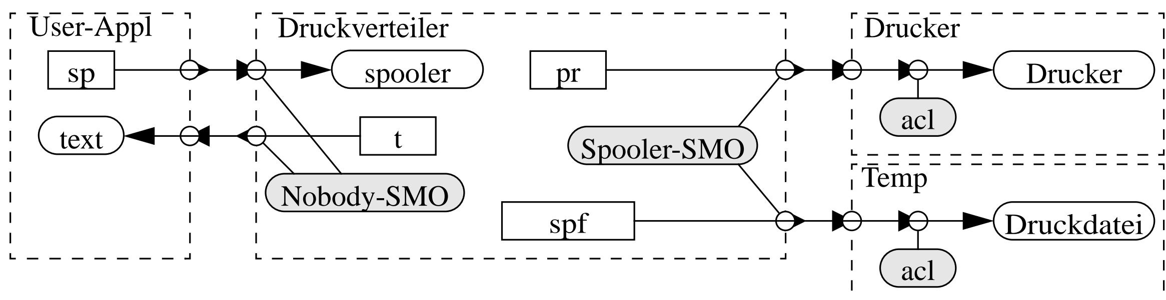

Abbildung 4.21 Rollenbasierte disjunkte Identitäten

Abbildung 4.21 zeigt einen Druckverteiler mit rollenbasierten disjunkten Identitäten. Jeder Benutzer verfügt über eine eigene virtuelle Domäne, der Druckverteiler besteht aus einer eigenen virtuellen Domäne, und die Drucker und temporären Druckdateien befinden sich in einer eigenen Domäne. Druckdateien und Drucker sind mit Zugriffslisten geschützt, die lediglich dem Druckverteiler Zugriff erlauben. Hier soll nur der Druckverteiler genauer betrachtet werden. Die Identitäts-SMOs der anderen Domänen sind für diese Betrachtung irrelevant und sind daher nicht in der Abbildung enthalten. Für Interaktion mit Benutzern verwendet der Druckverteiler das `Nobody-SMO`, das keine Authentifizierung implementiert. Für Interaktion mit temporären Druckdateien und Druckern wird das `Spooler-SMO` verwendet, das Authentifizierung als Druckverteiler implementiert. Es werden nur die minimal benötigten Rechte für jede nötige Operation verwendet: Interaktion mit dem Benutzer wird nicht authentifiziert.

Wenn nun der Benutzer eine Referenz auf eine temporäre Druckdatei besitzt, kann er auf diese nicht zugreifen, da er sich nicht als Druckverteiler authentifizieren kann. Er kann nun versuchen, diese temporäre Datei zu drucken, indem er die `print`-Methode des Druckverteilers aufruft und ihr die Referenz auf die temporäre Datei als Parameter übergibt.

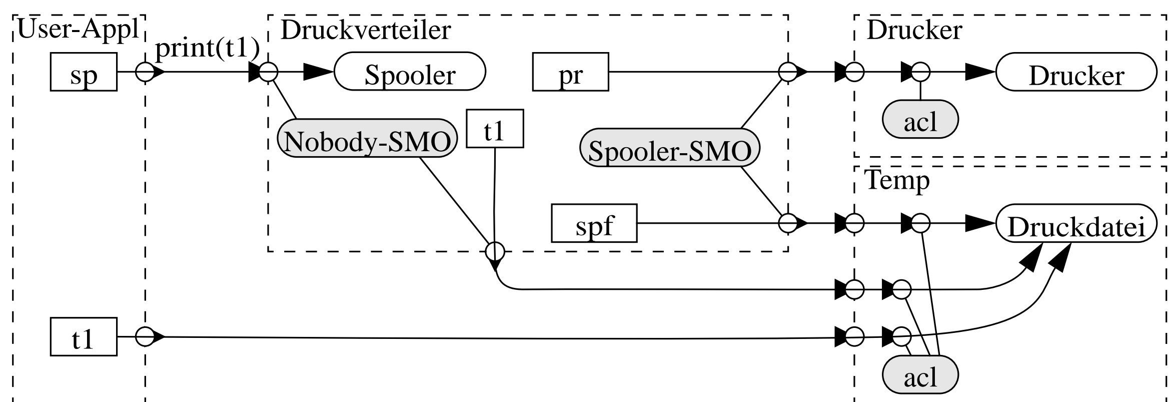

Abbildung 4.22 Versuch, eine Referenz unterzuschieben

In Abbildung 4.22 versucht die Benutzerapplikation durch den Aufruf `sp.print(t1)` dem Druckverteiler die Referenz auf das von ihr nicht ansprechbare Dateiobjekt (`t1`) unterzuschieben. Da jedoch im Gegensatz zu domänenbasierten Identitäten bei rollenbasierten Identitäten die Authentifizierung geerbt wird, greift der Druckverteiler auf die Textreferenz anonym zu und das Unterschieben schlägt fehl. Der Druckverteiler hat nun zwei Referenzen auf die gleiche temporäre Druckdatei: `t1`, die er von einer Benutzerapplikation erhalten hat, ohne Authentifizierung und `spf`, die er selbst generiert hat, mit Authentifizierung.

Damit das Problem untergeschobener Referenzen vollständig gelöst wird, muß disjunkte Interaktion vorausgesetzt werden. Anderenfalls, wenn beispielsweise die Referenz auf die zu drukkende Datei an den Drucker weitergegeben wird, geht die Information verloren, daß die Datei vom Benutzer stammt. Das `nobody-SMO` verschwindet und wird durch ein SMO des Druckers ersetzt (Abbildung 4.23). Der Drucker nutzt unabsichtlich seine möglicherweise hohen Rechte, um auf die vom Benutzer erhaltene Datei zuzugreifen. Es muß also sichergestellt sein, daß der Druckverteiler mit disjunkter Interaktion auskommt, sonst ist diese Konfiguration nicht brauchbar.

Zur Implementation von `Nobody-SMO` und `Spooler-SMO` könnte die Implementation von Grenz-SMOs aus Abschnitt 4.4.5 verwendet werden. Die beiden SMOs liefern verschiedene Identitätsinformationen: Das `Spooler-SMO` authentifiziert Aufrufe als Druckverteiler, das `Nobody-SMO` authentifiziert nicht. Ein Nachteil der Implementation ist jedoch, daß die disjunkte Interaktion nicht sichergestellt wird. Falls der Druckverteiler versehentlich in gewissen Situationen nicht-disjunkt arbeitet, entstehen Sicherheitsprobleme. Es scheint daher besser, die disjunkte Arbeitsweise durch SMOs zu garantieren, indem erzwungen wird, daß über eine SMO-behaftete Referenz lediglich Referenzen fließen dürfen, die das gleiche SMO angeheftet haben.

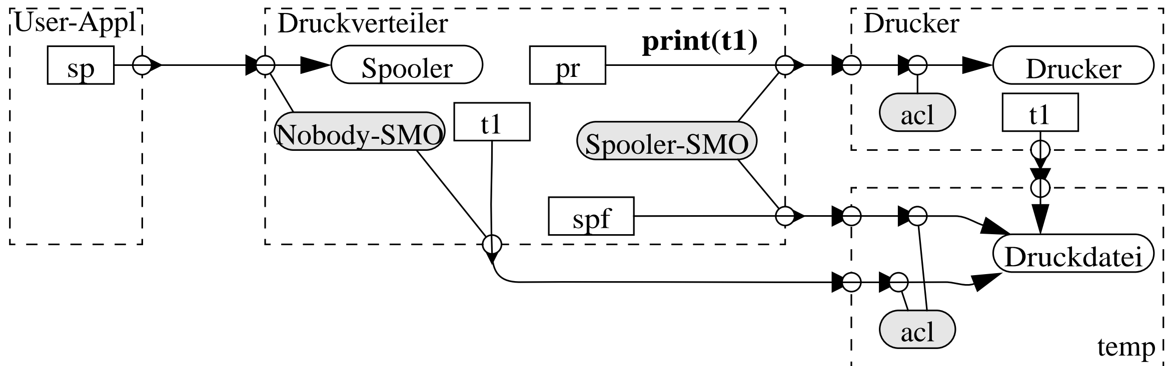

Abbildung 4.23 Sicherheitsprobleme bei nicht-disjunkter Interaktion

```
class DisjunctBorderPrincipalSMO extends SecurityMeta {
     ....
     void outgoingRef (ObjectRef outRef) {
          .....
               if (metaObjs[n] instanceof PrincipalSMORemovable) {
                    metaObjs[n].removeFrom(outRef);
                    if (metaObjs[n].isBorderObject())      // Grenz-SMO?
                         if (!this.equals(metaObjs[n]))    // dann muß es das
                              throw new SecException();    // gleiche sein
                         return;
               }
          .....
     }
     ....
}
```

Listing 4.11 Implementation eines disjunkten Grenz-Identitäts-SMOs

Listing 4.11 zeigt die Implementationsänderungen gegenüber einem normalen Grenz-SMO (Listing 4.10): Es wurde lediglich eine Überprüfung eingefügt, ob das Grenz-SMO, das von der Referenz zu entfernen ist, dasselbe ist, über das der Aufruf durchgeführt wird. Falls beide Grenz-SMOs verschieden sind, wird der Aufruf mit einer Sicherheitsausnahme abgebrochen. Diese Implementation stellt disjunkte Interaktion sicher.

4.5.4 Hierarchische Domänen

Eine zweite Möglichkeit zur Lösung des Problems stellen hierarchische Domänen dar. Mehrere Domänen sind dabei ineinander geschachtelt. Referenzen, die innerhalb einer Domäne (auch zwischen inneren Domänen) ausgetauscht werden, gelten dabei als vertrauenswürdiger, Referenzen, die von außerhalb kommen, gelten als weniger vertrauenswürdig. Dementsprechend wird die Authentifizierung gewählt: Aufrufe über Referenzen, die von außen kommen, werden

mit schwachen Identitäten (Identitäten mit wenig Rechten), Aufrufe über interne Referenzen mit starken Identitäten authentifiziert. Dabei gibt es keine Interaktionseinschränkungen, d.h. diese Lösung ist auch geeignet, falls nicht-disjunkte Interaktion nötig ist.

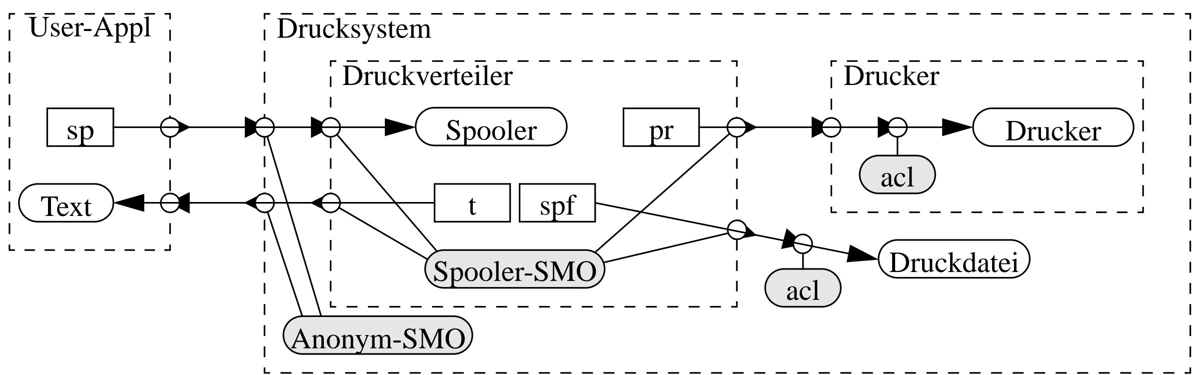

Abbildung 4.24 Druckverteiler mit hierarchischen Domänen

Abbildung 4.24 zeigt die Druckverteilerimplementation mit hierarchischen Domänen. Es besteht aus einer Drucksystem-Domäne, die das komplette Drucksystem enthält. Das Drucksystem ist wiederum unterteilt in eine Druckverteiler-Domäne und eine Drucker-Domäne. Die temporäre Druckdatei kann entweder auch in eine eigene Domäne plaziert werden oder wie in der Abbildung direkt in der Drucksystem-Domäne liegen.

Die entscheidenden SMOs in dieser Konfiguration sind einerseits das `Spooler-SMO`, das für abgehende Aufrufe als Authentifizierungsinformation "Druckverteiler" setzt und andererseits das `Anonym-SMO`, das Authentifizierungsinformation wieder löscht. Dies bedeutet: Aufrufe über die Referenz `spf` und `pr` werden mit "Druckverteiler" authentifiziert. Aufrufe über `t` werden nicht authentifiziert: das `Spooler-SMO` fügt dem Aufruf Authentifizierungsinformation hinzu, das `Anonym-SMO` nimmt sie wieder weg.

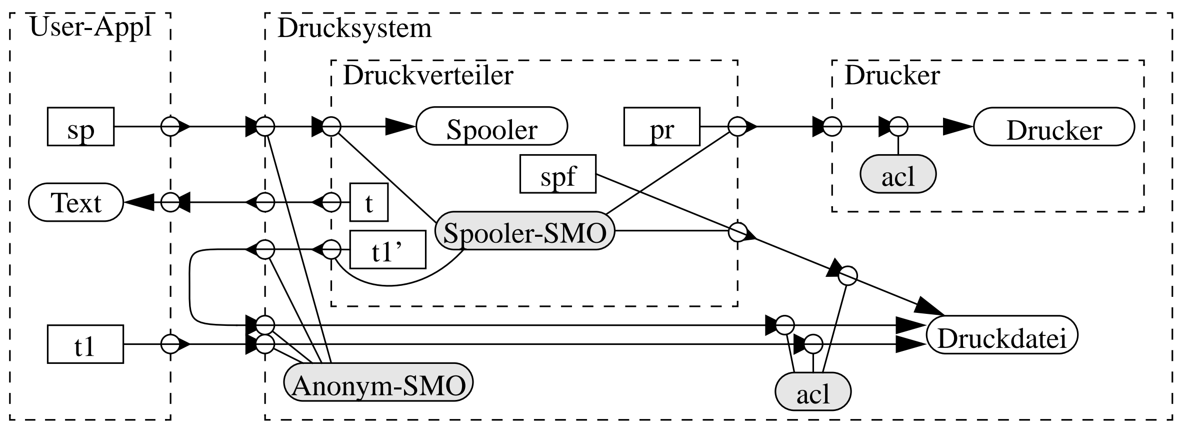

Abbildung 4.25 Unterschieben von Referenzen bei hierarchischen Domänen

Wenn der Benutzer die `print`-Methode des Druckverteilers aufruft (Abbildung 4.25) und dabei eine Referenz (`t1`) auf eine Druckdatei an den Druckverteiler übergibt, behält diese das `Anonym-SMO` angeheftet (`t1'`). Sie geht weiterhin aus dem Druckverteiler heraus (und dann

wieder hinein), d.h. Aufrufe werden nicht authentifiziert. Diese Referenz kann nun auch ohne Probleme an andere Drucksystemkomponenten gegeben werden. In keinem Fall werden Aufrufe authentifiziert.

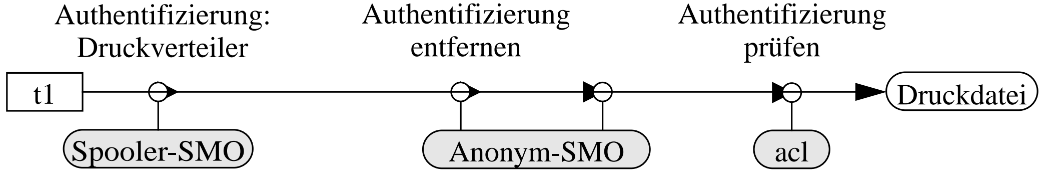

Abbildung 4.26 Die untergeschobene Referenz “ausgeklappt”

Abbildung 4.26 zeigt die untergeschobene Referenz `t1`. Bei Aufrufen über diese Referenz fügt das `Spooler-SMO` dem Aufruf Druckverteiler-Authentifizierungsinformation hinzu, das `Anonym-SMO` entfernt diese wieder (an der Stelle, wo es quell-angeheftet ist). Bei Weitergabe aus der Druckverteiler-Domäne wird zunächst das `Spooler-SMO` entfernt, danach das `Anonym-SMO`. Es kann durch Weitergabe dieser Referenz niemals eine Referenz entstehen, die nur das `Spooler-SMO`, nicht aber das `Anonym-SMO` angeheftet hat.

Die beiden SMOs sind gewöhnliche Grenz-SMOs (Abschnitt 4.4.5). Lediglich die Authentifizierungsroutine muß bei dem `Anonym-SMO` anders implementiert werden (Listing 4.12).

```
class AnonymousBorderPrincipalSMO extends BorderPrincipalSMO {
     void outgoingCall (...) {
          pList.removeAll();                // Identitätsinformation komplett
     }                                      // entfernen
}
```

Listing 4.12 Implementation eines Anonym-Grenz-Identitäts-SMOs

4.5.5 Explizite Identitäten

Eine dritte Möglichkeit, den Druckverteiler zu implementieren, stellen explizite Identitäten dar. Bei expliziten Identitäten werden neue Referenzen nicht automatisch mit Authentifizierung von Aufrufen versehen. Das jeweilige Programm muß explizit für Referenzen einstellen, daß und wie Aufrufe authentifiziert werden. Dies führt zu einer teilweisen Vermischung von Sicherheitsstrategie und Applikationssemantik und ist daher nur zu empfehlen, wenn generische Strategien wie in den vorigen Abschnitten nicht ausreichen oder die Applikation aus anderen Gründen direkt auf die Sicherheitsstrategie Einfluß nehmen möchte.

Man benötigt für eine Domäne ein Grenz-Identitäts-SMO, das keine Authentifizierung implementiert (`Nobody-SMO`) und nach Bedarf weitere Identitäts-SMOs, die Authentifizierung implementieren, aber nicht transitiv sind. Die Applikation selbst kann dann, falls nötig, ein solches Identitäts-SMO zusätzlich an eine Referenz binden. Das `Nobody-SMO` realisiert also nur die virtuelle Domäne, d.h. es entfernt Identitäts-SMOs von abgehenden Referenzen und wirkt transitiv. Die anderen Identitäts-SMOs implementieren nur Authentifizierung.

Die Realisierung sieht ähnlich wie bei rollenbasierten disjunkten Identitäten aus (Abbildung 4.27). Die Interaktion mit Benutzerapplikationen und die Lösung des Problems untergeschobener Referenzen ist intuitiv: Alle ankommenden Referenzen erhalten automatisch nur das `Nobody-SMO`, so daß alle Angriffe durch untergeschobene Referenzen fehlschlagen – auch bei nicht-disjunkter Interaktion.

Einziges Problem ist, daß an einigen Stellen explizit authentifiziert werden muß und dadurch Applikationssemantik und Sicherheitsstrategie gemischt wird. Man hat jedoch das Prinzip der sicheren Standardwerte: Falls man die explizite Authentifizierung vergißt, entstehen keine Sicherheitslücken, sondern eine Operation schlägt fehl.

Ein Beispiel für nötige explizite Authentifizierung ist Interaktion mit dem Drucker (Abbildung 4.27). Der Drucker könnte über eine Methode `getConf()` verfügen, die eine Referenz auf ein Druckerkonfigurationsobjekt liefert. Dieses Objekt ist ebenfalls mit einer Zugriffsliste geschützt. Der Druckverteiler speichert in der Abbildung die erhaltene Referenz in der Variablen `prc`. Da das `Spooler-SMO` sich nicht transitiv anheftet, ist die Referenz in der Form nicht brauchbar. Der Druckverteiler muß explizit das Spooler-SMO anheften, um genügend Rechte für Aufrufe über die Referenz zu erhalten.

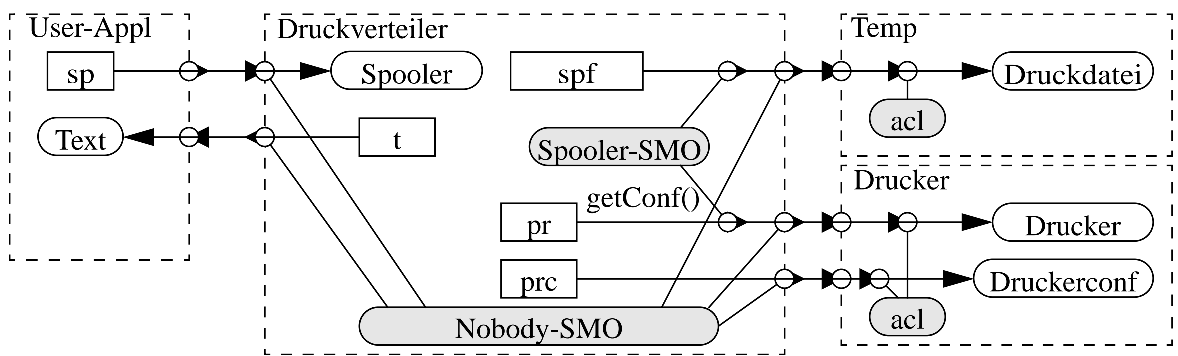

Abbildung 4.27 Explizite Identitäten

Das `Nobody-SMO` ist ein gewöhnliches Grenz-SMO, das `Spooler-SMO` ein gewöhnliches Identitäts-SMO.

Explizite Authentifizierung, d.h. das explizite Anheften von SMOs, muß nur einmal bei jeder neu erhaltenen Referenz erfolgen, die Authentifizierung für Aufrufe benötigt. Es ist daher wesentlich seltener nötig als bei expliziter Authentifizierung pro Aufruf.

4.5.6 Zusammenfassung: Rollenbasierte Identitäten

In den vorigen Abschnitten wurden drei Konfigurationsszenarien für die im Rahmen dieser Arbeit entwickelte Strategie *rollenbasierte Identitäten* dargestellt. Mit rollenbasierten Identitäten kann man bei Verwendung von Zugriffslisten das mit allen klassischen Strategien unlösbare Problem der untergeschobenen Referenzen lösen. Diese Strategie ist besonders für objektorientierte Systeme geeignet, da dort das Problem der untergeschobenen Referenzen wegen der hohen Dynamik verstärkt auftritt. Die Identität für die Authentifizierung wird bei rollenbasierten

Identitäten – ähnlich wie bei dem in Kapitel 2.5 vorgestellten Konzept der rollenbasierten Zugriffskontrolle – nicht an Benutzern festgemacht, sondern an der Rolle, die der Benutzer oder Applikationsteil für einen gewissen Ablauf hat. Das Konzept der rollenbasierten Zugriffskontrolle legt diese Rolle jedoch für einen kompletten Ablauf fest und verhält sich daher ähnlich wie die Strategie threadbasierte Identitäten: Es tritt ebenfalls das Problem der untergeschobenen Referenzen auf. Rollenbasierte Identitäten machen die Rolle an Referenzen fest: Ein Applikationsteil kann gleichzeitig in unterschiedlichen Rollen mit verschiedenen Partnern arbeiten. Die Implementation der SMOs für diese Strategie kann in vielen Fällen (bei allen drei oben gezeigten Szenarien) generisch erfolgen. Die Applikation muß lediglich bei expliziten Identitäten selbst Einfluß auf die Sicherheitsstrategie nehmen, und auch hier nur in seltenen Fällen. Sicherheitsstrategie und Applikationssemantik bleiben weitgehend getrennt.

4.6 Elementare Wertobjekte

Es gibt in den meisten objektorientierten Systemen zwei Arten von Werten, die als Parameter und Rückgabewerte von Methoden übergeben werden können: Objektreferenzen und elementare Wertobjekte. Elementare Wertobjekte sind Objekte, die nicht verändert werden können, wie dies in vielen Programmiersprachen bei Integerobjekten, Gleitkommazahlobjekten und Zeichenkettenobjekten der Fall ist[4]. Es taucht nun die Frage auf, ob die SMO-Referenzmethoden `incomingRef` und `outgoingRef` auch bei diesen elementaren Wertobjekten aufgerufen werden sollen, ob es also Sinn macht, diese Objekte möglicherweise mit speziellem Schutz zu versehen. Für die Trennung von Applikationsteilen (virtuelle Domänen, transitive Zugriffslisten) ist die Entscheidung nicht relevant, da über solche Objekte keine neuen Referenzen ausgetauscht werden können. Semantisch kann eine Übergabe von solch einem elementaren Wertobjekt an einen anderen Programmteil auch als Übergabe per Kopie betrachtet werden, so daß der neue Programmteil dann ein eigenes solches Objekt besitzt.

Bei elementaren Objekten ist meist nur der Wert als ganzes interessant, und man möchte einem anderen Programmteil entweder kompletten Zugriff oder gar keinen Zugriff darauf erlauben. Auch Revokation macht keinen Sinn, da der Kommunikationspartner einfach das Objekt vor der Revokation auslesen kann und dadurch die Information nicht mehr revoziert werden kann. Es könnte lediglich sinnvoll sein, die Übergabe zu verhindern, um die in den Wertobjekten enthaltene Information zurückzuhalten. Beispielsweise könnte man die Rückgabe von Ergebnissen insgesamt verhindern, um nur einen Informationsfluß in das Programm hinein, nicht aber in die umgekehrte Richtung zu erlauben (dieses Problem wird in Abschnitt 4.10 genauer betrachtet).

---

4. In den meisten objektorientierten Sprachen können auch elementare Typen wie “int”-Werte als Objekte betrachtet werden. Es gibt dann für jeden Wert (1,2,3,...) genau ein int-Objekt, das nicht-modifizierbar ist. Beispielsweise der Ausdruck “1+2” liefert dann eine Referenz auf das nur einmal im System vorhandene Objekt “3” zurück. Diese Sichtweise wird beispielsweise von Smalltalk [GoR89] verwendet. In Java kann man diese Sichtweise nur eingeschränkt verwenden, da z.B. die Typkompatibilität zu generischen Zeigern (“Object”) fehlt.

Dies kann man jedoch auch auf diese Weise nicht verhindern, da über sogenannte Zeitkanäle weiterhin Information fließen könnte. Daher erscheint es nicht sinnvoll, elementare Wertobjekte durch den SMO-Übergabemechanismus laufen zu lassen.

4.7 Reduktion von SMO Ketten

Eine Problematik, die bisher nicht betrachtet wurde, sind SMO-Ketten. Wenn Referenzen öfter zwischen Applikationsteilen transferiert werden, können sich SMO-Ketten bilden, d.h. je öfter sie transferiert werden, desto mehr SMOs werden an die Referenz angeheftet. Dies bewirkt zwar keine Sicherheitsprobleme oder Zugriffsprobleme, die langen Ketten können sich aber bei einer Implementierung des Modells auf die Effizienz auswirken. Während das allgemeine SMO-Ketten-Problem schwer lösbar ist, kann man für einige oft auftretende Fälle eine Lösung finden: Man kann unter gewissen Bedingungen ein SMO, das mehrfach an einer Referenz hängt, an manchen Stellen ersatzlos entfernen. Wir werden hier nur die ersatzlose Entfernung einer Anheftung von mehrfach auftretenden SMOs betrachten. Abbildung 4.28 zeigt ein Beispiel: `SMO1` ist zweimal an die Referenz `o` angeheftet. Eine der beiden Anheftungen kann unter bestimmten Bedingungen entfernt werden. `SMO2` ist zwar auch zweifach angeheftet, allerdings in verschiedenen Modi, daher kann keine dieser Anheftungen entfernt werden.

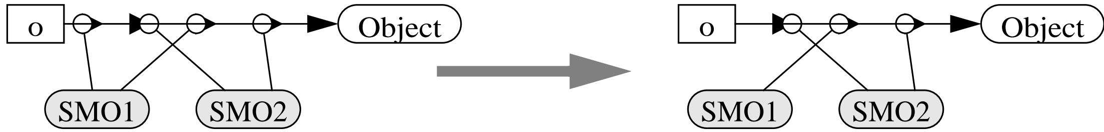

Abbildung 4.28 Beispiel: Entfernung mehrfach angehefteter SMOs

Bei der SMO-Ketten-Reduktion muß man verschiedene Arten von SMOs unterscheiden. Wir betrachten hier nur transitive SMOs, da im wesentlichen durch diese solche Ketten entstehen können. Außerdem dürfen SMOs nicht (z.B. mit Metainteraktion) durch den Benutzer entfernt werden, da sonst die Semantik verändert wird: Ein doppelt angeheftetes SMO ist bei einmaligem Entfernen durch den Benutzer noch ein weiteres Mal angeheftet. Wenn man jedoch vorher durch Kettenreduktion eine der beiden Anheftungen entfernt, ist es nach einmaligem Entfernen durch den Benutzer völlig von der Referenz entfernt, wodurch sich die Semantik ändert.

Grenz-Identitäts-SMOs kann man nicht wegoptimieren, da man sonst die virtuelle Domäneneigenschaft verliert bzw. sonst die rollenbasieren Identitäten nicht funktionieren. Grenz-Identitäts-SMOs sind allerdings unproblematisch, da durch diese nicht beliebig lange Ketten entstehen können. Die maximale Kettenlänge ist das Doppelte der maximalen Schachtelungstiefe von Domänen, da mit Grenz-SMOs nur äußere Schleifen entstehen können (Abbildung 4.29). Wir betrachten auch keine Reduktion von SMOs, die sowohl innerhalb als auch außerhalb einer virtuellen Domäne angeheftet sind. Ein Grenz-SMO bildet also eine Grenze, über die hinweg wir keine Optimierung vornehmen können.

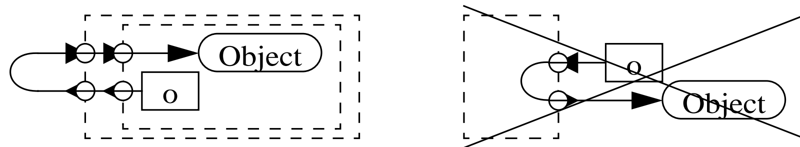

Abbildung 4.29 Grenz-SMO-Ketten: Es können nur äußere Schleifen entstehen

Bei der Optimierung müssen wir berücksichtigen, daß sie jeweils für die Referenz selbst sowie für alle transitiv durch die Referenz erzeugten Referenzen semantikerhaltend sein muß. Transitiv erzeugte Referenzen können entweder in die gleiche Richtung wie die Ursprungsreferenz (dies ist der Fall für Ergebniswerte) oder in die Gegenrichtung zeigen (das ist der Fall bei Übergabeparametern). Bei ersterem Fall müssen wir nichts berücksichtigen: Wenn die Optimierung für eine beliebige Originalreferenz semantikerhaltend ist, ist sie dies automatisch auch für alle Rückgabewerte. Für Referenzen in Gegenrichtung müssen wir jedoch zeigen, daß die Optimierung auch für diese gültig ist. Dazu unterscheiden wir verschiedene Arten von angehefteten transitiven SMOs. Jede SMO-Anheftungs-Art hat ein Inverses. Dieses Inverse stellt das in Gegenrichtung angeheftete SMO dar. D.h. wenn ein SMO in Quellanheftung die Art “x+” darstellt, ist dasselbe SMO in Zielrichtung angeheftet automatisch “x-”.

Wir unterscheiden:

- negative Identitäts-SMO-Anheftung (Id_-)

  Das sind SMOs, die Identitätsinformation entfernen oder in entgegengesetzte Richtung (d.h. mit Id_+-Anheftung) Identitätsinformation zur Verfügung stellen, jedoch nicht auf die bereits vorhandene Identitätsinformation zugreifen.
  Zu dieser Kategorie gehören ziel-angeheftete Identitäts-SMOs und quell-angeheftete Anonym-Identitäts-SMOs.

- positive Identitäts-SMO-Anheftung (Id_+)

  Das sind Identitäts-SMOs, die Identitätsinformation zur Verfügung stellen oder in entgegengesetzte Richtung Identitätsinformation entfernen, jedoch nicht auf die bereits vorhandene Identitätsinformation zugreifen.
  Zu dieser Kategorie gehören quell-angeheftete Identitäts-SMOs und ziel-angeheftete Anonym-Identitäts-SMOs.

- identitäts-benutzende SMO-Anheftung ($Id_>$ und $Id_<$)

  Das sind SMOs, die bei zusätzlich zur Verfügung gestellter Identitätsinformation geringere Restriktionen implementieren, selbst aber keine Identitätsinformation zur Verfügung stellen oder entfernen. Wir benötigen auch hier positive und negative SMO-Anheftungen, obwohl beide die gleichen Eigenschaften haben, da solche SMOs meist in verschiedene Richtungen verschiedene Restriktionen implementieren.
  Zu dieser Kategorie gehören Zugriffslisten-SMOs.

- identitäts-neutrale SMO-Anheftung (Id_{N+} und Id_{N-})

  Das sind SMOs, die weder Identitätsinformation zur Verfügung stellen, noch Identitätsinformation verwenden. Zu dieser Klasse gehören beispielsweise einfache Capability-SMOs.

Nun wollen wir überlegen, unter welchen Bedingungen wir SMO-Anheftungen ersatzlos entfernen können. Da wir von transitiven SMOs ausgehen, muß die Optimierung jeweils für die SMO-Kette und für die inverse SMO-Kette gelten. In Abbildung 4.30 ist eine Referenz (o) und eine zugehörige inverse Referenz (op) abgebildet, die durch Parameterübergabe über o entstehen kann. An einer durch Parameterübergabe bzw. Ergebnisrückgabe entstehenden Referenz können weitere SMOs hängen, jedoch nur vor oder nach der durch die Transitivität entstehenden Kette, nicht jedoch innerhalb der Kette, da die SMOs sequentiell angehängt werden. Wenn sich nun das lokale Verhalten des transitiven Kettenstücks nicht ändert, kann sich auch keine globale Änderung ergeben. Wir brauchen daher immer nur lokale SMO-Teilstücke zu betrachten.

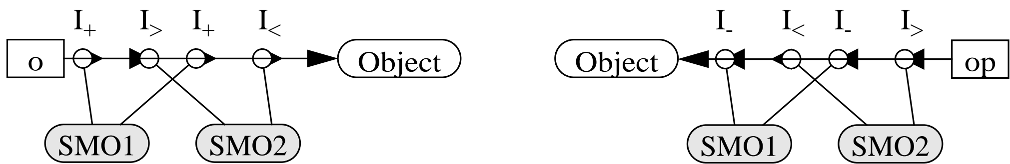

Abbildung 4.30 Beispiel: Referenz und inverse Referenz

Falls ein SMO zweimal direkt hintereinander auf die gleiche Art angeheftet ist, kann eine der Anheftungen ersatzlos entfernt werden, da die hier betrachteten SMOs entweder den Zustand (Identitätsinformation) nicht ändern oder ihn ohne Berücksichtigung der bereits vorhandenen Identitätsinformation ändern, d.h. zweimalige Änderung ist gleich einmalige Änderung (Tab. 4.1).

Optimierung	Optimierung (invers)
$Id_{SMO1\,>} \bullet Id_{SMO1\,>} = Id_{SMO1\,>}$	$Id_{SMO1\,<} \bullet Id_{SMO1\,<} = Id_{SMO1\,<}$
$Id_{SMO1\,+} \bullet I_{SMO1\,+} = I_{SMO1\,+}$	$Id_{SMO1\,-} \bullet Id_{SMO1\,-} = Id_{SMO1\,-}$
$Id_{SMO1\,N+} \bullet Id_{SMO1\,N+} = Id_{SMO1\,N+}$	$Id_{SMO1\,N-} \bullet Id_{SMO1\,N-} = Id_{SMO1\,N-}$

Tabelle 4.1 Optimierung hintereinander hängender SMOs

Falls ein SMO jedoch nicht zweimal direkt hintereinander hängt, sondern ein anderes SMO dazwischen hängt, muß man möglicherweise angehängte SMOs vertauschen, um obige Optimierung vornehmen zu können. Folgende Vertauschungen können vorgenommen werden:

Vertauschung	Vertauschung (invers)
$Id_{SMO1\,+} \bullet Id_{SMO2\,+}$ $= Id_{SMO2\,+} \bullet Id_{SMO1\,+}$	$Id_{SMO2\,-} \bullet Id_{SMO1\,-}$ $= Id_{SMO1\,-} \bullet Id_{SMO2\,-}$
$Id_{SMO1\,N} \bullet Id_{SMO2\,+}$ $= Id_{SMO2\,+} \bullet Id_{SMO1\,N}$	$Id_{SMO2\,-} \bullet Id_{SMO1\,N}$ $= Id_{SMO1\,N} \bullet Id_{SMO2\,-}$
$Id_{SMO1\,<,>,N} \bullet Id_{SMO2\,<,>,N}$ $= Id_{SMO2\,<,>,N} \bullet Id_{SMO1\,<,>,N}$	

Tabelle 4.2 Vertauschung von SMOs

Dies sind die wichtigsten möglichen Vertauschungen. Wir werden im Kapitel 5.1 durch ein formales Modell beweisen, daß man diese Vertauschungen tatsächlich vornehmen kann und noch einige zusätzliche Reduktionsmöglichkeiten zeigen.

Wir wollen nun ein Beispiel betrachten, bei dem die Optimierung wichtig ist. Abbildung 4.31 zeigt eine Situation, in der dupliziert angeheftete SMOs entstehen: Die linke Applikation verfügt über eine Referenz auf ein Objekt in der rechten Applikation (`o`), an der zwei Zugriffslisten-SMOs, die nur Aufrufe mit gewissen Identitätsinformation zulassen, hängen. Über diese Referenz wird nun eine Methode aufgerufen, wobei die Referenz selbst als Parameter übergeben wird (wir betrachten nicht die Bereitstellung von eventuell benötigter Identitätsinformation, die z.B. durch virtuelle Domänen zur Verfügung gestellt werden könnte). An der entstehenden Referenz kann noch nicht optimiert werden, da die beiden Zugriffslisten-SMOs nicht zweimal auf die gleiche Weise angeheftet sind. Wenn diese Referenz nun jedoch wieder über dieselbe Referenz zurückgegeben wird (`return o`), werden die Zugriffslisten-SMOs noch einmal angeheftet.

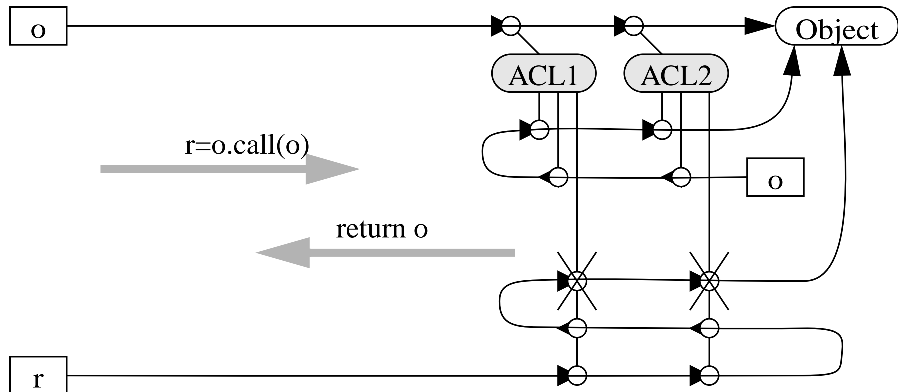

Abbildung 4.31 Beispiel: Reduktion mehrfach angehängter SMOs

Nun kann optimiert werden: Die Zugriffslisten-SMOs sind $I_{>}$-SMOs. D.h. die Referenz `r` besteht im hinteren Teil aus:

$Id_{ACL1\,>} \bullet Id_{ACL2\,>} \bullet Id_{ACL2\,<} \bullet Id_{ACL1\,<} \bullet Id_{ACL1\,>} \bullet Id_{ACL2\,>}$

“>” und “<” kann man beliebig vertauschen, d.h. man kann optimieren zu:

$Id_{ACL1\,>} \bullet Id_{ACL1\,>} \bullet Id_{ACL2\,>} \bullet Id_{ACL2\,>} \bullet Id_{ACL2\,<} \bullet Id_{ACL1\,<}$

Die doppelten Anheftungen kann man dann weglassen, d.h. übrig bleibt:

$Id_{ACL1\,>} \bullet Id_{ACL2\,>} \bullet Id_{ACL2\,<} \bullet Id_{ACL1\,<}$

4.8 Meta-Hierarchien

Wenn man die SMOs als zugehörig zur Meta-Ebene betrachtet, taucht die Frage auf, ob man den Zugriff auf SMOs ebenfalls durch SMOs einschränken kann. Abbildung 4.32 zeigt denkbare SMO-Verschachtelungen:

- ein SMO angeheftet an eine SMO-Referenz
  Dadurch ist man in der Lage, Zugriffe auf SMOs zu beschränken. Man kann beispielsweise eine Referenz auf ein Identitäts-SMO weitergeben, die einem lediglich das Anheften des SMOs an bestimmte Objektreferenzen erlaubt.
- ein SMO angeheftet an eine Referenz auf sich selbst
  Wenn man den Zugriff auf das SMO mit der gleichen Strategie beschränken möchte, mit der man auch andere Objektreferenzen schützen möchte, ist diese Möglichkeit durchaus sinnvoll. Wenn beispielsweise das SMO begrenzte Gültigkeit implementiert, könnte man festlegen, daß nach Ablauf der Gültigkeit das SMO nicht mehr angeheftet werden kann (es wird beim Anheftungsversuch eine Ausnahme erzeugt).
- ein SMO angeheftet an eine Anheftung
  Damit könnte man die Ereignisse, die das SMO bekommt, einschränken. Da die Anheftung jedoch nicht einfach eine Objektreferenz ist, müßte man eine neue SMO-Semantik für diesen Fall definieren. Dieser Fall wird daher nicht zugelassen.

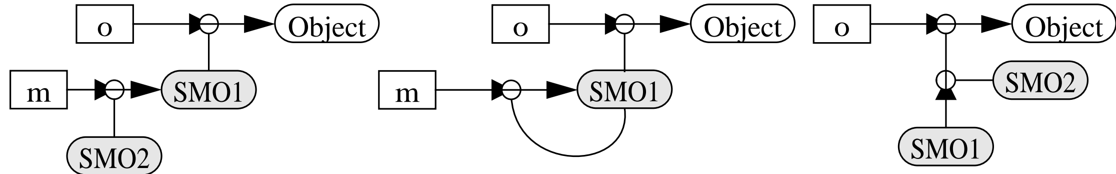

Abbildung 4.32 Denkbare Meta-Hierarchien

4.9 Gesamtes System

Bisher haben wir nur ein bereits laufendes System untersucht und Sicherheitsprobleme zwischen verschiedenen Objekten eines solchen Systems betrachtet. Um jedoch überhaupt eine initiale Kontaktaufnahme zwischen verschiedenen Programmstücken auf verschiedenen Rechnern zu ermöglichen, muß es global erreichbare Objekte im System geben. In objektorientierten Systemen wird dies oft durch einen Nameserver realisiert.[5] Dieses Nameserver-Objekt kann kontaktiert werden, und Objekte können unter gewissen Namen eingetragen werden. Die Nameserver müssen eine gewisse Lokalität haben, d.h. lokale Referenzen implementieren, die einen Ersatz für globale Variablen darstellen, und es muß bei manchen Nameservereinträgen Schutz gegen unbefugten Zugriff geben. Dies wollen wir jedoch nicht genauer betrachten – mit SMOs können problemlos Zugriffslisten erstellt werden, mit denen der Nameserver geschützt werden kann.

Man könnte nun einer Applikation beim Start eine Referenz auf den Nameserver mitgeben. Die Applikation müßte dann die Nameserver-Referenz speichern. Diese Lösung ist jedoch unbefriedigend, da man so keine Möglichkeit hat, von beliebigen Stellen im Programm auf den Nameserver zuzugreifen – die Nameserver-Referenz müßte dann allen Methoden, die die Referenz benötigen, als Parameter mitgegeben werden bzw. in Objekte eingetragen werden. Besser wäre es daher, wenn jedes Objekt implizit über eine Nameserverreferenz verfügt. Beim Start eines Programmes wird in das initiale Objekt dann automatisch diese Referenz eingetragen. Wenn neue Objekte generiert werden, wird die Nameserverreferenz normalerweise vererbt. Es kann aber auch Situationen geben, in denen man diese Referenz selbst festlegen möchte, daher kann man bei der Objekterzeugung auch eine andere Referenz dort eintragen (beispielsweise eine mit Capability-SMO geschützte Nameserverreferenz). Wenn ein globaler Name aufgelöst werden soll (auch beispielsweise beim Ansprechen von globalen Variablen), wird automatisch über die objektinterne Nameserverreferenz zugegriffen – eine andere Möglichkeit für globale Zugriffe gibt es nicht. Abbildung 4.33 zeigt drei Objekte mit Instanzvariablen. Die Variable `ns` ist keine Instanzvariable, sondern die in jedem Objekt implizit vorhandene Nameserverreferenz.

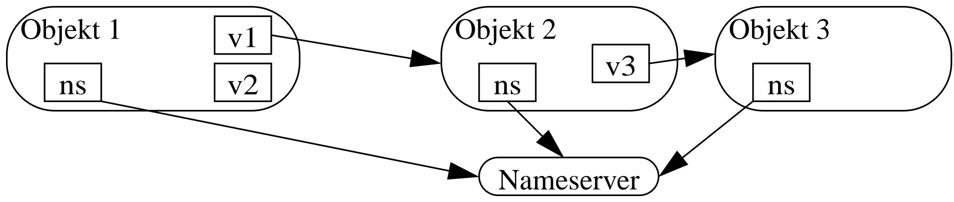

Abbildung 4.33 Nameserver

5. Dies kann auch die Kommunikation innerhalb eines Programmes betreffen. An Stellen, wo man in herkömmlichen Systemen globale Variablen verwendet, wird in unserem System ebenfalls der Namensdienst benutzt.

4.10 Realisierung verschiedener Modelle

In diesem Abschnitt soll überlegt werden, welche klassischen Semantiken man mit SMOs implementieren kann.

Wie in den vorigen Abschnitten gezeigt, lassen sich mit SMOs alle Arten von Capabilities und Zugriffslisten realisieren:

- Zeitlich begrenzte Capabilities
- Revokation von Capabilities
- Einschränkung des Zugriffs auf gewisse Methoden
- Einschränkung des Zugriffs auf bestimmte Benutzer bzw. Identitäten

Zusätzlich lassen sich eine Reihe von erweiterten Sicherheitsproblemen lösen. Wir wollen hier verschiedene Sicherheitsprobleme, die von klassischen Systemen gelöst werden, bzw. Sicherheitsforderungen diskutieren und für einige der Probleme eine Lösung mit SMOs präsentieren.

- Gegenseitiges Mißtrauen (Hydra, Abschnitt 3.1.1)
  Es sollte möglich sein, beim Methodenaufruf die übergebenen Referenzen mit Schutz zu versehen. Dies ist mit SMOs durch Einsatz konfigurierbarer Capabilities möglich.
- Konstanten-Problem (Hydra, Abschnitt 3.1.1)
  Um zu verhindern, daß ein Objekt unbefugt Informationen speichert, sollte man festlegen können, daß ein Methodenaufruf keine Daten schreiben kann.
  Dies ist mit SMOs nicht möglich. Die Forderung widerspricht der objektorientierten Abstraktion. Selbst bei Operationen die nur Werte lesen, kann es nötig sein, Daten im Objekt zu verändern – beispielsweise könnte das Objekt einen Cache für Lesezugriffe verwalten.
- Weitergabe-Problem (Hydra, Abschnitt 3.1.1)
  Um die Verbreitung einer Capability unter Kontrolle zu haben, sollte man festlegen können, ob sie weitergegeben werden kann.
  Diese Forderung schützt in dieser Form nur gegen versehentliche Weitergabe. Wenn jemand eine Capability besitzt, die er nicht weitergeben kann, kann er statt dessen selbst ein Objekt erzeugen, das als Stellvertreter fungiert und Methodenaufrufe weiterleitet. Gegen die versehentliche Herausgabe von Objektreferenzen kann man sich mit transitiven SMOs schützen.
- Speicherungs-Problem (Hydra, Abschnitt 3.1.1)
  Bei einer Capability sollte man festlegen können, daß sie nicht gespeichert werden kann. Dadurch kann man bei Übergabe an eine Funktion sicher sein, daß die aufgerufene Funktion nach Beendigung nicht weiterhin auf sie zugreifen kann.
  Dieses Problem kann man mit SMOs sogar generisch lösen. Wenn man eine Referenz auf ein Objekt in einem nicht-vertrauenswürdigen Applikationsteil besitzt, kann man an diese Referenz ein transitives SMO anheften, das Parameter gegen Zugriffe nach Beendigung der Methode schützt. Abbildung 4.34 zeigt ein Beispiel: Die Referenz `o` ist mit einem `nosaveSMO` geschützt. Wenn eine Methode mit Parametern über die Referenz aufgeru-

fen wird, erzeugt das `nosaveSMO` ein weiteres SMO, das Referenzen ungültig machen kann, und heftet dieses an alle Parameter. Sobald der Methodenaufruf terminiert, werden die Parameterreferenzen dieses Aufrufs ungültig gemacht. Da das `nosaveSMO` transitiv ist, erhält man die Semantik, daß nach Beendigung aller Aufrufe der rechte Applikationsteil keine gültigen, durch den Aufruf entstandenen Referenzen mehr auf den linken besitzt.

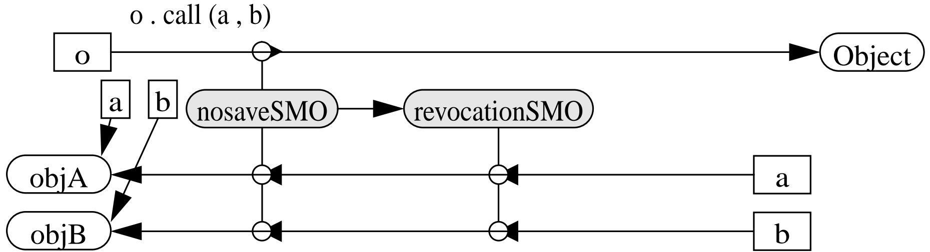

Abbildung 4.34 Schutz gegen Referenzspeicherung

- Einschluß (Confinement) (Hydra, Abschnitt 3.1.1)
  Bei einem Funktionsaufruf sollte man verhindern können, daß die Informationen, die man an eine Funktion übergibt, nach außen dringen können.
  Dies kann man mit SMOs durch Verwendung eines auf die Objektorientierung angepaßten Programmierparadigmas erreichen. Dazu instantiiert man selbst die Klassen, die die einzuschließenden Operationen implementieren, und gibt den dabei entstehenden Objekten keine implizite Nameserverreferenz. Die Objekte können dann nur untereinander und mit Objekten, deren Referenz man ihnen gibt, kommunizieren. Abbildung 4.35 zeigt ein Beispiel: `Objekt 1` wurde ohne Nameserverreferenz instantiiert. Man kann ohne Einschränkungen mit dem Objekt interagieren, das Objekt kann auch weitere Objekte erzeugen, es hat jedoch keine Möglichkeit mit anderen Objekten, die außerhalb dieser Menge von Objekten liegen, zu kommunizieren und kann daher erhaltene Informationen nicht außerhalb der Anwendung ablegen.
  Zusätzlich zum strengen Einschluß von Hydra kann man mit SMOs auch Einschluß mit kontrollierten Ausnahmen implementieren, indem man den neuen Objekten zwar eine implizite Nameserverreferenz gibt, diese aber mit einem SMO schützt. Das SMO wird nun über alle Aufrufe, die den Einschluß verletzen würden, unterrichtet und kann bei jedem einzelnen prüfen, ob dieser zulässig ist.

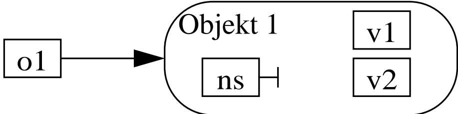

Abbildung 4.35 Lösung des Einschluß-Problems

- Initialisierungsproblem (Hydra, Abschnitt 3.1.1)
  Wenn man ein neues Objekt generiert, sollte man festlegen können, daß man nach der Initialisierung als einziger eine Referenz auf dieses besitzt. Dies kann man genau wie das

Confinement-Problem lösen: Wenn man bei der Objekterzeugung angibt, daß das erzeugte Objekt keine Nameserverreferenz besitzt, kann es die Referenz auf sich selbst nirgends eintragen. Man ist dann sicher, daß man nur selbst eine Referenz auf das Objekt besitzt.

- Informationsflußkontrolle, *-Eigenschaft
  Ein Methodenaufruf transferiert normalerweise Informationen in zwei Richtungen: Der Aufruf selbst übermittelt Informationen vom Aufrufer zum Aufgerufenen (Parameter, aufgerufene Methode), die Ergebnisrückgabe überträgt Informationen in die Gegenrichtung (Rückgabewert, Dauer des Aufrufes). Es gibt keine Möglichkeit, transparente, objektorientierte Programmierung zuzulassen, aber dabei den Informationsfluß nur in eine Richtung zu erlauben. Eine Möglichkeit, überhaupt Informationsfluß mit objektorientierter Programmierung zu beschränken, ist die Verwendung von nicht-zurückkehrenden Methoden. Der Aufrufer führt eine Methode aus und übergibt ihr Parameter (keine beliebigen Objektreferenzen, sondern nur Referenzen auf elementare, konstante Wertobjekte). Diese Methode wird dann ausgeführt, über den Ergebniswert bzw. die Beendigung erhält der Aufrufer keine Information. Dies ist mit SMOs implementierbar: Die Referenz auf ein Objekt, von dem keine Information zurückkommen soll, wird mit einem SMO geschützt. Beim Aufruf verhindert das SMO die Übergabe von Objektreferenzen. Bei der Rückkehr des Aufrufes terminiert das SMO den Thread, ohne den Aufrufer zu benachrichtigen.

- Delegationsproblem I (Spring, Abschnitt 3.1.3)
  Falls man anderen Teilnehmern Zugriff auf ein bestimmtes, durch eine Zugriffsliste geschütztes Objekt geben möchte, auf das man nur durch Verwendung seiner Identität Zugriff hat, sollte man dies explizit ermöglichen können.
  Dies kann man mit SMOs folgendermaßen implementieren: Man hängt an diese Referenz ein Identitäts-SMO, das die nötige Identität zur Verfügung stellt und sich beim Überschreiten einer Domänengrenze nicht entfernt. Falls nötig, kann das SMO zusätzlich Restriktionen (zeitliche Beschränkung, etc.) implementieren. Diese Referenz kann man nun jemandem übergeben, der dann genau auf diese Referenz mit der delegierten Identität zugreifen kann.

- Delegationsproblem II (DCE, DCOM, Abschnitt 3.2.2, 3.2.3)
  Beim Aufruf einer Methode eines anderen Objektes sollte man einen Teil oder alle seine Rechte delegieren können, so daß das Zielobjekt unter der Aufruferidentität arbeiten kann.
  Implizite Delegation wird von SMOs nicht unterstützt, da dies bei objektorientierter Programmierung wenig sinnvoll ist. Wenn mit einem Aktivitätsträger eine Methode mit Delegation aktiviert wird, behält der Aktivitätsträger die Rechte und kann mit den Rechten des Aufrufers weiteragieren. Dies ist einerseits gefährlich, weil man keine Kontrolle darüber hat, was das Zielobjekt mit diesen Rechten tut. Andererseits kann man dann im Zielobjekt keine Aufgabenteilung durchführen. Wenn das Zielobjekt beispielsweise einen internen Aktivitätsträger besitzt, an den die eigentlichen Aufgaben delegiert werden, agiert dieser nicht mit den delegierten Rechten. Daher scheint der Aktivitätsträger nicht die richtige Abstraktion für Delegation in objektorientierten Systemen zu sein.
  Explizite Delegation kann mit SMOs realisiert werden, indem beispielsweise der Aufrufer

dem Aufgerufenen ein Identitäts-SMO als Parameter übergibt, das dieser dann für Aufrufe verwenden kann. Dieses Identitäts-SMO kann natürlich eingeschränkt werden: Der Aufrufer kann beispielsweise festlegen, daß das Zielobjekt nur einen einzigen, festgelegten Aufruf authentifizieren kann.

- Proxies (Abschnitt 3.2.5)
  Proxies erlauben den Zugriff auf eine Objektreferenz nur gewissen Identitäten. Falls der Zugriff gestattet wird, wird er unter einer anderen Identität durchgeführt.
  Dies kann durch zwei SMOs an der Referenz implementiert werden: Ein Zugriffskontrolllisten-SMO, das den Zugriff beschränkt, und dahinter ein Identitäts-SMO, das die Identität zur Verfügung stellt.
- Gruppen, Rollen, verschiedene Identitäten
  Es sollte eine Möglichkeit geben, Identitäten zu Gruppen oder Rollen zusammenzufassen und verschiedene Identitäten zu verwenden.
  SMOs unterstützen mehrere Identitäten, es können auch Identitäten zu Gruppen oder Rollen zusammengefaßt werden. Da dies wie in klassischen Systemen erfolgen kann, wird dies hier nicht näher betrachtet.

4.11 Vergleich

Während im vorigen Abschnitt die Fähigkeit der SMOs unter Beweis gestellt wurde, alle wichtigen, von klassischen Systemen realisierten Sicherheitsmechanismen nachzubilden, soll nun ein Vergleich von klassischen Systemen und dem SMO-Sicherheitsmodell anhand der Kriterien aus Abschnitt 2.1 erfolgen. Wir vergleichen das SMO-Modell mit der Summe der untersuchten, klassischen Systeme. Ein Kriterium wird von klassischen Systemen dann als erfüllt angesehen, wenn eines der klassischen Systeme gemäß der Untersuchung in Abschnitt 3.3 das Kriterium erfüllt. Die Tabelle 4.3 zeigt den Vergleich. Man muß berücksichtigen, daß diese Tabelle nur eine eng umgrenzte Sicht widerspiegeln kann, sie kann und soll keinesfalls einen kompletten Vergleich aller Fähigkeiten der untersuchten Systeme und der SMOs beinhalten.

	klassische Systeme	Sicherheitsmetaobjekte
Einfachheit des Designs	+	+
Sichere Standardwerte	+	+ (ohne Objektreferenz ist kein Zugriff möglich)
Sichere Standardwerte (bei objektorientierter Programmierung)	-	+ (durch Transitivität ist automatischer, restriktiver Schutz von neu etablierten Referenzen möglich)

Tabelle 4.3 Vergleich klassische Systeme – Sicherheitsmetaobjekte

	klassische Systeme	Sicherheitsmetaobjekte
Komplette Überwachung	+	+
Komplette Überwachung (bei objektorientierter Programmierung)	(+) nur bei Domänen	+ (durch Transitivität)
Teilung der Rechte	-	+ (durch Zugriffslisten, die nur mehreren Identitäten gemeinsam Zugriff erlauben)
Kleinstmögliches Zugriffsrecht	+	+ (Schutz feingranular konfigurierbar)
Kleinstmögliches Zugriffsrecht (bei objektorientierter Programmierung)	-	+ (Schutz mittels Transitivität feingranular konfigurierbar)

Tabelle 4.3 Vergleich klassische Systeme – Sicherheitsmetaobjekte

4.12 Problemlösungen

Um einen Eindruck zu vermitteln, wie man Standardsicherheitssituationen mit SMOs bewältigt, wird in diesem Abschnitt zusammengefaßt, welche Mechanismen der SMOs für welche Problemklassen am besten geeignet sind. Die Unterschiede in der Vorgehensweise gegenüber klassischen Systemen ergeben sich vor allem bei Problemen mit komplexen Datenstrukturen, d.h. mit vielen Objekten, bei denen klassische Systeme alle Objekte einzeln schützen müssen, SMOs jedoch transitive Strategien verwenden können. Tabelle 4.4 zeigt einen Überblick über die verschiedenen Problemklassen.

Problemklasse	klassische Systeme	Sicherheitsmetaobjekte
Initiale Kontaktaufnahme zwischen Programmteilen	Zugriffslisten	Zugriffslisten
Schutz von Einzelobjekten gegen Aufrufe von gewissen Methoden	Capabilities	Capabilities
Schutz von Einzelobjekten gegen Aufrufe von nicht-autorisierten Benutzern	Zugriffskontrolllisten	Zugriffskontrolllisten

Tabelle 4.4 Problemlösungen mit klassischen Systemen im Vergleich zu SMOs

Problemklasse	klassische Systeme	Sicherheitsmetaobjekte
Schutz von komplexen Datenstrukturen (Listen, hierarchische Strukturen)	expliziter Schutz jedes einzelnen Objektes nötig	transitive Capabilities, transitive Zugriffskontrolllisten
Geschützte Kommunikation mit vielen Objekten eines Kommunikationspartners	Zugriffskontrolllisten auf Domänenbasis	transitive Zugriffskontrolllisten
Kommunikation mit verschiedenen Partnern in unterschiedlichen Rollen	Explizite Festlegung der Aufruferidentität pro Aufruf	rollenbasierte Identitäten
Einschluß (Confinement) von Information	Restriktion der Zielobjekte durch spezielle Zusatzmechanismen	Instantiierung der einzuschließenden Objekte ohne Nameserverreferenz, dann: normale objektorientierte Programmierung möglich

Tabelle 4.4 Problemlösungen mit klassischen Systemen im Vergleich zu SMOs

4.13 Zusammenfassung und Ausblick

Mit dem hier vorgestellten Modell kann man alle wichtigen Arten von Sicherheitsstrategien implementieren, die in herkömmlichen Systemen Verwendung finden. Zusätzlich lassen sich weitere Strategien realisieren, die in herkömmlichen Systemen unmöglich, jedoch besonders für objektorientierte Programmierung wichtig sind, wie transitive Capabilities, transitive Zugriffslisten und rollenbasierte Identitäten. Diese Strategien berücksichtigen die hohe Dynamik der Objektorientierung – trotz häufigem Referenzaustausch zwischen Programmteilen und Interaktion mit vielen verschiedenen Kommunikationspartnern läßt sich eine Sicherheitsstrategie auf hohem Abstraktionsniveau festlegen.

Zudem kann die Implementation der Sicherheitsstrategie weitgehend getrennt von der Implementation der Applikationsklassen selbst erfolgen. Dies wurde in diesem Kapitel motiviert, indem gezeigt wurde, wie man Klassen durch Verwendung von entsprechenden SMOs an verschiedene Sicherheitsanforderungen anpassen kann. Der Sicherheitscode wurde dazu im wesentlichen in die SMOs verlagert, die initiale Bindung des SMOs an wenige initiale Referenzen beim Applikationsstart wurde jedoch entweder vorausgesetzt oder in das Applikationsprogramm mit hineingesetzt. Das reduziert die Festlegung der Sicherheitsstrategie auf wenige Zeilen in einer einzigen zentralen Methode. (Meist wird dies in der main-Methode eines Programmes erfolgen.) Natürlich wäre es wünschenswert, dies völlig von dem Programm zu trennen. Dazu würde man eine Registratur benötigen, in der man zu jeder Programminstanz SMO-Bindungen an die initialen Referenzen konfigurieren kann. Dies wird jedoch im Rahmen dieser Arbeit nicht weiter ausgeführt.

Es zeigt sich, daß auch die Sicherheitsmetaobjekte selbst von der Trennung vom Applikationscode profitieren: In vielen Situationen kann man Sicherheitsmetaobjekte verwenden, die Sicherheitsstrategien auf abstraktem Niveau implementieren, so daß man diese Sicherheitsmetaobjekte ohne Modifikationen für verschiedene Applikationen wiederverwenden kann.

5 Formales Modell

Wir wollen nun zwei Aspekte unseres Sicherheitsmodells formalisieren: Die Reduktion von SMO-Ketten (Abschnitt 4.7) und die rollenbasierten Identitäten (Abschnitt 4.5). Beide Aspekte sind relativ komplex, so daß es oft schwierig ist, intuitiv zu entscheiden, ob durch Kettenreduktion bzw. Verwendung rollenbasierter Identitäten möglicherweise Sicherheitslücken geschaffen werden. Wir werden bei der Formalisierung jeweils nur exakt einen Aspekt formalisieren, d.h. keine der beiden Formalisierungen gibt das Gesamtmodell wieder. Der Vorteil einer Aspekt-Formalisierung ist die Einfachheit des resultierenden formalen Modells. Beweise für relativ komplexe Probleme lassen sich mit einem solchen Modell noch durchführen. Eine Formalisierung des Gesamtmodells wurde nicht durchgeführt, da das resultierende formale Modell dann so kompliziert würde, daß kaum interessante Eigenschaften des Modells bewiesen werden könnten und daher das resultierende formale Modell uninteressant ist.

Die beiden folgenden Abschnitte sind völlig unabhängig voneinander, da sie verschiedene Aspekte des Modells formalisieren. Wir gehen davon aus, daß die Mengen von Dingen im System (Objekte, SMOs, Identitäten, virtuelle Domänen) während der Laufzeit des Systems konstant bleiben. Wenn im realen System beispielsweise ein SMO instantiiert wird, muß dieses SMO in unseren Modellen bereits von Anfang an existieren, wird jedoch erst ab dem Instantiierungszeitpunkt benutzt. Ferner sind zur Vereinfachung alle Mengen endlich. Dies stellt keine Einschränkung dar, wenn wir nur endliche Laufzeit des Systems betrachten.

5.1 Reduktion von SMO Ketten

In diesem Abschnitt wird die in Abschnitt 4.7 schon kurz vorgestellte SMO-Kettenreduktion formalisiert. Wenn ein SMO mehrfach an dieselbe Referenz angeheftet ist, kann man unter bestimmten Bedingungen Anheftungen ohne Änderung der Semantik der Referenz entfernen. Das Ziel der Formalisierung ist also, Semantikerhaltung bei Weglassen gewisser SMO-Anheftungen an einer Referenz nachzuweisen.

Dazu müssen wir Teilstücke von Referenzen formalisieren und die Semantik der angehefteten SMOs definieren.

Wir definieren eine Menge von SMOs im System:

M = Menge von SMOs im System

Da wir insbesondere SMOs betrachten, die Identitätsinformation zur Verfügung stellen bzw. verwenden, definieren wir nun eine Menge von Identitäten:

I = Menge von Identitäten

Die von uns betrachteten SMOs können nun die Menge von für einen Methodenaufruf verwendeten Identitäten beeinflussen (beispielsweise kann ein Identitäts-SMO eine Identität hinzufügen) bzw. anhand von Identitätsinformation und Information über den auszuführenden Methodenaufruf den Aufruf zulassen oder abweisen. Wir definieren dazu eine Menge von Aufrufinformation. Ein Element dieser Menge stellt Information über einen konkreten Aufruf dar, anhand derer der Aufruf zugelassen oder abgewiesen werden kann. Diese Information wird beispielsweise die aufzurufende Methode umfassen. Sie kann weitere Informationen, wie Parametertypen oder auch Zustände von globalen Variablen enthalten. Der Einfachheit halber werden wir in den Beispielen aber davon ausgehen, daß sie nur die aufzurufende Methode beinhaltet.

C = Menge von Aufrufinformationen

Jede SMO-Anheftung an eine Referenz wird in diesem Modell durch eine Funktion (*Id*) repräsentiert. Die Funktion berechnet, ob ein gewisser Aufruf zugelassen wird und – falls der Aufruf zugelassen wird – welche Identitätsinformation an das nächste in der Kette befindliche SMO weitergeleitet wird. Das Ergebnis der Funktion *Id* ist also entweder eine Menge von Identitätsinformationen, die an das nächste SMO weitergeleitet werden oder das Fehlersymbol $\bot$, um den Aufruf abzuweisen. Falls wir im Zuge von SMO-Komposition einer Funktion *Id* das Fehlersymbol $\bot$ als Parameter übergeben, ist das Ergebnis der Funktion ebenfalls $\bot$. D.h. sobald eine *Id*-Funktion in der Kette entschieden hat, den Aufruf abzuweisen, kann eine spätere Funktion den Aufruf nicht nachträglich genehmigen, sondern der Aufruf gilt damit als endgültig gescheitert (Abbildung 5.1). Die Funktion *Id* kann von den hier betrachteten SMOs nicht frei gewählt werden, sondern muß, je nach SMO-Typ, bestimmte Anforderungen erfüllen. Zusätzlich entscheidet die Art der Anheftung des SMOs über die Funktion *Id*. Daher wurde als Index hier zunächst ein “?” angegeben, die Abhängigkeit von SMO und Anheftung wird später definiert.

$c \in C, i \subseteq I$: $Id_?(c, i)$ = Identitäten, die das nächste SMO bekommt

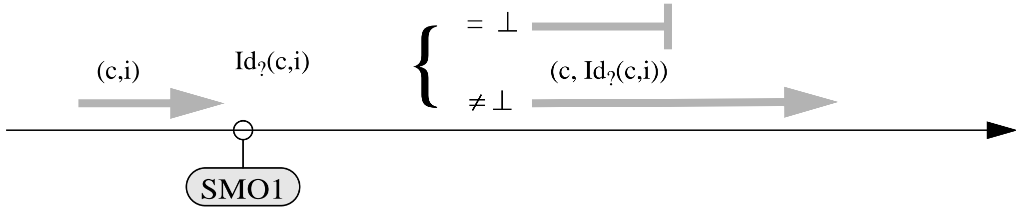

Abbildung 5.1 Informationsfluß bei Aufrufen

Wir unterscheiden drei Typen von SMOs: identitätsmodifizierende SMOs, identitätsbenutzende SMOs und identitätsneutrale SMOs. Jedes SMO hat genau einen dieser drei Typen. Abhängig von der Art der Anheftung an eine Referenz wird es innerhalb seines Typs als "positiv" oder "negativ", also beispielsweise "positiv identitätsbenutzend", bezeichnet. Die Zuordnung von quell- bzw. ziel-angeheftet zu positiv bzw. negativ ist dabei nicht festgelegt, sondern kann bei verschiedenen SMOs unterschiedlich sein.

5.1.1 Identitätsmodifizierende SMOs

Identitätsmodifizierende SMOs stellen Identitäten zur Verfügung oder entfernen Identitäten, implementieren selbst aber keine Zugriffsbeschränkungen. Zu jedem solchen SMO *m* gibt es zwei Mengen von Identitäten, I_{m+} und I_{m-}, die als zusätzliche Identitätsinformation in eine Richtung zur Verfügung gestellt werden und in die andere Richtung entfernt werden. Falls das SMO *m* also positiv identitätsmodifizierend angeheftet ist, werden die Identitäten I_{m+} zusätzlich zur Verfügung gestellt, falls es andersherum (d.h. negativ identitätsmodifizierend) angeheftet ist, werden die Identitäten I_{m-} entfernt. Die *Id*-Funktionen werden folgendermaßen definiert:

$$Id_{m+}(c, i) = i \cup I_{m+}$$

$$Id_{m-}(c, i) = i \setminus I_{m-}$$

Der "\" Operator bedeutet dabei, daß alle Elemente, die sich in der rechten Menge befinden, aus der linken entfernt werden, sofern sie in der linken Menge vorhanden sind.

Ein transitives Identitäts-SMO, das die Identität *myidentity* zur Verfügung stellen möchte, ist dann in Quellanheftung positiv identitätsmodifizierend, in Zielanheftung negativ identitätsmodifizierend, mit den Mengen: $I_{m+} = \{\text{myidentity}\}$ und $I_{m-} = \{ \}$.

Ein transitives Anonym-SMO, das alle Identitäten entfernen soll, ist in Quellanheftung negativ identitätsmodifizierend, in Zielanheftung positiv identitätsmodifizierend, mit den Mengen: $I_{m+} = \{ \}$ und $I_{m-} = I$.

5.1.2 Identitätsbenutzende SMOs

Identitätsbenutzende SMOs implementieren Zugriffskontrolllisten; sie erlauben Methodenaufrufe abhängig von den Aufruferidentitäten. D.h. wenn ein Aufrufer eine bestimmte Methode aufrufen möchte, muß er möglicherweise gewisse Identitätsinformation besitzen, um den Aufruf durchführen zu können. Da wir transitive identitätsbenutzende SMOs betrachten wollen, benötigen wir diese als positiv identitätsbenutzend und negativ identitätsbenutzend angeheftete SMOs. Diese beiden Anheftungen verhalten sich genau gleich, lediglich die Information, welche Aufrufe durchzulassen sind, stammt aus verschiedenen Mengen. Die erlaubten Aufrufe für ein positiv identitätsbenutzend angeheftetes SMO *m* befinden sich in der Menge $C_{m>} \subseteq C \times I$, bei negativ identitätsbenutzend angeheftetem SMO *m* in $C_{m<} \subseteq C \times I$.

Die *Id*-Funktion ergibt sich dann folgendermaßen:

$$Id_{m>}(c, i) = \begin{cases} i & \text{falls } \exists d \in i : (c, d) \in C_{m>} \\ \bot & \text{sonst} \end{cases}$$

$$Id_{m<}(c, i) = \begin{cases} i & \text{falls } \exists d \in i : (c, d) \in C_{m<} \\ \bot & \text{sonst} \end{cases}$$

Ein Zugriffskontrolllisten-SMO, das nur der Identität *myidentity* Zugriff gestatten möchte, ist dann in Zielanheftung positiv identitätsbenutzend, in Quellanheftung negativ identitätsbenutzend, mit den Mengen $C_{m>} = \{(c, d) | d = \text{myidentity}\}$ und $C_{m<} = C \times I$.

5.1.3 Identitätsneutrale SMOs

Identitätsneutrale SMOs implementieren Capabilities, d.h. sie erlauben bzw. verbieten Zugriffe unabhängig von der Aufruferidentität. Positiv und negativ identitätsneutral angeheftete SMOs verhalten sich gleich. Trotzdem benötigen wir wiederum beide Arten, um zwischen quell- und zielangehefteten identitätsneutralen SMOs unterscheiden zu können. Wir benötigen für ein identitätsneutral angeheftetes SMO daher eine Menge $C_{mN+} \subseteq C$ und $C_{mN-} \subseteq C$, die jeweils für die entsprechende Anheftungsrichtung die erlaubten Aufrufe beinhalten. Die *Id*-Funktionen ergeben sich dann folgendermaßen:

$$Id_{m\text{N+}}(c, i) = \begin{cases} i & \text{falls } c \in C_{m\text{N+}} \\ \bot & \text{sonst} \end{cases}$$

$$Id_{m\text{N-}}(c, i) = \begin{cases} i & \text{falls } c \in C_{m\text{N-}} \\ \bot & \text{sonst} \end{cases}$$

Ein Capability-SMO, das nur den Aufruf der Methode `Get` gestatten möchte, könnte dann in Zielanheftung positiv identitätsbenutzend, in Quellanheftung negativ identitätsbenutzend sein, mit den Mengen $C_{mN+} = \{\text{Get}\}$ und $C_{mN-} = C$.

5.1.4 Komposition und Inverse

Falls mehrere SMOs hintereinander hängen, werden sie vom Laufzeitsystem sequentiell involviert. Die SMOs können den Aufruf erlauben oder verbieten bzw. die Indentitätsinformation, die dem Aufruf mitgegeben wird, modifizieren. In unserem Modell wird dies durch die Komposition der *Id*-Funktionen ausgedrückt.

Wir definieren die Komposition von *Id*-Funktionen folgendermaßen:

$$Id_1 \bullet Id_2 \ (c, i) = Id_2(c, Id_1(c, i))$$

Die Methodenaufrufsinformation c wird an alle Funktionen übergeben und die Identitätsinformation wird jeweils aus der vorherigen *Id*-Funktion berechnet, wobei die linke Funktion zuerst aufgerufen wird.

Falls bei einem Methodenaufruf eine Referenz als Parameter übergeben wird, heften sich automatisch alle transitiven SMOs in umgekehrter Reihenfolge und umgekehrter Anheftungsrichtung an diese Referenz. Um dies mit *Id*-Funktionen ausdrücken zu können, definieren wir die Inverse einer *Id*-Funktion:

$$(Id_{m>})^{-1} = Id_{m<} \qquad (Id_{m<})^{-1} = Id_{m>}$$

$$(Id_{m+})^{-1} = Id_{m-} \qquad (Id_{m-})^{-1} = Id_{m+}$$

$$(Id_{m\mathrm{N}+})^{-1} = Id_{m\mathrm{N}-} \qquad (Id_{m\mathrm{N}-})^{-1} = Id_{m\mathrm{N}+}$$

$$(Id_1 \bullet Id_2)^{-1} = (Id_2)^{-1} \bullet (Id_1)^{-1}$$

Aus den Definitionen der Inversion wird sofort klar, daß für alle drei Arten von *Id*-Funktionen gilt:

$$((Id)^{-1})^{-1} = Id$$

Für die Komposition gilt dies ebenfalls:

$$((Id_1 \bullet Id_2)^{-1})^{-1} = ((Id_2)^{-1} \bullet (Id_1)^{-1})^{-1} = (((Id_1)^{-1})^{-1} \bullet ((Id_2)^{-1})^{-1}) = Id_1 \bullet Id_2$$

In den folgenden Abschnitten werden wir zeigen, daß wir bei SMO-Anheftungen gewisse Umstellungen bzw. Reduktionen vornehmen können. Wir müssen dazu jeweils zeigen, daß die Umstellung bzw. Reduktion die resultierende Funktion *Id*, die durch die betroffenen SMO-Anheftungen gebildet wird, erhält (d.h. $Id_{old} = Id_{new}$). Dies müssen wir für die Funktion selbst und für die inverse Funktion zeigen (d.h. $(Id_{old})^{-1} = (Id_{new})^{-1}$), damit transitive Parameterübergaben ebenfalls strategie-erhaltend sind. Wenn diese beiden Eigenschaften erfüllt sind, sind weitere Parameterübergaben unproblematisch: Bildet man von der inversen Gleichung wiederum das Inverse, erhält man die nicht-inverse Gleichung.

5.1.5 Reduktion doppelter SMOs

Wir wollen nun zeigen, daß man bei einem direkt hintereinander doppelt angehefteten SMO eine Anheftung entfernen kann.

Satz 5.1 **Wenn ein SMO zweimal direkt hintereinander auf die gleiche Art angeheftet ist, läßt sich eine Anheftung ohne semantische Änderung entfernen.**

Beweis:

(1) **Identitätsmodifizierende SMOs**

$$Id_{m+} \bullet Id_{m+}(c, i) = Id_{m+}(c, i \cup I_{m+}) = i \cup I_{m+} \cup I_{m+} = i \cup I_{m+} = Id_{m+}(c, i)$$

$$Id_{m-} \bullet Id_{m-}(c, i) = Id_{m-}(c, i \setminus I_{m-}) = (i \setminus I_{m-}) \setminus I_{m-} = i \setminus I_{m-} = Id_{m-}(c, i)$$

Da wir nun gezeigt haben, daß $Id_{m+} \bullet Id_{m+} = Id_{m+}$ und

$(Id_{m+} \bullet Id_{m+})^{-1} = Id_{m-} \bullet Id_{m-} = Id_{m-} = (Id_{m+})^{-1}$, kann man bei positiv identitätsmodifizierenden SMO-Dubletten eine der beiden Anheftungen ersatzlos entfernen. Bei negativ identitätsmodifizierenden SMO-Dubletten gilt dies ebenfalls, der Beweis kann analog geführt werden.

(2) **Identitätsbenutzende SMOs**

$$Id_{m>} \bullet Id_{m>}(c, i) = Id_{m>}\left(c, \begin{cases} i & \text{falls } \exists d \in i: (c, d) \in C_{m>} \\ \perp & \text{sonst} \end{cases}\right) = \begin{cases} i & \text{falls } \exists d \in i: (c, d) \in C_{m>} \\ \perp & \text{sonst} \end{cases}$$

$$= Id_{m>}(c, i)$$

Der Beweis für $Id_{m<}$ kann analog geführt werden. Da wir wiederum für die Funktion selbst und die Inverse den Beweis geführt haben, haben wir damit gezeigt, daß doppelt angeheftete identitätsbenutzende SMOs ersatzlos entfernt werden können.

(3) **Identitätsneutrale SMOs**

$$Id_{mN+} \bullet Id_{mN+}(c, i) = Id_{mN+}\left(c, \begin{cases} i & \text{falls } c \in C_{mN+} \\ \perp & \text{sonst} \end{cases}\right) = \begin{cases} i & \text{falls } c \in C_{mN+} \\ \perp & \text{sonst} \end{cases} = Id_{mN+}(c, i)$$

Der Beweis kann für negativ-identitätsneutrale SMOs analog geführt werden.

Damit haben wir gezeigt, daß man unabhängig von weiteren SMOs, die an einer Referenz hängen, bei SMO-Dubletten eine Anheftung ersatzlos entfernen kann.

5.1.6 Vertauschung von SMOs

Oft ist es nötig, SMOs zu vertauschen, um eine der im vorigen Abschnitt beschriebenen Reduktionen vornehmen zu können.

Satz 5.2 Zwei auf die gleiche Art identitätsmodifizierend angeheftete SMOs kann man vertauschen.

Beweis:

$$Id_{m1+} \bullet Id_{m2+}(c, i) = Id_{m2+}(c, i \cup I_{m1+}) = i \cup I_{m1+} \cup I_{m2+} = i \cup I_{m2+} \cup I_{m1+} = Id_{m2+} \bullet Id_{m1+}(c, i)$$

$$Id_{m2-} \bullet Id_{m1-}(c, i) = Id_{m1-}(c, i \setminus I_{m2-}) = (i \setminus I_{m2-}) \setminus I_{m1-} = (i \setminus I_{m1-}) \setminus I_{m2-} = Id_{m1-} \bullet Id_{m2-}(c, i)$$

Die obigen Gleichungen sind jeweils invers zueinander, denn es gilt:

$(Id_{m1+} \bullet Id_{m2+})^{-1} = Id_{m2-} \bullet Id_{m1-}$

q.e.d.

Satz 5.3 Identitätsbenutzende SMOs und identitätsneutrale SMOs kann man beliebig vertauschen.

Beweis:

Seien Id_1 und Id_2 die *Id*-Funktionen der zwei identitätsbenutzenden oder identitätsneutralen SMO-Anheftungen.

Zu zeigen ist dann, daß $Id_1 \bullet Id_2(c, i) = Id_2 \bullet Id_1(c, i)$

Wir unterscheiden nun vier Fälle.

(1) $Id_1(c, i) = \bot$ und $Id_2(c, i) = \bot$

Hier ist sofort einsichtig, daß $Id_1 \bullet Id_2(c, i) = \bot = Id_2 \bullet Id_1(c, i)$, da bei Komposition gemäß Definition das Ergebnis immer $\bot$ ist, sofern einer der Parameter $\bot$ ist.

(2) $Id_1(c, i) = \bot$ und $Id_2(c, i) \neq \bot$

Da die *Id*-Funktionen identitätsbenutzend oder identitätsneutral sind, bedeutet $Id_2(c, i) \neq \bot$ automatisch, daß $Id_2(c, i) = i$.

D.h.:

$Id_1 \bullet Id_2(c, i) = Id_2(c, Id_1(c, i)) = Id_2(c, \bot) = \bot$

$Id_2 \bullet Id_1(c, i) = Id_1(c, Id_2(c, i)) = Id_1(c, i) = \bot$

(3) $Id_1(c, i) \neq \bot$ und $Id_2(c, i) = \bot$

Der Beweis kann analog zu (2) geführt werden. Beide Seiten der zu beweisenden Gleichung resultieren zu $\bot$.

(4) $Id_1(c, i) \neq \bot$ und $Id_2(c, i) \neq \bot$

Gemäß Argumentation von (2) ist dann $Id_1(c, i) = i$ und $Id_2(c, i) = i$.

$Id_1 \bullet Id_2(c, i) = Id_2(c, Id_1(c, i)) = Id_2(c, i) = i$

$Id_2 \bullet Id_1(c, i) = Id_1(c, Id_2(c, i)) = Id_1(c, i) = i$

q.e.d.

Satz 5.4 Identitätsmodifizierende SMOs und identitätsneutrale SMOs kann man beliebig vertauschen.

Sei $Id_{\pm}$ identitätsmodifizierend (positiv oder negativ) und Id_N identitätsneutral.

$$Id_{\pm} \bullet Id_N(c, i) = Id_N(c, Id_{\pm}(c, i)) = \begin{cases} Id_{\pm}(c, i) & \text{falls } c \in C_{\mathrm{N}} \\ \bot & \text{sonst} \end{cases}$$

$$Id_N \bullet Id_{\pm}(c, i) = Id_{\pm}(c, Id_N(c, i)) = Id_{\pm}\left(c, \begin{cases} i & \text{falls } c \in C_{\mathrm{N}} \\ \bot & \text{sonst} \end{cases}\right) = \begin{cases} Id_{\pm}(c, i) & \text{falls } c \in C_{\mathrm{N}} \\ \bot & \text{sonst} \end{cases}$$

q.e.d.

5.1.7 Nicht vertauschbare SMOs

In einigen Fällen ist ein SMO mehrfach angeheftet, die Anheftungsstellen liegen allerdings nicht direkt nebeneinander, sondern sind durch andere SMOs getrennt. Man kann nun versuchen, durch Vertauschung von SMOs die fraglichen Anheftungen nebeneinander zu bekommen. Wenn dies jedoch nicht möglich ist, kommt man mit obigen Regeln nicht weiter.

Wir wollen daher untersuchen, in welchen Fällen man trotzdem SMO-Anheftungen entfernen kann.

Satz 5.5 Ein doppelt positiv (negativ) identitätsbenutzend angeheftetes SMO läßt sich bei mittlerem positiv identitätsmodifizierendem SMO nach vorne reduzieren.

D.h.: $Id_{m1>} \bullet Id_{m2+} \bullet Id_{m1>} = Id_{m1>} \bullet Id_{m2+}$ bzw. $Id_{m1<} \bullet Id_{m2+} \bullet Id_{m1<} = Id_{m1<} \bullet Id_{m2+}$

Da sich positiv und negativ identitätsbenutzende SMOs gleich verhalten, beweisen wir hier nur die erste Gleichung.

Beweis:

Wir unterscheiden zwei Fälle:

(1) $Id_{m1>}(c, i) = \bot$

Aus der Definition der Komposition folgt dann, daß

$$Id_{m1>} \bullet Id_{m2+} \bullet Id_{m1>}(c, i) = \bot = Id_{m1>} \bullet Id_{m2+}(c, i)$$

(2) $Id_{m1>}(c, i) \neq \bot$

Damit ist zwangsläufig $Id_{m1>}(c, i) = i$, und es muß also gelten: $\exists d \in i: (c, d) \in C_{m1>}$

Das bedeutet, daß

$$Id_{m1>} \bullet Id_{m2+}(c, i) = Id_{m2+}(c, i) = i \cup I_{m2+}$$

und

$$Id_{m1>} \bullet Id_{m2+} \bullet Id_{m1>}(c, i) = Id_{m1>}(c, i \cup I_{m2+})$$

$$= \begin{cases} i \cup I_{m2+} & \text{falls } \exists d \in i \cup I_{m2+}\text{: } (c, d) \in C_{m1>} \\ \bot & \text{sonst} \end{cases} = i \cup I_{m2+}$$

Das d muß existieren, denn man kann einfach das d aus der obigen Voraussetzung nehmen. Wenn es in I_{m2+} enthalten ist, ist es natürlich auch in $i \cup I_{m2+}$ enthalten.

q.e.d.

Um nun diese Reduktion verwenden zu können, muß das negative Äquivalent ebenfalls gelten:

Satz 5.6 Ein doppelt negativ (positiv) identitätsbenutzend angeheftetes SMO läßt sich bei mittlerem negativ identitätsmodifizierendem SMO nach hinten reduzieren.

D.h.: $Id_{m1<} \bullet Id_{m2-} \bullet Id_{m1<} = Id_{m2-} \bullet Id_{m1<}$

Beweis:

Wir unterscheiden wiederum zwei Fälle:

(1) $Id_{m1<} \bullet Id_{m2-} \bullet Id_{m1<}(c, i) = \bot$

Das bedeutet dann, daß entweder die vordere negativ identitätsbenutzende *Id*-Funktion oder die hintere negativ identitätsbenutzende *Id*-Funktion $\bot$ ergibt, also:

$$Id_{m1<}(c, i) = \bot \vee Id_{m1<} \bullet Id_{m2-}(c, i) = \bot$$

$$\Leftrightarrow (\neg\exists d \in i\text{: } (c, d) \in C_{m1<}) \vee (\neg\exists d \in i \setminus I_{m2-}\text{: } (c, d) \in C_{m1<})$$

$$\Leftrightarrow \forall d \in i\text{: } (c, d) \notin C_{m1<} \vee \forall d \in i \setminus I_{m2-}\text{: } (c, d) \notin C_{m1<}$$

$$\Leftrightarrow \forall d \in i\text{: } (c, d) \notin C_{m1<} \vee \forall d \in i\text{: } ((c, d) \notin C_{m1<} \vee c \in I_{m2-})$$

Den linken Teil der Aussage kann man weglassen, da der rechte Teil die weniger restriktive Aussage enthält: immer wenn der linke Teil wahr wird, gilt dies automatisch auch für den rechten.

$$\Leftrightarrow \forall d \in i \setminus I_{m2-}\text{: } (c, d) \notin C_{m1<}$$

Dies bedeutet nun, daß $Id_{m2-} \bullet Id_{m1<}(c, i) = \bot$

q.e.d.

(2) $Id_{m1<} \bullet Id_{m2-} \bullet Id_{m1<}(c, i) \neq \bot$

Da keine der beiden $Id_{m1<}$ Funktionen $\bot$ ergeben kann, gilt:

$$Id_{m1<} \bullet Id_{m2-} \bullet Id_{m1<}(c, i) = Id_{m1<}(c, Id_{m2-}(c, Id_{m1<}(c, i))) = Id_{m1<}(c, Id_{m2-}(c, i))$$

$$= Id_{m1<}(c, i \setminus I_{m2-})$$

Da diese Gleichung nicht $\perp$ ergibt, muß also gelten:

$\exists d \in i \setminus I_{m2\text{-}}: (c, d) \in C_{m1<}$

und

$Id_{m1<}(c, i \setminus I_{m2\text{-}}) = i \setminus I_{m2\text{-}}$

Damit folgt:

$Id_{m1<} \bullet Id_{m2\text{-}} \bullet Id_{m1<}(c, i) = i \setminus I_{m2\text{-}} = Id_{m2\text{-}} \bullet Id_{m1<}(c, i)$

q.e.d.

Satz 5.7 Ein doppelt positiv identitätsmodifizierend angeheftetes SMO läßt sich bei mittlerem positiv (negativ) identitätsbenutzendem SMO nach vorn reduzieren.

Satz 5.8 Ein doppelt negativ identitätsmodifizierend angeheftetes SMO läßt sich bei mittlerem negativ (positiv) identitätsbenutzendem SMO nach hinten reduzieren.

Der Beweis für diese beiden Sätze kann analog zu dem Beweis der Sätze 5.5 und 5.6 geführt werden. Wir wollen den Beweis daher hier nicht ausführen.

Die bisherigen Sätze können nur angewandt werden, wenn zwischen zwei SMO-Anheftungsstellen genau eine mittlere Anheftungsstelle liegt. Wenn jedoch mehrere andere SMOs dazwischen liegen, lassen sich die Sätze nicht anwenden. Statt nun für komplexe Kombinationen von SMOs zu zeigen, wann man SMOs entfernen kann, wollen wir statt dessen eine generische Aussage beweisen, die die obigen Sätze auch für komplexe Situationen anwendbar macht:

Satz 5.9 Die Reduktionsgesetze sind transitiv.

D.h.:

$$(Id_1 \bullet Id_2 \bullet Id_1 = Id_2 \bullet Id_1) \wedge (Id_1 \bullet Id_3 \bullet Id_1 = Id_3 \bullet Id_1)$$
$$\Rightarrow (Id_1 \bullet Id_2 \bullet Id_3 \bullet Id_1 = Id_2 \bullet Id_3 \bullet Id_1)$$

bzw.:

$$(Id_1 \bullet Id_2 \bullet Id_1 = Id_1 \bullet Id_2) \wedge (Id_1 \bullet Id_3 \bullet Id_1 = Id_1 \bullet Id_3)$$
$$\Rightarrow (Id_1 \bullet Id_2 \bullet Id_3 \bullet Id_1 = Id_1 \bullet Id_2 \bullet Id_3)$$

Die Beweise für diese beiden Eigenschaften können in analoger Weise geführt werden. Wir wollen daher hier lediglich eine der beiden Aussagen beweisen.

Sei also $Id_1 \bullet Id_2 \bullet Id_1 = Id_1 \bullet Id_2$ und $Id_1 \bullet Id_3 \bullet Id_1 = Id_1 \bullet Id_3$.

Dann gilt:

$$Id_1 \bullet Id_2 \bullet Id_3 \bullet Id_1 = (Id_1 \bullet Id_2) \bullet Id_3 \bullet Id_1 = (Id_1 \bullet Id_2 \bullet Id_1) \bullet Id_3 \bullet Id_1$$
$$= Id_1 \bullet Id_2 \bullet (Id_1 \bullet Id_3 \bullet Id_1) = Id_1 \bullet Id_2 \bullet (Id_1 \bullet Id_3) = (Id_1 \bullet Id_2 \bullet Id_1) \bullet Id_3$$
$$= (Id_1 \bullet Id_2) \bullet Id_3 = Id_1 \bullet Id_2 \bullet Id_3$$

q.e.d.

5.1.8 Grenzen des Modells

Bei diesem Modell wurden zwei Dinge vorausgesetzt:

(1) Mit den SMOs wird nur mittels der *Id*-Funktion interagiert.

(2) Die betrachteten SMOs verhalten sich wie durch die *Id*-Funktion spezifiziert.

Die Aussage (1) bedeutet, daß nicht von anderen SMOs oder von Benutzerapplikationen auf die betrachteten SMOs Einfluß genommen wird, d.h. daß sie nicht durch andere SMOs oder Benutzerapplikationen entfernt, ersetzt oder modifiziert werden.

Die Aussage (2) scheint zunächst trivial. Es muß jedoch berücksichtigt werden, daß SMOs auch durch Benutzer implementiert werden können. Da die Semantik eines SMOs im allgemeinen unentscheidbar ist, läßt sich nicht feststellen, ob das SMO eine bestimmte Art von *Id*-Funktion realisiert. Man darf es nun keinesfalls dem Benutzer überlassen, die Art der *Id*-Funktion zu spezifizieren, da er durch falsche Angabe nicht nur die Semantik seines SMOs modifizieren kann, sondern, bei vermeintlich semantikerhaltender Vertauschung von SMOs durch das System, die Strategien anderer SMOs außer Kraft setzen kann.

Man sollte also für SMO-Klassen aus einer Standard-SMO-Bibliothek durch den Systemadministrator die Eigenschaften festlegen lassen und bei benutzerimplementierten SMOs keine Eigenschaften annehmen. D.h. benutzerimplementierte SMOs lassen sich nicht durch das System reduzieren. Falls natürlich an einer Referenz mehrere SMOs eines Benutzers hintereinander hängen, kann der Benutzer für diesen Abschnitt die Reduktion selbst implementieren.

5.1.9 Zusammenfassung

Durch mehrmaliges Transferieren einer Referenz über eine SMO-geschützte Referenz können sich SMO-Ketten bilden, also Referenzen, bei denen ein SMO mehrmals an der Referenz hängt. Solche mehrmals angehefteten SMOs kann man in vielen – wenn auch nicht in allen – Fällen reduzieren, so daß lediglich eine Anheftung übrig bleibt. Die SMOs wurden in drei Kategorien unterteilt, die die gängigen SMOs umfassen: Zugriffslisten, Capabilities, Identitäts-SMOs. Dadurch wurden die Regeln für die Reduktion vereinfacht, so daß die Regeln auch automatisch und effizient durch ein Laufzeitsystem angewandt werden können.

5.2 Virtuelle Domänen und rollenbasierte Identitäten

In Abschnitt 4.5 wurden virtuelle Domänen und, darauf aufbauend, rollenbasierte Identitäten vorgestellt. Eine virtuelle Domäne besteht aus einer Menge von Objekten und Objektreferenzen. Nur Objekte der virtuellen Domäne können auf die Objektreferenzen der Domäne zugreifen. Die Domäne wird jedoch nicht durch die zugehörigen Objekte definiert, sondern durch den Domänenrand, der durch Grenz-SMOs gebildet wird. Die Grenz-SMOs erhalten dabei die Domänenrand-Eigenschaft, d.h. sie sorgen dafür, daß verschiedene Domänen getrennt bleiben und nicht unkontrolliert verschmelzen. Da die Anforderungen an die SMOs groß sind und die Strategie verteilt ist (sie bildet sich durch die Gesamtheit aller SMOs des Systems), soll hier anhand eines formalen Modells gezeigt werden, daß die Grenz-SMOs Domänen erhalten. Ferner soll die Semantik rollenbasierter Identitäten formalisiert werden, und verschiedene Eigenschaften rollenbasierter Identitäten sollen bewiesen werden [RiH98]. Wir wollen hier nur Grenz-SMOs betrachten. Alle anderen SMOs bleiben unberücksichtigt, da sie keinen Einfluß auf die Domäneneigenschaft haben.

Das formale Modell, das im folgenden vorgestellt wird, umfaßt Grenz-SMOs, Objektreferenzen, virtuelle Domänen und Identitäten. Es umfaßt keine Objekte; der Zustand der Objekte ist für die Formalisierung nicht relevant.

5.2.1 Basisdefinitionen

Für unser formales Modell benötigen wir SMOs, virtuelle Domänen und Objektreferenzen. Grenz-SMOs und virtuelle Domänen können wir direkt als Mengen definieren:

M = Menge von Grenz-SMOs im System

D = Menge von virtuellen Domänen im System

Die Definition von Objektreferenzen gestaltet sich komplizierter. Unser Modell soll die Referenzen inklusive angehefteter SMOs umfassen, wobei die Art der Anheftung (*src*, *dst*) beachtet werden muß. Die Menge aller möglichen Anheftungen ist dann:

A = Menge aller möglichen SMO-Anheftungen = $\{src, dst\} \times M$

Eine Objektreferenz besteht dann aus einer beliebigen Sequenz von Anheftungen und einem Zielobjekt. Das Zielobjekt ist in diesem Modell ohne Belang, daher besteht in diesem Modell eine Objektreferenz nur aus den Anheftungen:

OR = Menge von Objektreferenzen im System = A^*

Ein Beispiel für eine Objektreferenz wäre dann: $((src, m_1), (src, m_2), (dst, m_3))$

Wir berücksichtigen nur Grenz-SMOs, d.h. wenn an der Objektreferenz an irgendeiner Stelle nicht-Grenz-SMOs angeheftet sind, werden diese ignoriert, sie werden nicht als Anheftung in der Referenz dargestellt.

Die virtuellen Domänen sind hierarchisch organisiert (Abbildung 5.2).

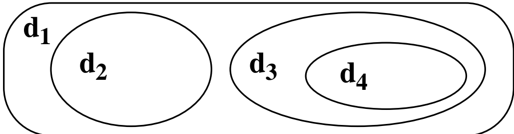

Abbildung 5.2 Virtuelle Domänen

Die hierarchische Organisation ergibt sich aus der Funktion *parent*, die die jeweils übergeordnete Domäne berechnet. Beispiel: $parent(d_2)$ ergibt d_1.

$$parent : D \cup \{\bot\} \rightarrow D \cup \{\bot\}$$

Die Funktion *parent* ist hierarchisch, d.h. sie hat keine Zyklen. Wenn wir also die Funktion *parent* mehrmals auf eine Domäne iteriert anwenden, ist das Ergebnis eine Top-Level-Domäne, also eine Domäne, die keine umgebende Domäne mehr besitzt. Die Funktion *parent* ergibt dann "undefiniert" ($\bot$). Die meisten betrachteten Systeme werden allerdings nur eine solche oberste Domäne besitzen. Für die Funktion *parent* muß gelten:

$$parent(\bot) = \bot$$

$$\forall d \in D: \exists n \in N: parent^{(n)}(d) = \bot$$

Die Menge der obersten Domänen ergibt sich aus:

$$D_{Top} = \{d \in D | parent(d) = \bot\}$$

5.2.2 Domänen

Wir wollen nun die Domänen und die enthaltenen Objektreferenzen, SMOs und Objekte genauer betrachten. Zunächst definieren wir den Ort der SMOs. Jedes SMO befindet sich in genau einer Domäne, die durch die Funktion *metadom* berechnet wird. Top-Level-Domänen können keine SMOs enthalten, da über den Rand von Top-Level-Domänen keine Referenzen zeigen können und damit keine Referenzen existieren, die mit SMOs in Top-Level-Domänen versehen werden müßten. Ein System, das zwischen allen Domänen Interaktion gestattet, muß daher genau eine Top-Level-Domäne besitzen, alle anderen Domänen sind in dieser enthalten. Separierte Systeme können auch aus mehreren Top-Level-Domänen bestehen.

$metadom : M \rightarrow D \backslash D_{Top}$ (definiert den Ort eines SMOs)

Nun soll der Ort von Objektreferenzen festgelegt werden. In unserem Modell ist der Ort einer Objektreferenz (d.h. in welcher Domäne sie verwendet werden kann) nicht beliebig, sondern ergibt sich direkt aus der Objektreferenz. Eine bestimmte Objektreferenz unseres Modells kann also nur in genau einer Domäne gültig sein. Würde sie an eine andere Domäne übergeben, würde sie durch SMOs modifiziert. Die modifizierte Objektreferenz wäre dann nur genau in der Zieldomäne gültig.

Wir können daher eine Funktion *dom* definieren, die die Domäne berechnet, in der eine Objektreferenz gültig ist.

$$dom : OR \cup \{\bot\} \rightarrow D \cup \{\bot\}$$

Die Funktion *dom* berechnet zu einer Objektreferenz eine Domäne oder ergibt "undefiniert", wenn die Referenz im System nicht existieren kann. (Eine Sonderrolle nimmt dabei die leere Referenz ( ) ein. Diese wird später genauer diskutiert.) Wenn die Funktion *dom* eine Domäne berechnet, bedeutet dies, daß in der berechneten Domäne eine oder mehrere solche Objektreferenzen existieren können (aber natürlich nicht müssen). Da das Zielobjekt durch die Objektreferenz in unserem Modell nicht festgelegt wird, können im realen System durchaus mehrere, verschiedene Referenzen existieren, die jedoch im Modell zu dieser einen Referenz zusammengefaßt werden.

Wir definieren nun die Funktion *dom* rekursiv:

$$dom(\bot) = \bot,\ dom(\ (\ )\ ) = \bot$$

$$dom((\mathrm{src}, m), or) = \begin{cases} metadom(m) & \text{wenn } or = (\ ) \vee dom(or) = parent(metadom(m)) \\ \bot & \text{sonst} \end{cases}$$

$$dom((\mathrm{dst}, m), or) = \begin{cases} parent(metadom(m)) & \text{wenn } or = (\ ) \vee dom(or) = metadom(m) \\ \bot & \text{sonst} \end{cases}$$

Eine Objektreferenz des Modells legt kein Zielobjekt fest. Jedoch ergibt sich aus der Objektreferenz (also aus den Anheftungen) direkt der Ort des Zielobjektes der Referenz. Dieser wird durch die Funktion *target* berechnet. Die Funktion *target* ergibt – wie die Funktion *dom* – "undefiniert", falls die Referenz nicht existieren kann, sonst gibt sie die Zieldomäne einer Referenz an.

$$target : OR \cup \{\bot\} \rightarrow D \cup \{\bot\}$$

$$target(\bot) = \bot,\ target(\ (\ )\ ) = \bot$$

$$target(or, (\mathrm{dst}, m)) = \begin{cases} metadom(m) & \text{wenn } dom(or, (\mathrm{dst}, m)) \neq \bot \\ \bot & \text{sonst} \end{cases}$$

$$target(or, (\mathrm{src}, m)) = \begin{cases} parent(metadom(m)) & \text{wenn } dom(or, (\mathrm{src}, m)) \neq \bot \\ \bot & \text{sonst} \end{cases}$$

Abbildung 5.3 zeigt vier Domänen, die wie abgebildet verschachtelt sind (beispielsweise ist $parent(d_2) = d_1$). Die SMOs des Systems befinden sich ebenfalls wie abgebildet in den entsprechenden Domänen (also z.B. $metadom(m_3) = d_4$).

In diesem System ist die Objektreferenz $or = ((src, m_1), (dst, m_2), (dst, m_3))$ gültig mit $dom(or) = d_2$ und $target(or) = d_4$. Die Objektreferenz *or* kann also in unserem System existieren und, falls sie existiert, befindet sie sich in Domäne d_2.

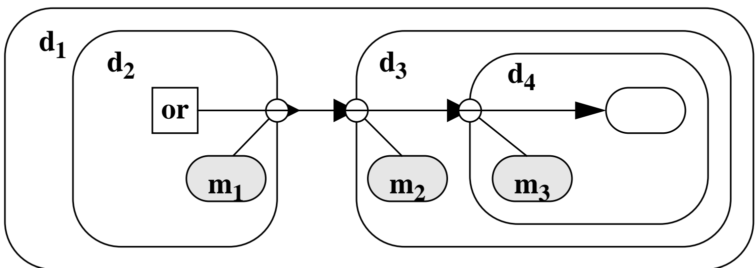

Abbildung 5.3 Beispiel für eine Objektreferenz

Eine Sonderrolle nimmt die leere Objektreferenz ( ) ein. Diese Referenz spezifiziert eine lokale Referenz, also beispielsweise eine Referenz in Domäne d_2 auf ein Objekt in Domäne d_2 ohne angeheftetes SMO. Da nun *dom* eine Funktion ist, kann sie nicht berechnen, in welchen Domänen die Referenz gültig ist, sondern ergibt "undefiniert". Diese Referenz ist jedoch trotzdem gültig in unserem System; es ist die einzige gültige Referenz, bei der die Funktion *dom* "undefiniert" ergibt.

5.2.3 Methodenaufrufe

Bis jetzt haben wir nur die Mengen (D und M) und die statischen Beziehungen zwischen den Systemkomponenten (die Funktionen *metadom* und *parent*) definiert (die Funktionen *dom* und *target* ergeben sich aus diesen Definitionen). Um Aussagen über das Systemverhalten machen zu können, benötigen wir die Definition von Systemzuständen und Systemübergängen.

Der momentane Zustand unseres Systems wird definiert durch die Menge von aktuell vorhandenen Objektreferenzen (die natürlich alle gültig sein müssen).

$$OR_{cur} \subseteq OR, \forall or \in OR_{cur}: dom(or) \neq \bot \vee or = (\ )$$

Wenn eine Objektreferenz in der Menge der momentanen Objektreferenzen ist, bedeutet das, daß eine oder mehrere solche Objektreferenzen zu diesem Zeitpunkt im realen System existieren können (aber nicht müssen). Wenn eine Objektreferenz nicht in der Menge der momentanen Objektreferenzen ist, heißt das, daß keine solche Objektreferenz zu diesem Zeitpunkt im realen System existieren kann.

Wir betrachten ein System ausgehend von einer Menge initialer Objektreferenzen OR_{init}. Der Systemzustand kann sich nun ausschließlich durch Methodenaufrufe ändern, da wir ein objektorientiertes System modellieren. Wenn wir also einen Methodenaufruf durchführen, können neue Objektreferenzen durch Parameterübergabe und Ergebnisrückgabe erzeugt werden. Diese werden dann zu der Menge aktuell vorhandener Objektreferenzen hinzugefügt.

Dazu wird die Funktion *meth* definiert. Sie benötigt eine Menge momentaner Objektreferenzen OR_1, eine Zielobjektreferenz *or*, an der die Methode aufgerufen wird, eine Parameterreferenz *p* und einen Rückgabewert *r*. Sie berechnet aus diesen Werten eine neue Menge von Objektreferenzen OR_2, die sich nach dem Methodenaufruf im System befinden. Diese Menge enthält dann zusätzlich zu den schon vorher im System befindlichen Referenzen die durch Parameterübergabe und Ergebnisrückgabe erzeugten Referenzen. Da das Modell weder Objekte noch Zustände von Objekten umfaßt, muß auch die aufgerufene Methode nicht spezifiziert werden. Wir wollen nur Aussagen über den Fluß von Objektreferenzen machen, daher ist die aufgerufene Methode irrelevant.

$$OR_2 = meth(OR_1, or, p, r) \quad \text{mit } OR_1, OR_2 \subseteq OR; \;\; or \in OR_1; \;\; p, r \in OR_1 \cup \{(\;)\}$$

Zur Vereinfachung unseres Modells nehmen wir an, daß jeder Methodenaufruf einen Parameter übergibt und einen Ergebniswert zurückliefert. Ferner benötigen Methodenaufrufe keine Zeit. D.h. in dem Modell sind zumindest durch direkte Abbildung keine verschachtelten Methodenaufrufe möglich. Um nun diese Möglichkeiten zu simulieren, kann folgende Abbildung verwendet werden:

- Ein Methodenaufruf ohne Parameter bzw. ohne Rückgabewert kann auf einen Methodenaufruf mit Übergabe bzw. Rückgabe eines zusätzlichen Objektes (dummy-Objekt) abgebildet werden.
- Ein Methodenaufruf mit mehreren Parametern kann durch mehrere Methodenaufrufe mit je einem Parameter modelliert werden.
- Verschachtelte Aufrufe können ebenfalls durch mehrere Aufrufe modelliert werden, wobei dann jeweils ein Aufruf für die Initiierung des realen Aufrufs (der die Parameter übergibt) und einer für die Beendigung des realen Aufrufes (der die Ergebnisse zurückgibt) anfällt (Abbildung 5.4).

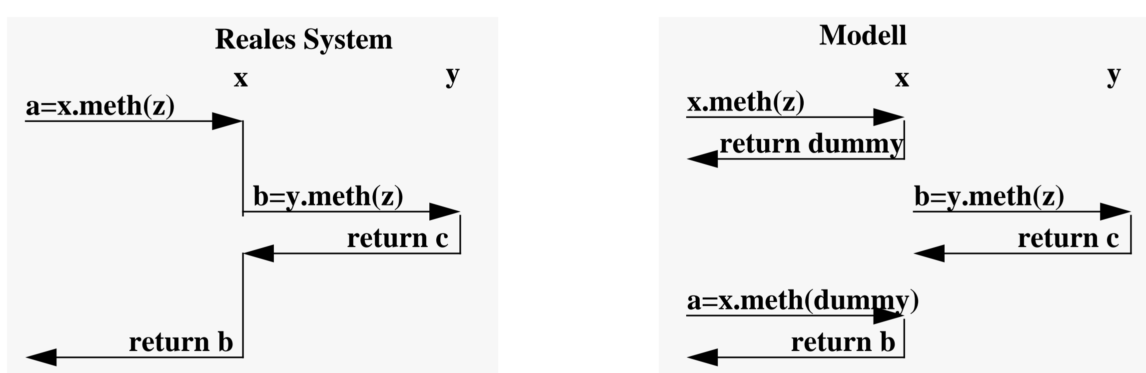

Abbildung 5.4 Abbildung von verschachtelten Aufrufen

Nun soll die Funktion *meth* genauer definiert werden. Bei gültigen Aufrufen fügt die Funktion *meth* zu der Menge von momentanen Objektreferenzen die von dem Methodenaufruf übergebenen Referenzen hinzu. Diese werden dazu von den Funktionen *param* und *ret* modifiziert, welche ihrerseits möglicherweise SMOs an die Referenzen hängen oder entfernen, bevor sie tatsächlich übergeben werden.

Der Aufruf ist nur gültig, wenn sich die Parameterreferenz in der Aufruferdomäne befindet und sich die Rückgabewertreferenz in der Domäne des Zielobjektes befindet. Außerdem müssen die Übergabefunktionen *param* und *ret* erfolgreich den zu übergebenden Wert berechnen. Falls der Aufruf nicht gültig ist, wird die Menge der momentanen Objektreferenzen nicht geändert.

$$meth(OR_1, or, p, r) = \begin{cases} OR_1 \cup \{param(or, p)\} \cup \{ret(or, r)\} \\ \quad \text{wenn } (dom(or) = dom(p) \vee p = (\ )\ ) \wedge ret(or, r) \neq \bot \wedge \\ \quad\ (target(or) = dom(r) \vee r = (\ )) \wedge param(or, p) \neq \bot \\ OR_1 \ \ \text{sonst} \end{cases}$$

Nun müssen die Funktionen *param* und *ret* definiert werden:

$$param((\ ), p) = p$$

$$param(((src, m), or), p) = param(or, out_m(p))$$

$$param(((dst, m), or), p) = param(or, in_m(p))$$

$$ret((\ ), r) = r$$

$$ret((or, (src, m)), r) = ret(or, in_m(r))$$

$$ret((or, (dst, m)), r) = ret(or, out_m(r))$$

Die beiden Funktionen greifen auf die Funktionen *in* und *out* zurück, die von jedem SMO einzeln definiert werden und bestimmen, wie Referenzen beeinflußt werden, die über eine Objektreferenz ein- oder ausgehen. Sie entsprechen den Methoden *incomingRef* und *outgoingRef* der SMOs. Die Funktionen *in* und *out* müssen bestimmte Eigenschaften besitzen, damit übergebene Referenzen gültig bleiben, also die *dom*-Funktion nicht $\bot$ zurückliefert. Dazu wird nun eine Definition von *in* und *out* angegeben, die diese Eigenschaft hat und die gewisse Freiheitsgrade besitzt, um später verschiedene Modelle realisieren zu können.

Die *in*-Funktion jedes SMOs hängt das SMO selbst transitiv an die Referenz.

$$in_m(p) = ((src, m), p)$$

Die *out*-Funktion prüft zunächst, ob die Übergabe der Referenz erlaubt ist. Dazu wird die von jedem SMO frei definierbare Funktion *allow* aufgerufen. Falls diese Funktion die Übergabe nicht erlaubt, liefert die Funktion *out* “undefiniert” und untersagt damit den Aufruf. Sonst wird der Aufruf zugelassen. Falls ein anderes SMO bereits in Quell-Anheftung vorn an der Referenz hängt, wird dieses entfernt, ansonsten hängt das SMO sich selbst transitiv in Ziel-Anheftung an die Referenz.

$$out_m((\ )) = \begin{cases} \perp & \text{wenn } \neg allow_m((\ )) \\ (dst, m) & \text{sonst} \end{cases}$$

$$out_m((src, m_2), or) = \begin{cases} \perp & \text{wenn } \neg allow_m((src, m_2), or) \\ or & \text{sonst} \end{cases}$$

$$out_m((dst, m_2), or) = \begin{cases} \perp & \text{wenn } \neg allow_m((dst, m_2), or) \\ ((dst, m), (dst, m_2), or) & \text{sonst} \end{cases}$$

Die Funktionen *param* und *ret* müssen bestimmte Eigenschaften besitzen, um die virtuelle Domäneneigenschaft zu erhalten. Die Funktion *param* muß entweder $\perp$ liefern, um den Aufruf abzuweisen, oder muß eine in der Zieldomäne gültige Referenz liefern.

(I) $dom(or) \neq \perp \wedge (dom(p) = dom(or) \vee p = (\ ))$
$\Rightarrow param(or, p) \in \{(\ ), \perp\} \vee dom(param(or, p)) = target(or)$

Die Funktion *ret* kann ebenfalls entweder $\perp$ oder eine in der Quelldomäne gültige Referenz liefern.

(II) $target(or) \neq \perp \wedge (dom(r) = target(or) \vee r = (\ ))$
$\Rightarrow ret(or, r) \in \{(\ ), \perp\} \vee dom(ret(or, r)) = dom(or)$

5.2.4 Beweis der Domänenerhaltung

Wir werden nun den Beweis für die Funktion *param* führen, d.h. wir werden (I) aus dem vorigen Abschnitt beweisen. Falls die Funktion *out* während der rekursiven Aufrufe einmal $\perp$ zurückliefert, ergibt die Funktion *param* ebenfalls $\perp$, und damit ist (I) erfüllt. Im folgenden Beweis werden wir daher den zweiten Fall annehmen, d.h. daß keine der aufgerufenen *out*-Funktionen $\perp$ liefert.

Der Beweis wird mit vollständiger Induktion geführt. Dazu wird die Größe einer Referenz als die Anzahl von Anheftungen, aus der sie besteht, definiert. Wir zeigen (I) zunächst für Objektreferenzen der Größe 1. (An Referenzen der Größe 0 kann man keine Methoden aufrufen, da dann die *dom*-Funktion $\perp$ ergibt.) Danach zeigen wir, daß (I) für Objektreferenzen der Größe n+1 gilt, wobei wir annehmen, daß (I) bereits für Referenzen mit einer Größe kleiner n+1 bewiesen wurde.

Sei also $dom(or) \neq \perp \wedge (dom(p) = dom(or) \vee p = (\ ))$

Induktionsverankerung

Es gibt zwei Möglichkeiten für Objektreferenzen der Größe 1, (src, m) und (dst, m). Betrachten wir zunächst den ersten Fall:

(1) $or = (src, m)$

Je nach Art des Parameters *p* verhält sich nun die *out*-Funktion unterschiedlich:

(1a) $p = ((dst, m_2), p_2) \vee p = ()$

$dom(param(or, p)) = dom(out_m(p)) = dom((dst, m), p)$

Nun kann man die Definition der *dom*-Funktion einsetzen. Die Bedingung für das Einsetzen, die durch die Definition der *dom*-Funktion festgelegt ist, fordert, daß entweder *p* leer ist, oder, falls *p* nicht leer ist, daß *dom(p)=metadom(m)*. Dies ist erfüllt, denn gemäß Voraussetzung ist *dom(p)=dom(or)=metadom(m)*. Es gilt also:

$dom((dst, m), p) = parent(metadom(m)) = target(or)$

q.e.d.

(1b) $p = ((src, m_2), p_2)$

Falls $p_2 = ()$, ist $param(or, p) = out_m(p) = ()$. Damit ist (I) erfüllt.

Sei also jetzt $p_2 \neq ()$

Dann gilt aber:

$dom(param(or, p)) = dom(out_m(p)) = dom(p_2)$

$= parent(metadom(m_2))$ (gemäß Definition von *dom(p)*, da *p* gültige Referenz ist)

$= parent(metadom(m))$ (da *dom(p) = dom(or)*)

$= target(or)$

q.e.d.

(2) $or = (dst, m)$

$dom(param(or, p)) = dom(in_m(p)) = dom((src, m), p)$

$= metadom(m)$ (da entweder *p = ( )* oder sonst *dom(p) = dom(or) = parent(metadom(m))*)

$= target(or)$

q.e.d.

Induktionsschluß

Nun gehen wir also davon aus, daß (I) bereits für Objektreferenzen der Größe n bewiesen wurde. Die Referenz *or* hat nun also Größe n+1, wobei n>0, d.h. die Größe von *or* ist mindestens 2. Wir unterscheiden wieder zwei Möglichkeiten:

(1) $or = ((src, m), or_2)$

(1a) $p = ((dst, m_2), p_2) \vee p = ()$

$dom(param(or, p)) = dom(param(or_2, out_m(p))) = dom(param(or_2, ((dst, m), p)))$

Da *or* eine gültige Referenz ist, gilt:

$dom(or_2) = parent(metadom(m)) = dom((dst, m), p)$

Die Länge von or_2 ist n, daher können wir die Induktionsannahme verwenden:

$dom(param(or_2, ((dst, m), p))) = target(or_2) = target(or)$

q.e.d.

(1b) $p = ((src, m_2), p_2)$

$dom(param(or, p)) = dom(param(or_2, p_2))$

Sofern nun $p_2 \neq (\ )$, gilt wegen der Definition von *dom(or)* und *dom(p)* und da *dom(or)=dom(p)*:

$dom(or_2) = parent(metadom(m)) = dom(p_2)$

Wir können daher die Induktionsannahme verwenden und erhalten:

$dom(param(or_2, p_2)) = target(or_2) = target(or)$

q.e.d.

Falls jedoch $p_2 = (\ )$ wird der Beweis komplexer. Da dann $dom(p_2) = \perp$ gilt, können wir die Induktion noch nicht verwenden, sondern müssen einen Schritt tiefer gehen:

(1b.i) $p_2 = (\ )$ und $or = ((src, m), (src, m_3), or_3)$

Falls $or_3 = (\ )$, ist

$dom(param(or, p)) = dom(param(((src, m), (src, m_3)), (src, m_2)))$

$= dom(param((src, m_3), (\ )))$

$= dom((dst, m_3)) = target((src, m), (src, m_3)) = target(or)$

q.e.d.

Falls $or_3 \neq (\ )$, gilt:

$dom(param(or, p)) = dom(param(((src, m), (src, m_3), or_3), (src, m_2)))$

$= dom(param(((src, m_3), or_3), (\ ))) = dom(param(or_3, (dst, m_3)))$

Da *or* gültige Objektreferenz ist, gilt: $dom(or_3) = parent(metadom(m_3)) = dom((dst, m_3))$, und wir können wiederum die Induktionsannahme verwenden:

$dom(param(or_3, (dst, m_3))) = target(or_3) = target(or)$

(1b.ii) $p_2 = (\ )$ und $or = ((src, m), (dst, m_3), or_3)$

Falls $or_3 = (\ )$, ist

$$dom(param(or, p)) = dom(param(((src, m), (dst, m_3)), (src, m_2)))$$

$$= dom(param((dst, m_3), ( ))) = dom((src, m_3)) = target((src, m), (dst, m_3)) = target(or)$$

q.e.d.

Falls $or_3 \neq ( )$, gilt:

$$dom(param(or, p)) = dom(param(((src, m), (dst, m_3), or_3), (src, m_2)))$$

$$= dom(param(((dst, m_3), or_3), ( ))) = dom(param(or_3, (src, m_3)))$$

Da nun $dom(or_3) = metadom(m_3) = dom((src, m_3))$, können wir wiederum die Induktionsannahme verwenden, und es gilt:

$$dom(param(or_3, (src, m_3))) = target(or_3) = target(or)$$

q.e.d.

(2) $or = ((dst, m), or_2)$

$$dom(param(or, p)) = dom(param(or_2, ((src, m), p)))$$

Da *or* eine gültige Referenz ist, gilt: $dom(or_2) = metadom(m) = dom((src, m), p)$,

und mit der Induktionsannahme folgt:

$$dom(param(or_2, ((src, m), p))) = target(or_2) = target(or)$$

q.e.d.

Damit wurde bewiesen, daß *param* stets in der Zieldomäne gültige Objektreferenzen erzeugt. Der Beweis für *ret* kann analog geführt werden und wird daher hier nicht gezeigt.

Wir haben damit bewiesen, daß Methodenaufrufe mit unserer SMO-Definition die virtuelle-Domänen-Eigenschaft erhalten. Die SMO-Funktionalität, die wir hier festgelegt haben, entspricht der Funktionalität der Grenz-SMOs aus Abschnitt 4.4.2. Wenn wir in einem System also initial nur Objektreferenzen haben, die die virtuelle-Domänen-Eigenschaft beachten, bleibt diese Eigenschaft erhalten.

5.2.5 Innere Schleifen

Innere SMO-Schleifen (Abbildung 5.5) sind gemäß der *dom*-Funktion zulässig, sind aber für die in weiteren Kapiteln vorgestellten Modelle problematisch. Für diese Modelle setzen wir daher voraus, daß es keine solchen Schleifen gibt. Wir definieren dazu nicht die *dom*-Funktion anders, sondern verlangen, daß in der Menge initialer Objektreferenzen OR_{init} keine der Referenzen innere Schleifen besitzt, d.h. daß für alle $or \in OR_{init}$ gilt:

$$\neg \exists or_1, m_2, m_3, or_4 : or = (or_1, (dst, m_2), (src, m_3), or_4)$$

Die Eigenschaft, daß keine inneren Schleifen existieren, bleibt im System dann unabhängig von den ausgeführten Methodenaufrufen erhalten.

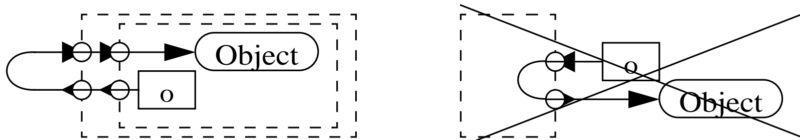

Abbildung 5.5 Innere SMO Schleifen sind für die meisten Modelle unzulässig

Beweis

Sei OR_1 ein Systemzustand, in dem keine Objektreferenz innere Schleifen enthält, und OR_2 der darauf folgende Zustand, der Objektreferenzen mit inneren Schleifen enthält. Dann muß es einen Methodenaufruf geben, der diese Referenz mit innerer Schleife erzeugt. Methodenaufrufe verändern Objektreferenzen nur mittels der *in*- und *out*-Funktionen. Es muß dann also eine Objektreferenz *or* geben, die keine innere Schleife enthält, auf die man die *in*- oder *out*-Funktion anwenden kann, und die dann eine innere Schleife enthält.

Wenn man die *in*-Funktion anwendet, wird an die Referenz vorn eine (src, m) Komponente gehängt. Dadurch kann keine innere Schleife entstehen. Wenn man die *out*-Funktion anwendet, wird entweder eine (src, m) Komponente entfernt, wodurch auch keine innere Schleife entstehen kann, oder es wird vorn eine (dst, m) Komponente angehängt. Dies geschieht jedoch nur, wenn vorher schon eine (dst, m_2) Komponente vorn hängt. Das bedeutet, wenn die Referenz nach der Anwendung der *out*-Funktion eine innere Schleife enthält, muß sie vorher auch schon eine innere Schleife enthalten haben. Dies widerspricht der Voraussetzung.

q.e.d.

5.2.6 Identitäten und Startwerte

Die virtuellen Domänen und rollenbasierten Identitäten dienen der Implementation von Authentifizierung von Aufrufen. Unser Modell kann bisher zwar die Domänen erhalten, wir haben jedoch nicht definiert, wie das System Identitätsinformation für Aufrufe verwendet. Wir müssen dazu eine Funktion definieren, die zu einer Objektreferenz (der Zielreferenz des Aufrufs) berechnet, welche Identität verwendet wird.

I = Menge von Identitäten

$Id: OR \rightarrow I^*$ bildet eine Objektreferenz auf Identitäten ab.

Für einen Methodenaufruf an einer Objektreferenz *or* werden die Identitätsinformationen *Id(or)* zur Authentifizierung verwendet.

Um ein System zu definieren, benötigen wir also zunächst die Grundmengen: *M*, *D*, *I*. Dann müssen wir die statischen Beziehungen zwischen Domänen (*parent*), SMOs und Domänen (*metadom*), Objektreferenzen und Identitäten (*Id*) festlegen, sowie die Funktionalität der SMOs definieren ($allow_m$ für jedes SMO).

Um nun einen Ablauf zu betrachten, benötigen wir einen Startzustand. Dieser wird durch die Menge initialer Objektreferenzen OR_{init} bestimmt.

Dadurch ist der Startzustand vollständig spezifiziert. Es kann dann mit der dynamischen Systemmodellierung begonnen werden, indem Methodenaufrufe auf dem Startzustand ausgeführt werden.

In den folgenden Abschnitten werden die Startzustände nicht vollständig spezifiziert, sondern es werden nur einige Eigenschaften des Startzustandes festgelegt. Innerhalb dieser Festlegung ist der Startzustand beliebig. Darauf basierend wird dann gezeigt, daß diese initial geforderten Eigenschaften persistent sind, d.h. daß diese Eigenschaften weiterhin gelten, egal welche Methodenaufrufe ausgeführt werden.

5.2.7 Beispiel: Disjunkte Interaktion

In diesem Abschnitt wird das in Abschnitt 4.5.3 betrachtete Beispiel für rollenbasierte Identitäten mit disjunkter Interaktion formalisiert. Die Formalisierung wird anhand des Beispiels vorgenommen, erfolgt aber so allgemein, daß sie auch für andere Systeme gültig ist.

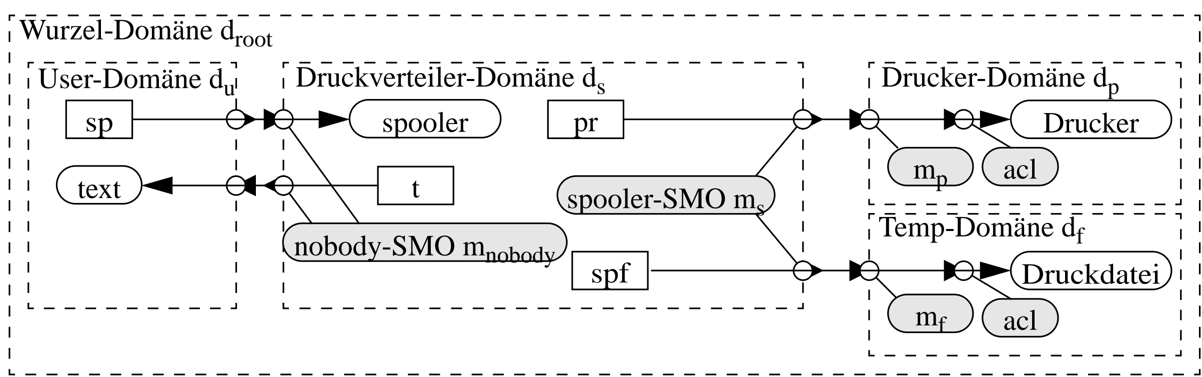

Abbildung 5.6 Rollenbasierte disjunkte Identitäten

In Abbildung 5.6 sind die wichtigsten Komponenten des Drucksystems mit disjunkter Interaktion dargestellt. Die Formalisierung erfordert folgende Schritte:

- Spezifikation der Grundmengen und statischen Funktionen
- Spezifikation des Anfangszustandes durch Definition von Eigenschaften des Systems
- Beweis, daß diese Eigenschaften unabhängig von den ausgeführten Methodenaufrufen erhalten bleiben

Definition der Grundmengen und statischen Funktionen

Unser System besteht aus einer Domäne d_s, die die disjunkte Interaktion implementiert (im konkreten Beispiel die Drucksystem-Domäne), und mehreren Domänen, deren Objekte Authentifizierung für Aufrufe benötigen (d_f, d_p). Diese Domänen werden, um allgemeine Beweise führen zu können, zu einer Menge von vertrauenswürdigen Domänen $D_{trusted}$ zusammengefaßt. Im Beispiel wäre also $D_{trusted} = \{d_f, d_p\}$. Im System muß es eine übergeordnete Domäne d_{root} zu diesen Domänen geben, und es kann noch weitere Domänen geben, die nicht-vertrauenswürdig sind, wie die Domäne d_u in der Abbildung.

Wir benötigen die SMOs m_p, m_s, m_f, m_{nobody}, die den Domänen wie in der Abbildung ersichtlich zugeordnet sind. Diese SMOs sind gleichzeitig die einzigen SMOs der betrachteten Domänen. Die *allow*-Funktion der SMOs wird folgendermaßen definiert:

$$allow_{m_{nobody}}(or) = \begin{cases} false & \text{falls } \exists or_2\colon or = ((src, m_s), or_2) \\ true & \text{sonst} \end{cases}$$

$$allow_{m_s}(or) = \begin{cases} false & \text{falls } \exists or_2\colon or = ((src, m_{nobody}), or_2) \\ true & \text{sonst} \end{cases}$$

Die *allow*-Funktionen der anderen SMOs können beliebig gewählt werden. Die Identitätenmenge und die Abbildung von Objektreferenzen auf diese geschieht folgendermaßen:

Die Menge der Identitäten enthält eine Identität I_s. Die Funktion *Id* ordnet einer Objektreferenz diese nur genau dann zu, wenn die Objektreferenz das Teilstück (src, m_s) enthält. Für die Implementation des SMOs bedeutet dies, daß das SMO m_s die Identität I_s für ausgehende Aufrufe bereitstellt.

Formal also: $I_s \in Id(or) \Leftrightarrow \exists or_2, or_3\colon or = (or_2, (src, m_s), or_3)$

Spezifikation des Anfangszustandes

Zur Vereinfachung des Schreibaufwands definieren wir die Menge von vertrauenswürdigen SMOs $M_{trusted}$. Dies ist die Menge der SMOs aus den vertrauenswürdigen Domänen und zusätzlich das SMO auf der vertrauenswürdigen Seite des Druckverteilers.

$M_{trusted} = \{m \in M | m = m_s \vee metadom(m) \in D_{trusted}\}$

Der Anfangszustand OR_{init} muß folgende Bedingung erfüllen: Vertrauenswürdige Domänen und der m_s-Teil der Druckverteiler-Domäne interagieren nur untereinander.

D.h. für alle $or \in OR_{init}$ gilt:

1) $m_1 \in M_{trusted} \wedge or = (or_2, (src, m_1), or_3) \Rightarrow or_2 = ( ) \wedge \exists m_2 \in M_{trusted}: or_3 = (dst, m_2)$

2) $m_1 \in M_{trusted} \wedge or = (or_2, (dst, m_1), or_3) \Rightarrow or_3 = ( ) \wedge \exists m_2 \in M_{trusted}: or_2 = (src, m_2)$

Diese Eigenschaften bleiben unabhängig von den ausgeführten Methodenaufrufen für spätere Zustände erhalten.

Beweis

Annahme: Die Eigenschaften bleiben nicht erhalten, d.h. es gibt einen Zustand OR_1, für den diese Eigenschaften (1) und (2) gelten, und dieser läßt sich durch Methodenaufruf (*meth*) auf einen Zustand OR_2 überführen, bei dem eine der Eigenschaften nicht mehr gilt.

Wenn (1) oder (2) nicht mehr erfüllt ist, bedeutet dies, daß es in OR_2 eine Objektreferenz gibt, die eine Komponente mit $m_1 \in M_{trusted}$ enthält, jedoch entweder (1) oder (2) nicht erfüllt. Da in OR_1 alle Objektreferenzen diese Eigenschaften hatten, muß durch den Methodenaufruf eine Objektreferenz neu entstanden sein, die die Eigenschaften nicht hat.

Es werden nun zwei Fälle unterschieden:

I) Der Methodenaufruf wurde an einer Referenz durchgeführt, die $m \in M_{trusted}$ enthält.

Da (1) und (2) gelten, muß die Zielreferenz *or* die Form $((src, m_1), (dst, m_2))$ mit $m_1 \in M_{trusted}$ und $m_2 \in M_{trusted}$ haben.

Dann ist $dom(or) = metadom(m_1)$ und $target(or) = metadom(m_2)$.

Der Parameter des Methodenaufrufes muß also ebenfalls aus $metadom(m_1)$ stammen, der Rückgabewert aus $metadom(m_2)$.

Daher gilt: $dom(p) = metadom(m_1)$, $dom(r) = metadom(m_2)$

Da weder die vertrauenswürdigen Domänen, noch die Druckverteiler-Domäne Sohn-Domänen haben, d.h. $parent(d_1) = d_2 \Rightarrow d_2 \notin \{d_s\} \cup D_{trusted}$, kann gemäß der Definition von *dom* also der Parameter p nur entweder die Form $p = ((src, m_1), (dst, m_3))$ mit $m_3 \in M_{trusted}$ oder, falls $m_1 = m_s$ ist, $p = ((src, m_{nobody}), or_2)$ haben.

Für die erste Möglichkeit ist dann

$$\begin{aligned} param(or, p) &= param(((src, m_1), (dst, m_2)), ((src, m_1), (dst, m_3))) \\ &= param((dst, m_2), (dst, m_3)) = ((src, m_2), (dst, m_3)) \end{aligned}$$

Die resultierende Referenz erfüllt aber (1) und (2).

Bei der zweiten Möglichkeit gilt wegen der Definition der *allow*-Funktion:

$$param(or, p) = param(((src, m_s), (dst, m_2)), ((src, m_{nobody}), (dst, m_3))) = \perp$$

Der Aufruf ist damit ungültig und der Methodenaufruf erzeugt keine neuen Referenzen.

Für den Rückgabewert (*ret*-Funktion) ist der Beweis analog und wird hier nicht dargestellt.

II) Der Methodenaufruf wurde an einer Referenz durchgeführt, die kein $m \in M_{trusted}$ enthält.

Da die SMOs sich nur jeweils selbst (und keine anderen SMOs) an neue Referenzen hängen, muß dann an der Parameterreferenz oder dem Rückgabewert bereits ein SMO $m_1 \in M_{trusted}$ vor dem Aufruf gehangen haben.

Wegen (1) und (2) muß der Parameter oder der Rückgabewert folgende Form haben: $or_2 = ((src, m_1), (dst, m_2))$ mit $m_1 \in M_{trusted}$ und $m_2 \in M_{trusted}$.

Wenn or_2 Parameter ist, bedeutet dies, daß die Zielreferenz des Aufrufs die Form $((src, m_3), or_3)$ mit $m_3 \in M_{trusted} \cup \{m_{nobody}\}$ haben muß, da sonst die *param*-Funktion "undefiniert" ergibt. Wegen Voraussetzung (II) muß dann $m_3 = m_{nobody}$ sein. Auch in diesem Fall ergibt sich allerdings wegen der *allow*-Funktion "undefiniert":

$$param(or, or_2) = param(((src, m_{nobody}), or_3), ((src, m_s), (dst, m_2))) = \perp$$

Wenn or_2 Rückgabewert ist, bedeutet dies, daß der Aufruf an einer Referenz der Form $(or_3, (dst, m_3))$ mit $m_3 \in M_{trusted} \cup \{m_{nobody}\}$ ausgeführt wird, da sonst die *ret*-Funktion "undefiniert" ergibt. Wegen Voraussetzung (II) muß dann $m_3 = m_{nobody}$ sein. Auch in diesem Fall ergibt sich allerdings wegen der *allow*-Funktion "undefiniert":

$$ret(or, or_2) = ret((or_3, (dst, m_{nobody})), ((src, m_s), (dst, m_2))) = \perp$$

Damit haben wir alle möglichen Fälle behandelt: Bei keinem Fall wurde (1) oder (2) verletzt, die Annahme, daß die Eigenschaften (1) oder (2) durch einen Methodenaufruf verletzt werden können, muß also fehlerhaft sein. Aus (1) und (2) ergibt sich dann mittels unserer Definition der *Id*-Funktion, daß tatsächlich die Druckverteiler-Identität nur für Interaktion des Druckverteilers mit den vertrauenswürdigen Domänen verwendet wird. In dem Beweis wurde außerdem deutlich (da wir dort alle Fälle betrachtet haben), daß durch Interaktion von nicht-vertrauenswürdigen Domänen keine vertrauenswürdigen Referenzen entstehen können.

5.2.8 Beispiel: Hierarchische Domänen

In diesem Abschnitt wird das in Abschnitt 4.5.4 betrachtete Beispiel der hierarchischen Domänen formalisiert. Wir werden zunächst wiederum die Grundmengen festlegen.

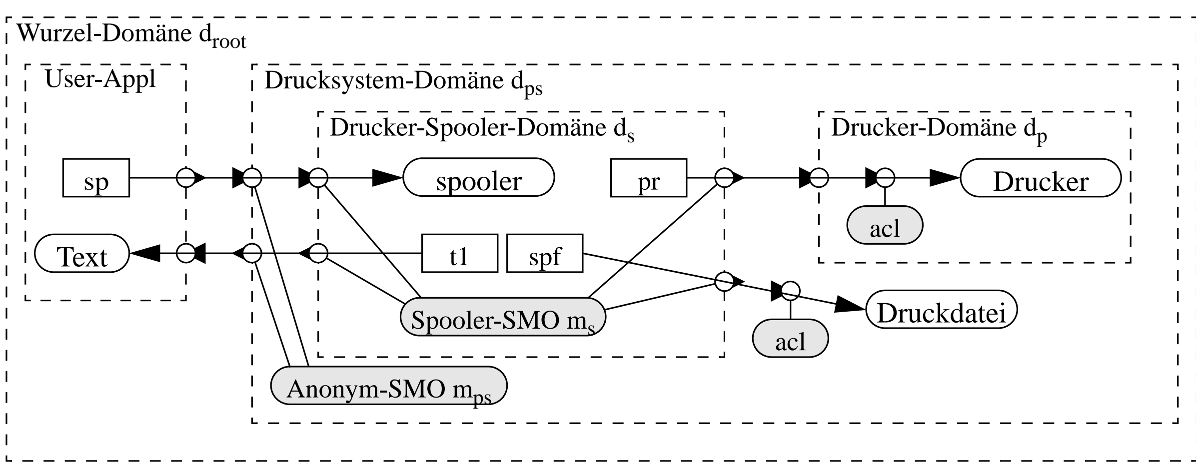

Abbildung 5.7 Druckverteiler mit hierarchischen Domänen

Unser System besteht aus einer Domäne d_{ps}, die das gesamte Drucksystem umfaßt, und mehreren vertrauenswürdigen Domänen, die innerhalb der Drucksystemdomäne liegen, in Abbildung 5.7 d_p und d_s. Diese Domänen werden, um allgemeine Beweise führen zu können, zu einer Menge von vertrauenswürdigen Domänen $D_{trusted}$ zusammengefaßt.

Wie abgebildet, benötigen wir das SMO m_{ps}, das die Anonymisierung von Aufrufen implementieren soll, und das SMO m_s, das Authentifizierung mit Druckverteiler-Identität I_s implementiert.

Die Authentifizierung (*Id*-Funktion) wird nun folgendermaßen definiert:

$$I_s \in Id(or) \Leftrightarrow \exists or_1, or_2, or_3 \colon or = (or_1, or_2, or_3) \wedge or_2 = (src, m_s)$$
$$\wedge \neg\exists or_4, or_5, or_6 \colon (or_3 = (or_4, or_5, or_6) \wedge or_5 = (src, m_{ps}))$$

Die Identität I_s wird also nur genau dann verwendet, wenn eine Komponente (src, m_s) in der Zielreferenz enthalten ist, sich jedoch keine Komponente (src, m_{ps}) weiter rechts davon befindet.

Die Menge der initialen Objektreferenzen OR_{init} darf keine inneren Schleifen enthalten, ist ansonsten aber beliebig.

Wir können nun folgendes beweisen:

(1) Methodenaufrufe an Objekte außerhalb der Domäne d_{ps} werden nicht mit I_s authentifiziert.

(2) Aufrufe an Referenzen, die von Objekten außerhalb der Domäne d_{ps} an eine Domäne oder ein Objekt in dieser Domäne übergeben werden, werden ebenfalls nicht mit I_s authentifiziert.

Die Aussage (1) ist trivial:

Eine Referenz *or* auf ein Objekt außerhalb der Domäne d_{ps} muß die Komponente (src, m_s) enthalten, damit Aufrufe mit I_s authentifiziert werden, d.h. $or = (or_1, (src, m_s), or_3)$. Das Teilstück or_3 kann nicht leer sein, da sonst die Referenz *or* auf ein Objekt in d_{ps} verweist. Das Stück or_3 kann auch nicht die Form $or_3 = ((dst, m), or_4)$ haben, da sonst eine innere Schleife nötig wäre, um die Referenz aus der Domäne zeigen zu lassen. Es muß also dann die Form $or_3 = ((src, m), or_4)$ haben. Aus der *dom*-Funktion folgt dann aber, daß $metadom(m) = d_{ps}$, daraus folgt wiederum, daß $m = m_{ps}$, da dies das einzige SMO dieser Domäne ist. Dann wird jedoch gemäß der *Id*-Funktion der Aufruf doch nicht mit I_s authentifiziert.

q.e.d.

Der Beweis für Aussage (2) ist etwas komplizierter:

Um eine Objektreferenz von außerhalb der Domäne d_{ps} an ein Objekt innerhalb der Domäne zu übergeben, muß der Aufruf entweder an einer außerhalb der Domäne befindlichen Referenz, die auf ein inneres Objekt verweist, durchgeführt werden. Dann kann die zu übergebende Referenz als Parameter übergeben werden. Oder der Aufruf wird an einer innerhalb der Domäne befindlichen Referenz, die auf ein äußeres Objekt verweist, durchgeführt, und die zu übergebende Referenz wird als Rückgabewert übergeben. Hier wird nur der erste Fall betrachtet, der zweite Fall kann analog bewiesen werden.

Sei also *or* eine äußere Referenz auf ein inneres Objekt, das bedeutet $dom(or) = d_1$ und $target(or) = d_2$ mit $\neg\exists n \in N_0\colon parent^{(n)}(d_1) = d_{ps}$ und $\exists n \in N_0\colon parent^{(n)}(d_2) = d_{ps}$.

Untersuchen wir nun die rechte Komponente der Referenz *or*. Es gibt zunächst zwei Möglichkeiten: $or = (or_2, (src, m))$ und $or = (or_2, (dst, m))$.

Wenn $or = (or_2, (src, m))$, kann or_2 keine *dst*-Komponenten mehr enthalten, da es keine inneren Schleifen bei Referenzen gibt. Wenn jedoch or_2 nur *src*-Komponenten enthält, muß $dom(or)$ eine Sohn-Domäne von $target(or)$ ergeben, was der Voraussetzung widerspricht.

Also muß $or = (or_2, (dst, m))$ gelten. Wir wollen nun zeigen, daß dann aber *or* die Komponente (dst, m_{ps}) enthalten muß.

Dazu werden wiederum zwei verschiedene Fälle unterschieden: $target((dst, m)) = d_{ps}$ und $target((dst, m)) \neq d_{ps}$.

Wenn $target((dst, m)) = d_{ps}$ dann muß $m = m_{ps}$ gelten.

q.e.d.

Wenn $target((dst, m)) \neq d_{ps}$, dann muß das Ziel in einer Sohn-Domäne von d_{ps} liegen. Damit gilt dann $\exists n \in N_0$: $parent^{(n)}(dom((dst, m))) = d_{ps}$.

Dann muß es wegen der Rekursion der *dom*-Funktion eine Stelle in der Referenz *or* geben mit $or = (or_1, or_2)$ und $dom(or_2) = d_{ps}$. Daher gilt, weil das Referenzstück or_1 nicht leer sein kann, $target(or_1) = d_{ps}$, und die rechte Komponente von or_1 muß (dst, m_{ps}) sein.

q.e.d.

Damit haben wir gezeigt, daß eine solche Referenz auf jeden Fall die Komponente (dst, m_{ps}) enthalten muß.

Jede übergebene Referenz p muß also über dieses Referenzstück übergeben werden, es wird also $in_{m_{ps}}(p)$ aufgerufen. Die *in*-Funktion hängt dann das Stück (src, m_{ps}) an die Referenz *p*. Weil es im System keine inneren Schleifen gibt, können weiter rechts von diesem Teilstück nur noch *src* Anheftungen mit SMOs aus Vater-Domänen von d_{ps} hängen, insbesondere kann also nicht (src, m_s) weiter rechts angeheftet sein. Aufrufe über diese Parameterreferenz werden daher gemäß der *Id*-Definition nicht mit I_s authentifiziert.

5.2.9 Grenzen des Modells

Die Formalisierung der virtuellen Domänen umfaßt nur Grenz-SMOs. Andere SMO-Anheftungen bleiben unberücksichtigt. Diese Problemreduktion ist nur zulässig, wenn andere SMOs keinen Einfluß auf die virtuellen Domänen bzw. Domänengrenzen nehmen können. Dies muß von den Grenz-SMOs garantiert werden.

5.3 Zusammenfassung

In diesem Kapitel wurden zwei Teilaspekte formalisiert: Reduktion von SMO-Ketten und virtuelle Domänen mit rollenbasierten Identitäten als Anwendung.

Mit dem formalen Modell für die Reduktion von SMO-Ketten wurde gezeigt, welche SMO-Anheftungen redundant sind und automatisch durch das System entfernt werden können. Dieses Modell kann in einem realen System direkt angewendet werden. Bei entsprechender Zuordnung von SMOs aus Klassenbibliotheken zu den verschiedenen Kategorien kann das System selbständig unnötige SMO-Anheftungen wegoptimieren.

Mit dem formalen Modell für virtuelle Domänen und rollenbasierte Identitäten wurden die zwei generischen Konfigurationen “disjunkte Interaktion” und “hierarchische Domänen” untersucht, und es wurde gezeigt, daß diese Konfigurationen die gewünschten Eigenschaften haben, insbesondere daß beide das Problem des böswilligen Unterschiebens von Referenzen lösen. Für viele

Situationen in realen Systemen ist eine der beiden Konfigurationen anwendbar. In einigen Fällen sind die Konfigurationen möglicherweise zu restriktiv. In diesen Fällen wird man das Modell als Standardwert nehmen, jedoch durch explizite Interaktion mit den SMOs Ausnahmen von den restriktive Regeln zulassen und diese genau durchdenken.

6 Implementation / Prototyp

Das hier vorgestellte Sicherheitsmodell basiert nur auf dem Basismechanismus des SMOs. Dieser ist allerdings sehr mächtig, so daß sich die Frage stellt, wie schwierig eine Implementation eines Systems ist, das solche SMOs anbietet. Dies soll in diesem Kapitel anhand des implementierten Prototyps dargestellt werden.

6.1 Implementationsplattform

Generell gibt es zwei Möglichkeiten für eine Implementation: Die SMOs können als Basismechanismus des Ablaufsystems eingeführt werden oder können auf dem Ablaufsystem aufbauend implementiert werden.

Wenn man die SMOs als Basismechanismus implementiert, benötigt man ein Ablaufsystem, das zu jeder Objektreferenz SMOs verwalten kann, die automatisch bei jedem Aufruf involviert werden. Ein System mit solchen Möglichkeiten ist das im Rahmen des SFB-182 entstandene System Metaxa[6] [KlG96], das aus einem erweiterten Java-Interpreter besteht. Eine Objektreferenz kann in diesem System mit Metaobjekten versehen werden, die jeden Methodenaufruf über die Referenz abfangen und beeinflussen können. So lassen sich auf einfache Weise beispielsweise Capability-SMOs implementieren. Mit solch einem Basissystem kann man das gesamte Modell implementieren. Der einzige, generelle Nachteil eines solchen Systems ist, daß existierende Systeme meist keine Metaobjekte kennen, so daß man (wie bei Metaxa) ein eigenes, proprietäres System verwenden muß. Für die Prototypimplementation wurde Metaxa evaluiert. Da auch komplexe Situationen modelliert werden sollten, wurden Geschwindigkeitstests mit Metaxa durchgeführt, die zeigten, daß das Metaxa-System (zumindest in der damaligen Version) einerseits bei der Programmausführung langsam ist (was für die Verwendung von Verschlüsselungsalgorithmen problematisch ist) und andererseits nicht skaliert. Bei zunehmender Anzahl von Metaobjekt-Anheftungen an Objektreferenzen wurde das Gesamtsystem immer langsamer. Da gerade bei virtuellen Domänen und allgemein bei Transitivität viele SMOs angeheftet und entfernt werden müssen, war dies ebenfalls ein entscheidendes Kriterium. Nicht zuletzt war

6. Das in der Literaturreferenz als MetaJava bezeichnete System wurde inzwischen in Metaxa umbenannt.

auch der fehlende Sicherheitsaspekt problematisch. In Metaxa konnte der Variablenzugriff über eine Referenz weder eingeschränkt noch überwacht werden, und das Meta-System hatte keine Zugriffskontrolle: Jeder konnte Metaobjekte anheften und insbesondere auch entfernen.

Die zweite Möglichkeit ist, die SMOs aufbauend auf einem Ablaufsystem zu implementieren. Dies ist, zumindest in eingeschränkter Form, mit nahezu jedem objektorientierten Ablaufsystem möglich. Um ein SMO an eine Objektreferenz zu heften, implementiert man ein Stellvertreterobjekt (Proxy), das die gleiche Schnittstelle wie das Zielobjekt der Referenz besitzt. Dieses übermittelt Aufrufe zunächst an das zuständige SMO und ruft, sofern das SMO dies zuläßt, dann die gewünschte Methode am Zielobjekt auf.

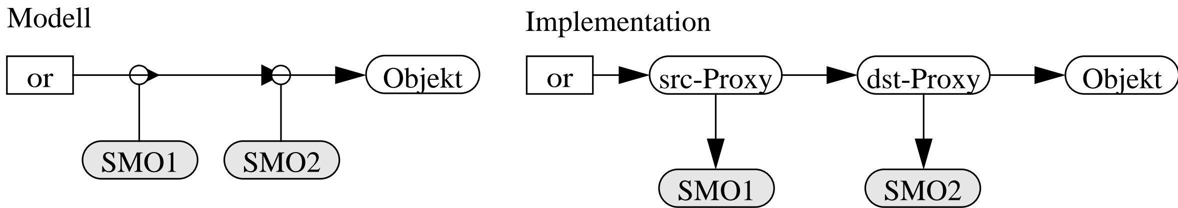

Abbildung 6.1 Implementation von SMOs durch Proxies

Abbildung 6.1 zeigt eine Objektreferenz mit einem quell- und einem zielorientiert angehefteten SMO. In der Implementation wird dies auf eine Referenz auf einen quellorientierten Proxy abgebildet, der wiederum eine Referenz auf ein SMO und auf den nächsten Proxy besitzt. Dieses Konzept wurde für den implementierten Prototyp gewählt und soll daher nun genauer dargestellt werden.

6.2 Eigenschaften der Proxy-Implementation

Die Implementation mit Proxies auf Programmiersprachenebene kann, die geeignete Basisplattform vorausgesetzt, ebenfalls das gesamte Sicherheitsmodell realisieren. Als Beispiel-Plattformen betrachten wir hier die Ablaufumgebungen bzw. Programmiersprachen Java, C++ und Smalltalk.

Folgende Eigenschaften sollen hier untersucht werden:

- Proxy-Fähigkeit
- Transparenz der Proxies
- Objektschutz

6.2.1 Proxy-Fähigkeit

Die Basisplattform muß Proxy-fähig sein, d.h. es muß möglich sein, zu einem Objekt ein Stellvertreterobjekt mit anderen Methodenimplementationen aber gleicher Schnittstelle zu erzeugen. Objektreferenzen auf dieses Objekt müssen statt Objektreferenzen auf das Ursprungsobjekt einsetzbar sein.

Dies ist bei vielen objektorientierten, typisierten Programmiersprachen problematisch, da oft keine oder nur eine begrenzte Unterscheidung zwischen Schnittstelle und Klasse erfolgt. Eine Proxy-Klasse definiert in solchen Systemen nur dann typkonforme Objekte, wenn sie von der Klasse des realen Objektes abgeleitet ist. Die Proxy-Klasse von der realen Klasse abzuleiten, widerspricht der Objektorientierung, bei der man die Ableitung als “ist-Spezialfall-von” Relation sieht. Außerdem birgt dieser Ansatz Probleme durch Vererbung der Instanzvariablen. Der Proxy würde dann ebenfalls die Instanzvariablen enthalten, die auch das Originalobjekt enthält. Der Proxy benötigt diese Instanzvariablen nicht, und in den meisten objektorientierten Sprachen lassen sich Instanzvariablenzugriffe nicht vollständig verhindern.

Daher läßt sich bei solchen Sprachen, zu denen beispielsweise C++ [Str92] und Java [Fla96] zählen, dieses Konzept nur eingeschränkt realisieren. Um Typkompatibilität unter Beibehaltung objektorientierter Designkriterien zu realisieren, muß man Schnittstellen definieren, die dann sowohl von der realen Klasse, als auch vom Proxy implementiert werden. Wenn dann ein SMO an eine Referenz geheftet wird, entsteht eine Referenz auf einen solchen Proxy, bei dem man alle Methoden, die durch eine Schnittstelle implementiert werden, aufrufen kann.

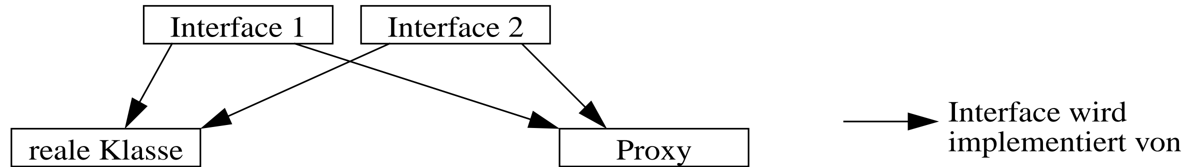

Abbildung 6.2 Realisierung von Proxies in typisierten Sprachen

In Abbildung 6.2 implementiert die reale Klasse zwei Schnittstellen. Diese realisiert auch der Proxy. Man kann über den Proxy daher alle Methoden aus diesen beiden Schnittstellen aufrufen. Java bietet dazu direkt Schnittstellen (Interfaces) an und verwendet dieses Verfahren beispielsweise bei Fernaufrufen (bei Java RMI). Bei C++ können abstrakte Klassen, die keine Implementation enthalten, als Interface angesehen werden.

Bei typlosen Sprachen wie Smalltalk [GoR89] hingegen läßt sich das Konzept vollständig implementieren. Der Proxy kann in einer völlig getrennten Vererbungshierarchie implementiert werden, da Smalltalk keine Typen kennt und Methodenaufrufe nur zur Laufzeit geprüft werden. Daher kann eine Proxy-Referenz überall dort verwendet werden, wo auch die echte Referenz gültig ist.

Bei Smalltalk besteht zusätzlich noch die Möglichkeit, Vorteile aus dem in Teilen vorhandenen Meta-System zu ziehen. Man braucht nur eine einzige Proxy-Klasse für alle realen Klassen zu implementieren, die die Methode “doesNotImplement” implementiert. Diese Methode wird durch das Laufzeitsystem automatisch bei jedem nicht-zustellbaren Methodenaufruf aktiviert und erhält alle Informationen wie Parameter und Zielmethode, so daß sie den Aufruf dann an das SMO bzw. das Zielobjekt weiterleiten kann.

6.2.2 Transparenz der Proxies

Die Proxies sollten transparent sein, der Programmierer sollte nicht merken, ob er eine Referenz auf einen Proxy oder auf ein reales Objekt besitzt.

Dies kann gar nicht vollständig zugesichert werden, da die Proxies mit den SMOs zusammen die Semantik der Referenz ändern. Beispielsweise kann ein SMO eine Ausnahme erzeugen. Bei einem Aufruf über eine reale Referenz kann jedoch (zumindest beim Transfer des Aufrufes über die Referenz) keine Ausnahme entstehen.

Wenn man von dieser prinzipbedingten Einschränkung absieht, könnte ein System diese Transparenz implementieren. Alle betrachteten Systeme erlauben allerdings zumindest in begrenztem Maße Zugriff auf Meta-Information wie beispielsweise die Information, aus welcher Klasse das Zielobjekt (reales Objekt oder Proxy) instantiiert wurde. Diese ist bei Proxy und realem Objekt verschieden. Ein weiteres Problem sind Eigenschaften von Referenzen wie das Feststellen der Gleichheit von Referenzen. Wenn man zwei Referenzen auf das gleiche Zielobjekt besitzt, sollten diese bei der Vergleichsoperation ("=" bzw. "==") auch identisch sein. Wenn dagegen die Referenzen über verschiedene Proxies auf das gleiche Zielobjekt verweisen, sind sie verschieden.

In keinem der betrachteten Systeme ist der Zugriff auf Instanzvariablen über einen Proxy möglich. Dies stellt in Smalltalk kein Problem dar, da hier auch über eine reale Referenz kein Zugriff auf Instanzvariablen möglich ist. In C++ und Java muß auf Instanzvariablenzugriffe in solchen Fällen verzichtet werden. Dies ist jedoch unproblematisch; aus Objektorientierungssicht verbietet die Objektkapselung sowieso direkte Instanzvariablenzugriffe.

6.2.3 Objektschutz

Generell muß das objektorientierte System Objektschutz realisieren, d.h. in dem System darf Zugriff auf Objekte nur über Objektreferenzen erlaubt sein und dies auch nur im Rahmen eines bestimmten Interfaces.

C++ bietet hier keinen Schutz. Durch Typumwandlung von Zeigern und Zeigerarithmetik kann auf alle Daten des Systems unkontrolliert zugegriffen werden.

Smalltalk besitzt zwar Objektkapselung, so daß unbeabsichtigte fehlerhafte Zugriffe vom System verweigert werden. Böswillige Zugriffe können jedoch nicht verhindert werden, da der Programmierer durch Metainteraktion direkt Einfluß auf das Laufzeitsystem nehmen und beispielsweise Klassen zur Laufzeit modifizieren kann.

Java bietet vollen Schutz. Durch Attribute kann festgelegt werden, in welchem Maße andere Klassen des Systems auf ein Objekt zugreifen können. Der so festgelegte Schutz wird durch Interpreter und Laufzeitsystem sichergestellt.

6.2.4 Auswahl des Zielsystems

Keines der betrachteten Systeme hat alle erforderlichen Eigenschaften. C++ schneidet am schlechtesten ab, Java und Smalltalk haben jeweils verschiedene Einschränkungen. Für die Prototypimplementation wurde Java gewählt, da mit Java zwar nicht die komplette Funktionalität zu realisieren ist, aber zumindest der Sicherheitsaspekt (Objektschutz) des Systems vollständig erfüllt wird.

6.3 Realisierung: Lokales Modell

Als Basis für die Prototypimplementation wurde das Java Developer Kit JDK-1.1 genommen. Da am System selbst keine Veränderungen vorgenommen wurden, ist die Implementation nicht stark versionsabhängig. Die Realisierung basiert auf Stellvertretern (Proxies), wie in den vorigen Abschnitten beschrieben.

Wenn ein SMO an eine Referenz gebunden werden soll, laufen folgende Schritte ab:

- Die geeignete Proxy-Klasse wird bestimmt. Für jede reale Klasse gibt es genau eine Proxy-Klasse.
- Wenn sich diese Klasse noch nicht im System befindet, muß sie dynamisch erzeugt werden. Dazu wird der Java-Quellcode generiert, übersetzt und die resultierende Klasse geladen. Die Proxy-Klasse implementiert alle Interfaces, die von der realen Klasse auch implementiert werden.
- Der Proxy wird instantiiert und erhält eine Referenz auf das reale Objekt und das SMO. Der Proxy merkt sich außerdem, ob er im Quell- oder im Zielmodus arbeiten muß.
- Die Referenz auf den Proxy wird an den Aufrufer zurückgegeben.

Die Implementation hat nahezu die gleiche Syntax wie im Konzept dargestellt (Listing 6.1). Um ein SMO an eine Referenz zu binden, muß die zu dem Objekt gehörige Klasse mindestens ein Interface realisieren: Nur die Methoden der Interfaces können über den Proxy aufgerufen werden. Beim Anheften von SMOs liefern die entsprechenden Methoden der SMOs eine Referenz vom Typ *Object* zurück, die dann manuell auf den gewünschten Typ umgewandelt werden muß. Die resultierende Referenz ist nun nicht mehr typkonform zu dem Typ der Originalreferenz (im Beispiel *List*). Sie realisiert aber alle Interfaces. Wie im Beispiel kann man dann zwischen den verschiedenen Interfaces Typumwandlungen vornehmen.

```
class List implements TestInterface, ListInterface {...}
ListInterface l = new List(...);                         // eine Ref. auf eine Liste
SecurityMeta s = new MetaExpire(new Date(1,7,99));       // Metaobjekt erzeugen
l = (ListInterface)s.dstAttachTo ( l );                  // Metaobjekt an Ref. heften
TestInterface i = (TestInterface) l;
```

Listing 6.1 Binden von SMOs im Prototyp

Für die Sicherheit der Proxies gegenüber der Anwendung wurde das Java-Sicherheitsmodell verwendet. Beispielsweise darf es der Anwendung nicht möglich sein, auf den internen Zustand der Proxies, wie z.B. die Referenz auf das reale Objekt und das SMO, zuzugreifen. Dies läßt sich durch das Java-Sicherheitsmodell und Paketkonzept erreichen. Alle Arten von Capability-SMOs lassen sich so implementieren.

Für die Implementation von Zugriffskontrolllisten und Identitäts-SMOs sind jedoch noch zusätzliche Informationen nötig, die zwischen den SMOs ausgetauscht werden müssen. Dazu wird eine Menge von Identitätsobjekten mit dem Aufruf mitgeführt, die an alle SMOs übergeben wird. Die SMOs können dann, wie im Konzept schon dargestellt, diese Menge verändern, um Identitätsinformation zur Verfügung zu stellen, oder prüfen, ob sich eine gewisse Identität in dieser Menge befindet, um Zugriffskontrolllisten zu implementieren. Für die Implementation der Identitäten gibt es mehrere Möglichkeiten: Wir unterscheiden symmetrische und asymmetrische Authentifizierung. Bei der symmetrischen Authentifizierung besteht eine Identität aus nur einem Objekt, das sowohl für das Setzen der Identitätsinformation vom Identitäts-SMO als auch für das Vergleichen der Identitätsinformation vom Zugriffskontroll-SMO verwendet wird. Diese Art der Authentifizierung entspricht einer Paßwort-Authentifizierung, bei der das Ziel (z.B. der Server) das Paßwort kennt. Man kann dabei nicht verhindern, daß das Ziel (der Server bzw. die Zugriffskontrollliste) ebenfalls (böswillig) die Identität einsetzen kann. Diese Strategie ist also nur zu empfehlen, wenn man dem Server bzw. der Zugriffskontrollliste am Ziel vertraut.

Besser ist daher in den meisten Fällen die asymmetrische Authentifizierung. Dabei gibt es zu einer Identität zwei Objekte. Ein Objekt stellt die Identität zur Verfügung (Quell-Identitätsobjekt) während das andere die Identität überprüfen kann (Ziel-Identitätsobjekt). Eine Zugriffsliste muß keine Kenntnis des Quell-Identitätsobjektes besitzen, um zu überprüfen ob ein Aufruf mit einer gewissen Identität durchgeführt wird. Die Methoden der Zugriffskontrollliste bekommen zwar eine Liste von Aufruferidentitäten, in der sich die Quell-Identitätsobjekte befinden, sie können jedoch lediglich mit Hilfe von Ziel-Identitätsobjekten prüfen, ob eine gewisse Identität in dieser Liste ist; sie können sich keinen Zugriff auf das Quell-Identitätsobjekt verschaffen. Abbildung 6.3 zeigt ein Beispiel: Auf der linken Seite sind symmetrische Identitäten: die Zugriffsliste benötigt Zugriff auf das Identitätsobjekt (oder mindestens auf eine exakte Kopie des Identitätsobjektes), das von dem Identitäts-SMO verwendet wird. Auf der rechten Seite wird dieser Zugriff nicht benötigt, für die beiden Aufgaben werden verschiedene Identitätsobjekte verwendet, die die gleiche Identität repräsentieren, aber jeweils nur entweder eine Identität bereitstellen oder prüfen können. Das Ziel-Identitätsobjekt kann man dann an andere, nicht-ver-

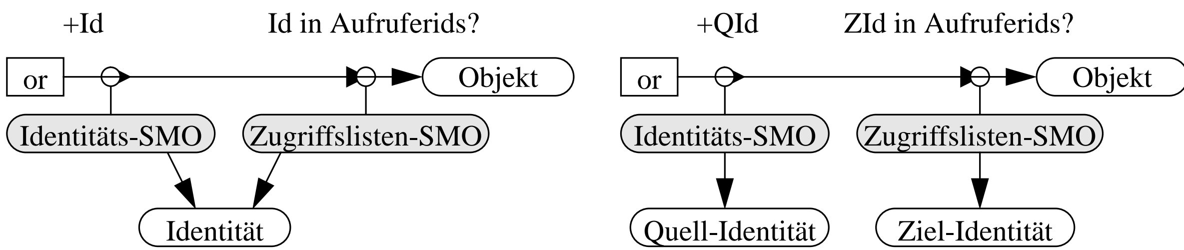

Abbildung 6.3 Symmetrische vs. asymmetrische Authentifizierung

trauenswürdige Applikationsteile weitergeben: Diese Applikationsteile können zwar prüfen, ob Aufrufe mit dieser Identität durchgeführt wurden, können aber nicht die Identität selbst verwenden.

6.4 Realisierung: Verteiltes Modell

Im vorigen Abschnitt wurde nur der lokale Fall betrachtet: Alle Objekte lagen auf dem gleichen Knoten bzw. in der gleichen Java-Maschine. Da jedes Objekt zumindest der Java-Maschine vertrauen muß, in der es sich befindet, kann man dort das Java-Sicherheitsmodell nutzen, um Schutz zu garantieren. Im verteilten Fall kann dies anders aussehen. Wenn man ein verteiltes Laufzeitsystem betrachtet, das von einer Instanz (Systemadministrator, Programmierer) zentral kontrolliert wird, und alle Komponenten vertrauenswürdig sind, kann man den lokalen Fall auf diese Situation übertragen. Im allgemeinen ist diese Situation jedoch nicht gegeben. Man möchte mit Objekten interagieren, die in anderen Java-Maschinen auf anderen Knoten laufen, ohne der Verbindung bzw. der Ziel-Java-Maschine vollständig vertrauen zu müssen. Wenn man einem Objekt traut, d.h. bereit ist, gewisse Daten an dieses zu übermitteln, möchte man sicher sein, daß nur genau dieses Objekt die Daten erhält. Die Identitätsinformation sollte zuverlässig sein; auch im verteilten Fall möchte man Identitätsinformation erhalten und sicher sein, daß diese nicht gefälscht ist.

Die Semantik aus Programmierersicht sollte die gleiche wie im lokalen Fall sein. Für den Applikationsprogrammierer und den Programmierer der SMOs sollte es also keinen Unterschied machen, ob eine Applikation lokal oder verteilt läuft. Bei der Implementation von SMOs wird dies nur teilweise möglich sein: Die SMOs müssen für den verteilten Fall zusätzliche Funktionalität bereitstellen.

Fernaufrufe zwischen Knoten wurden dazu neu implementiert. Zunächst wurde erwogen, das bei Java bereits vorhandene RMI als Basis zu nehmen. Dies erwies sich jedoch aus zwei Gründen als unbrauchbar: RMI ist nicht konfigurierbar und nicht transparent. Bei RMI kann nicht für jeden Aufruf separat festgelegt werden, welche Daten an den Aufgerufenen übermittelt werden sollen. Dies wäre aber für die Signierung von Aufrufen und die Verschlüsselung der Daten nötig. Außerdem können RMI-Methodenaufrufe nicht transparent verwendet werden. Wenn man eine verteilte Anwendung mit RMI programmieren möchte, muß man vorher überlegen, welche Objekte via Netzwerk zugänglich sein sollen. Man muß in die entsprechenden Klassen Unterstützung dafür implementieren (beispielsweise die Schnittstelle `Remote`) und entsprechende Ausnahmen (Exceptions), die in Java Teil der Schnittstelle sind, bei allen Methoden angeben (`RemoteException`). Dies führt dazu, daß man lokalen Code nicht für verteilte Anwendungen wiederverwenden kann. Ein Beispiel ist die Klasse Vector. Diese ruft von den Objekten die Methode `equals` auf, die nicht über Netz aufgerufen werden kann, da sie keine `RemoteException` erzeugen darf. Da es hier gerade um Wiederverwendung von Code geht, ist es wichtig, vorhandene Systemklassen einsetzen zu können, um zu prüfen, ob Klassen ohne entsprechende Unterstützung mit Sicherheitsmetaobjekten zusammen verwendet werden können.

6.4.1 Vertrauensbeziehungen

In Abbildung 6.4 sind die Vertrauensbeziehungen dargestellt. Jedes Objekt muß zumindest dem Laufzeitsystem vertrauen, in dem es sich befindet. Wenn man eine Objektreferenz besitzt und über diese einen Methodenaufruf ausführt, muß man der Objektreferenz und dem Zielobjekt insoweit vertrauen, wie man sich auf die korrekte Ausführung des Methodenaufrufs verläßt und ihr Informationen (Objektreferenzen, Parameterwerte) anvertraut. Wegen der Transitivität der Vertrauensbeziehungen muß das aufrufende Objekt dann auch dem Zielknoten im Rahmen des korrekten Arbeitens mit Parametern vertrauen. Gleiches gilt für Rückgabewerte.

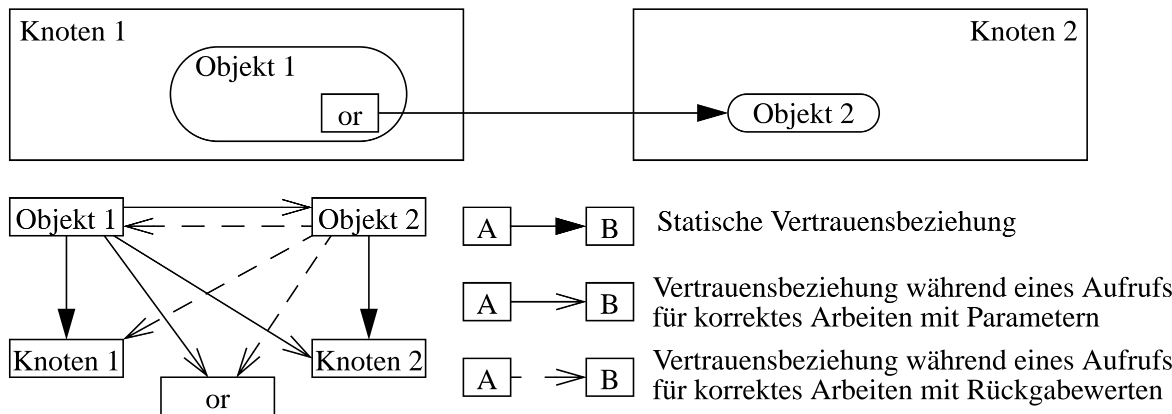

Abbildung 6.4 Vertrauensbeziehungen

6.4.2 Einheiten des verteilten Systems und Darstellung

Für die Implementation des verteilten Modells werden Knoten definiert. Ein Knoten entspricht einem Java-Interpreter. Wenn man auf einem realen Rechnerknoten mehrere Java-Interpreter startet und in das System einbezieht, stellen diese mehrere Knoten in der hier verwendeten Abstraktion dar, da man möglicherweise nicht jedem dieser Interpreter gleich stark vertraut (beispielsweise können die Interpreter von verschiedenen Benutzern gestartet werden). Weiterhin enthält das System Objekte, die in diesem Modell den Knoten fest zugeordnet werden. Für die Authentifizierung müssen Identitäten definiert werden, die gemäß Abschnitt 6.3 in Quell-Identität und Ziel-Identität unterteilt werden.

Die Komponenten müssen nun im System auf eindeutige und zuverlässige Weise identifiziert werden. Dazu werden geheime und öffentliche Schüssel verwendet. Ein Knoten besitzt einen geheimen Schlüssel, den er zur Authentifizierung verwenden kann. Der dazugehörende öffentliche Schlüssel kann dann verwendet werden, um sicherzustellen, daß man mit diesem Knoten kommuniziert.

Jedes Objekt im System besitzt eine weltweit eindeutige Objektidentität, die aus der Zieladresse des Knotens und einer nicht-ratbaren Objektnummer besteht (dies ist eine vom Knoten zufällig gewählte Zahl aus einem großen Zahlenbereich). Eine Objektreferenz besteht aus der Objektidentität plus dem öffentlichen Schlüssel des Zielknotens, auf dem sich das Objekt befindet.

Bei einem Methodenaufruf wird nun ein sicherer, verschlüsselter Kanal von dem Quellknoten zum Zielknoten erzeugt (Abbildung 6.5). Dazu werden zunächst die öffentlichen Schlüssel getauscht. Damit kann der Klient prüfen, ob er mit dem richtigen Server verbunden ist (er kennt ja dessen Schlüssel bereits), und der Server erhält den Schlüssel des Klienten, falls die Verbindung abbricht und der Server später die Verbindung neu erstellen möchte. Dann wird ein symmetrischer Schlüssel ausgehandelt und zur schnellen Verschlüsselung von Daten über den Kanal verwendet. Dies geschieht mit Hilfe der privaten und öffentlichen Schlüssel der beiden Teilnehmer, so daß dabei gleichzeitig die Authentizität des Partners verifiziert wird (d.h. jeweilige Kenntnis des privaten Schlüssels des anderen wird geprüft). Über die entstehende verschlüsselte Verbindung wird dann der Methodenaufruf abgewickelt.

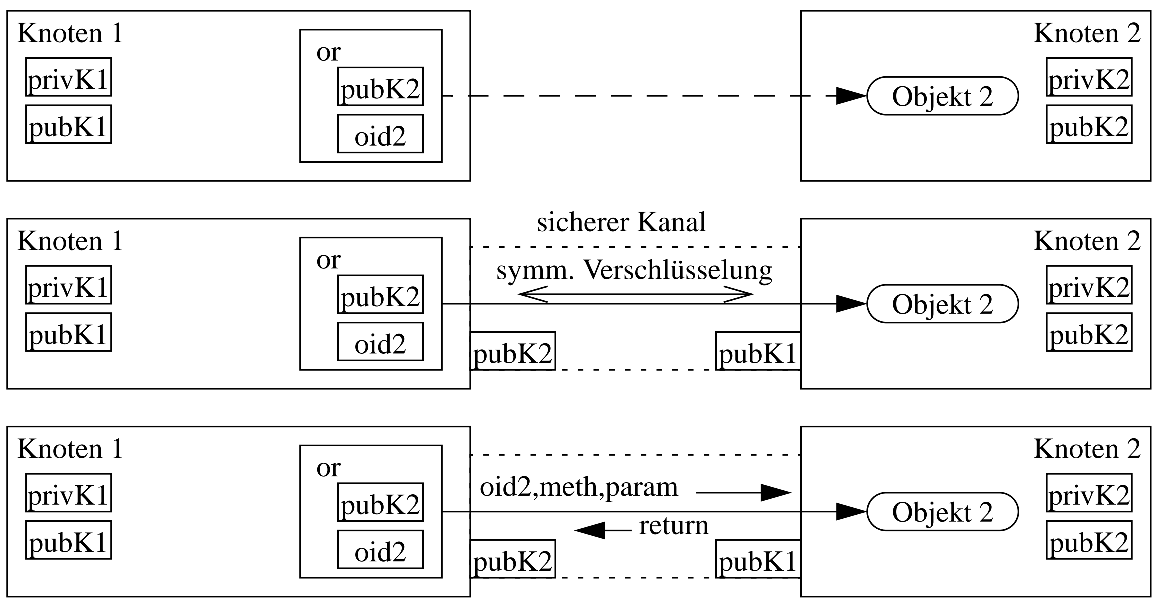

Abbildung 6.5 Methodenaufruf

Alle Aufrufe zwischen den beiden Knoten werden ab Aufbau der verschlüsselten Verbindung über diese abgewickelt. Die Verbindung kann beliebig auf- und abgebaut werden. Da der Aufbau relativ lange dauert, ist es sinnvoll, die Verbindung nicht sofort nach Ende eines Aufrufes abzubauen, da bei vielen Applikationen mehrere Aufrufe hintereinander an das gleiche Zielobjekt oder zumindest an Objekte auf dem gleichen Knoten erfolgen. Um Ressourcen zu sparen, bzw. bei Netzwerkfehlern kann es aber sinnvoll oder sogar nötig sein, die Verbindung abzubauen. Bei Bedarf wird sie dann automatisch von einem der beiden Teilnehmer wieder aufgebaut.

Die bisher betrachteten Objektreferenzen implementieren verschlüsselte Aufrufe über ein Netzwerk. Sie enthalten noch keine SMOs, d.h. sie sind Capabilities für das Gesamtinterface des Zielobjektes. Die Aufrufe sind verschlüsselt, d.h. nur der Empfänger des Aufrufes bekommt übergebene Objektreferenzen, Rückgabewerte erhält nur der Aufrufer. Das implementierte Modell ist voll objektorientiert: Ein Methodenaufruf an einem Zielobjekt ist nur demjenigen möglich, der eine Objektreferenz auf das Zielobjekt besitzt.

Was passiert nun mit Referenzen, an die ein SMO angeheftet ist? Wenn man die Implementation von SMO-Anheftungen betrachtet, sieht man, daß zumindest in einfachen Fällen, d.h. mit reinen Capability-SMOs, keine zusätzlichen Mechanismen nötig sind.

Abbildung 6.6 Capability-SMO-behaftete, rechnerübergreifende Referenzen

Abbildung 6.6 zeigt einen solchen Fall. An eine knotenübergreifende Referenz sind zwei SMOs geheftet. Für die SMOs ist dies jedoch völlig transparent: Sie verweisen einfach auf den Fernaufrufstellvertreter (remote-Proxy), bzw. auf sie wird von der Vermittlungseinheit für den Methodenaufruf (remote-Skeleton) verwiesen. Komplizierter stellt sich dieser Fall bei Zugriffskontroll-SMOs dar, weil diese zusätzlich Identitäts-Informationen benötigen. Diese Informationen können nun nicht einfach über das Netz übergeben werden, zumindest nicht, wenn die Knoten sich nicht gegenseitig trauen. Man müßte nämlich entweder Identitätsinformationen übergeben, die eindeutig beweisen, daß der Aufrufer tatsächlich über die Identitätsinformationen des angegebenen Aufrufers verfügt. Dazu könnte man beispielsweise ein Paßwort mit übergeben. Dann hat man aber keine Möglichkeit, zu verhindern, daß der Zielknoten dieses Paßwort nicht nur prüft, sondern auch selbst unberechtigt für andere Aufrufe zur Authentifizierung verwendet. Oder man müßte Identitätsinformationen übergeben, die nicht beweisen, daß der Aufrufer über die Identitätsinformation des angegebenen Aufrufers verfügt. Dazu könnte man beispielsweise den Namen des Aufrufers übergeben. Dann kann jedoch der Zielknoten nicht prüfen, ob der Aufrufer die Identität nur vortäuscht.

Hier wurde daher ein anderer Weg beschritten: Knotenübergreifende Aufrufe können von den Identitäts-SMOs signiert werden. Dazu werden asymmetrische Identitäten verwendet. Die Identität, die zur Authentifizierung verwendet wird, enthält einen privaten Schlüssel, während die Identität, die zur Prüfung verwendet wird, den zugehörigen öffentlichen Schlüssel enthält. Wenn nun ein Aufruf über das Netzwerk erfolgt, werden die Authentifizierungs-Identitäten automatisch aktiviert, und jede dieser Identitäten kann eine Signatur aus dem Aufruf (Parameter, Methode, Zielobjekt) generieren, die die zugehörige Zielidentität verifizieren kann.

Der Zielknoten ist dann sicher, daß der Quellknoten über die Authentifizierungs-Identität verfügt, kann sie jedoch selbst nicht unberechtigt nutzen.

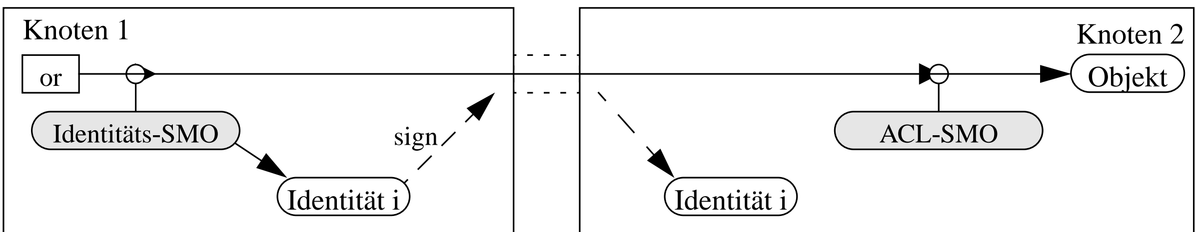

Abbildung 6.7 Authentifizierung von rechnerübergreifenden Aufrufen

Abbildung 6.7 zeigt ein Beispiel. Ein an eine Objektreferenz geheftetes Identitäts-SMO stellt eine Identität zur Verfügung. An der Stelle, wo die Knotengrenze überschritten wird, muß das Laufzeitsystem nun diese Identität dem anderen Knoten mit übergeben. Falls die Identität (wie in unserem Beispiel) nicht auf den anderen Knoten kopiert werden, sondern nur den Aufruf signieren soll, bekommt sie den Bytestrom, der den Aufruf enthält und kann diesem eine Signatur hinzufügen. Auf der anderen Seite wird das der Identität entsprechende Pendant automatisch erzeugt. Dieses prüft die Signatur und stellt dann – sofern dies erfolgreich war – die Identität für den weiteren Verlauf des Aufrufs bereit. Diese ist zwar eine Quell-Identität, kann aber keine weiteren Aufrufe signieren, da sie keine Kenntnis des privaten Schlüssels der Identität besitzt. Sie kann also vom Zielknoten nicht für andere Aufrufe mißbraucht werden.

6.4.3 Initiale Referenzen

Die Probleme mit initialen Vertrauensbeziehungen, die in Abschnitt 2.6 behandelt wurden, entstehen auch in diesem System. Die Teilnehmer des verteilten Systems müssen initial miteinander Verbindung aufnehmen. Wenn man beispielsweise auf zwei Knoten jeweils Java-Interpreter startet, muß man auf einem der beiden Interpreter initial eine Referenz auf ein Objekt in dem anderen Interpreter erzeugen können, damit Objekte in den verschiedenen Interpretern miteinander kommunizieren können. Dieses Objekt kann ein Nameserver-Objekt sein, das dann weitere Objektreferenzen verwaltet.

Eine ähnliche Problematik wie das Problem der initialen Schlüsselverteilung in Abschnitt 2.6 ergibt sich auch hier: Jeder Interpreter (bzw. die Objekte, die in dem Interpreter laufen) muß gewisse Kenntnisse über andere Interpreter besitzen, sonst kann er keine sichere Kommunikation aufbauen. Man kann die zentrale Lösung wählen: Jeder Interpreter besitzt initial nur eine Referenz (die den öffentlichen Schlüssel des Ziels umfaßt) auf einen zentralen Nameserver des Systems. Dieser besitzt dann Referenzen auf die verschiedenen Komponenten. Oder man kann die dezentrale Lösung wählen: Jeder Interpreter besitzt je nach Bedarf initiale Referenzen auf andere Objekte. Bei beiden Lösungen müssen initial bereits Referenzen (öffentliche Schlüssel) vorhanden sein, die außerhalb dieses Systems ausgetauscht werden. Für den Prototyp wurde lediglich eine interpreterlokale Schlüsselverwaltung implementiert: Jeder Interpreter lädt initial Schlüssel aus einer Datei. Diese Einschränkung betrifft aber nur die initialen Objektreferenzen: Bei weiterer Kommunikation zwischen den Teilnehmern werden automatisch dezentral Schlüssel ausgetauscht.

6.4.4 Implementierte Beispiele

Die in den vorigen Abschnitten beschriebene Implementation der SMO-Basismechanismen kann alle Arten von SMOs implementieren: Grenz-SMOs (für virtuelle Domänen), Zugriffslisten-SMOs und Capability-SMOs. Es wurden im Rahmen dieser Arbeit mehrere Beispiele für Transitivität von Capabilities und Zugriffskontrolllisten implementiert, und zwar sowohl lokal als auch verteilt, d.h. mit privaten und öffentlichen Schlüsseln für Identitäten und Knoten. Außerdem wurden die beiden betrachteten Szenarien "disjunkte Interaktion" (Abschnitt 5.2.7) und "hierarchische Domänen" (Abschnitt 5.2.8) implementiert, und die korrekte Funktionalität wurde verifiziert [Ger98].

6.5 Effizienz

Der Prototyp wurde nicht auf Effizienz optimiert. Dennoch wollen wir einige Geschwindigkeits-Meßwerte betrachten, um einen Eindruck von der Größenordnung des Mehraufwands durch SMOs zu bekommen und konzeptionelle Optimierungen zu diskutieren.

Dazu wurde das Java Developer Kit 1.1 auf einem Sun-Enterprise-4000 Server verwendet. Wir betrachten zunächst lokale Laufzeiten, also Laufzeit bei Objektinteraktion innerhalb eines Java-Interpreters (Tabelle 6.1).

leerer Methodenaufruf (ein Parameter, ein Rückgabewert)	0,16µs
Anheftung eines SMOs an eine Referenz	300 bis 610µs
Aufruf über Referenz mit angeheftetem SMO (ein Parameter, ein Rückgabewert)	12µs
Aufruf über Referenz mit 10 angehefteten SMOs (ein Parameter, ein Rückgabewert)	52µs

Tabelle 6.1 Meßwerte für lokale Interaktion

Ein Methodenaufruf dauert auf dem verwendeten System 0,16µs, ein Aufruf über eine Referenz mit angeheftetem SMO dauert hingegen 12µs, das ist das 75-fache eines normalen Methodenaufrufes. Die Verwendung von SMOs ist also nicht billig. Man könnte noch optimieren, indem man Methoden als *final* kennzeichnet, dann kann der Compiler Optimierungen vornehmen – allerdings verliert man dann die freie Erweiterbarkeit. Wenn mehr als ein SMO an eine Referenz geheftet ist, wird die Laufzeit pro SMO kürzer. Da dieser Fall oft eintritt, wurden dafür spezielle Optimierungen implementiert. Bei 10 angehefteten SMOs, was für reale Anwendungen schon eine sehr große Anzahl darstellt, benötigt der Prototyp dann nur noch 5,2µs pro SMO. Die Anheftung eines SMOs selbst ist relativ ineffizient, je nach Komplexität der Bestimmung des zur Anheftung nötigen Proxies dauert sie zwischen 300 und 610µs.

Betrachten wir nun verteilte Interaktion, also Interaktion zwischen Objekten, die in verschiedenen Java-Interpretern laufen (Tabelle 6.2). Ein leerer RMI-Methodenaufruf benötigt 2,3ms, ein leerer Fernaufruf mit unserem Prototypen benötigt 100ms, d.h. er ist um mehr als den Faktor 40 langsamer. Bei unserem Prototyp gehen Parameterübergabe und Rückgabewerte in der Meßungenauigkeit unter, lediglich wenn Objektreferenzen übergeben werden wird der Aufruf merklich langsamer, da dann Stellvertreterobjekte erzeugt werden müssen. Die im Vergleich zu RMI geringe Geschwindigkeit ist also durch den Grundaufwand des Aufrufes zu begründen: Verschlüsselung der Aufrufdaten und Rückgabewerte, Signierung der Aufrufdaten und Rückgabewerte und Test der Signaturen. Durch Verwendung einer schnelleren Verschlüsselungsbibliothek oder anderer Algorithmen ließe sich auch hier noch Zeit einsparen. Die Geschwindigkeit der Verschlüsselung selbst hängt natürlich auch von der Menge der Daten ab: Bei 100kB übergebenen Daten benötigt der Prototyp 579ms, man muß also ca. 6ms pro Kilobyte Daten einkalkulieren.

leerer RMI-Methodenaufruf	2,3ms
Aufruf über Remote SMO-Referenz ohne Parameter und Rückgabewerte	100ms
Aufruf über Remote SMO-Referenz mit String-Parameter und String-Rückgabewert	100ms
Aufruf über Remote SMO-Referenz mit 3 Objektreferenz-Parametern und Objektreferenz-Rückgabewert	140ms
Aufruf über Remote SMO-Referenz mit Parameterübergabe von 100kB Daten	579ms
Aufruf über Remote SMO-Referenz mit Identität auf der Quellseite und Zugriffsliste auf der Zielseite	284ms
Aufruf über Remote SMO-Referenz mit 2 Identitäten auf der Quellseite und Zugriffsliste auf der Zielseite	459ms

Tabelle 6.2 Meßwerte für verteilte Interaktion

Für die bisherigen Messungen wurde kein asymmetrischer Verschlüsselungsalgorithmus benötigt: Die initiale Kontaktaufnahme wurde nicht mitgemessen, da diese nur einmal erfolgen muß. Die Signierung der Daten erfolgt mit einer Hash-Funktion und der Übertragung des Ergebnisses in (symmetrisch) verschlüsselter Form. Wenn jedoch Identitäten verwendet werden, benötigt man asymmetrische Verschlüsselung, und zwar muß pro Identität eine asymmetrische Verschlüsselung erfolgen. Diese ist relativ langsam – eine Identität benötigt zur Signierung eines Aufrufes ca. 180ms. Hier ließe sich konzeptionell optimieren: Wenn ein Interpreter einmal einen Aufruf mit einer bestimmten Identität signiert hat, bräuchte er bei weiteren Aufrufen dann lediglich anzugeben, daß der Aufruf von der entsprechenden Identität ausgeführt wird. Der Zielinterpreter weiß, daß der Quellinterpreter über die Identitätsinformationen verfügt und könnte daher auf einen Nachweis durch Signatur verzichten.

6.6 Einschränkungen der Implementation

Die Implementation ist lediglich prototypisch. Daher hat sie einige Einschränkungen, die zum Teil systembedingt sind (der Java Interpreter hat gewisse Einschränkungen) und zum Teil wegen Fokussierung auf den Sicherheitsaspekt erfolgten.

6.6.1 Lokales Modell

Das lokale Modell hat mehrere Einschränkungen, die sich aufgrund der Limitierung von Java ergeben.

Eine Einschränkung resultiert aus der nur begrenzten Proxy-Fähigkeit: In Java können nur für Interfaces Proxies erzeugt werden. D.h. über eine Referenz mit angeheftetem SMO oder über eine Referenz auf ein Objekt auf einem anderen Knoten kann man nur Methoden aufrufen, die in einem von dem realen Objekt implementierten Interface deklariert sind. Man kann daher auch keine Instanzvariablen über einen Proxy ansprechen. Ferner lassen sich für Felder keine Proxies implementieren. Feldübergabe über Proxies und Anheften von SMOs an Felder konnte daher nicht realisiert werden.

Die globalen Variablen (statische Variablen) in Java stehen der virtuellen Domäneneigenschaft entgegen. Das Ansprechen einer globalen Variablen müßte auf einen Methodenaufruf an einen Nameserver abgebildet werden. Diese Möglichkeit gibt es in Java nicht. Man muß daher – wenn man virtuelle Domänen verwenden möchte – auf Benutzung globaler Variablen verzichten.

6.6.2 Verteiltes Modell

Bei einem Methodenaufruf über das Netzwerk treten Probleme wie Koordinierungsprobleme, Verklemmungen, verwaiste Aufrufe, Aufräumen nicht-referenzierter Objekte und die Frage der Semantik (at least once, exactly once) auf. Da diese Probleme nicht Teil der vorliegenden Arbeit und orthogonal zur Sicherheitsproblematik sind, wurden diese nicht bei der Prototypimplementation beachtet. Objektmigration und Knotenausfall wurden ebenfalls nicht berücksichtigt.

Bei Authentifizierung und Überprüfung der Identitäten wurde keine Schlüsselverwaltung und Verteilung implementiert. Außerdem kann implementationsbedingt nur Authentifizierung zwischen zwei beteiligten Knoten funktionieren. Falls eine Referenz über mehr als zwei Knoten geht, d.h. zwischen Quelle einer Referenz und dem Zielobjekt ein SMO angeheftet ist, das sich auf einem dritten Knoten befindet, funktionieren die Identitäten nicht transitiv über diesen Knoten hinweg.

6.7 Bewertung

Die Implementation ist lediglich prototypisch. Dennoch zeigt die Untersuchung in den vorigen Abschnitten, daß sie durchaus für reale Anwendungen brauchbar ist. Die Einschränkungen der Implementation umfassen im wesentlichen nur Restriktionen, die bei der Programmierung von

verteilten Java-Anwendungen sowieso beachtet werden müssen, wie beispielsweise die Schnittstellenproblematik von Java. Die Effizienz ist für die meisten Anwendungen ausreichend. Aufrufe über SMO-behaftete Referenzen sind zwar um einiges langsamer als Aufrufe über Referenzen ohne SMOs, jedoch wird auch nur ein sehr geringer Teil von Referenzen mit SMOs geschützt sein – innerhalb eines Moduls werden kaum geschützte Referenzen nötig sein. Die Effizienz verringert sich bei einfachem Schutz selbst bei elementaren Datenstrukturen (bei denen sich durch die geringe Komplexität bereits kleiner Mehraufwand bemerkbar macht) kaum: Der Aufruf einer Suchfunktion an eine Liste (Vector-Objekt) mit 10 Zeichenketteneinträgen benötigt beispielsweise statt 60µs dann 72µs, d.h. er wird um nur 17% langsamer. Je komplexer die Strukturen sind, die geschützt werden, desto weniger macht sich der Mehraufwand bemerkbar. Lediglich das Anheften von SMOs benötigt relativ viel Rechenzeit. Es müssen für jede Anheftung Proxy-Klassen bestimmt, Proxy-Objekte erstellt und initialisiert werden. Dies dauert 300 bis 610µs, es fällt also durchaus im Vergleich zu den oben beschriebenen primitiven Operationen stark ins Gewicht. Die Effizienz der Anheftung läßt sich jedoch nicht mit anderen Systemen vergleichen; die durch die Anheftung erreichte Semantik wird von anderen Systemen überhaupt nicht zur Verfügung gestellt. Man könnte das Anheften optimieren, indem man etwa einen Cache für Proxy-Klassen und Proxy-Objekte einrichten würde, so daß die Proxy-Erstellung schneller ginge, oder direkt die Erstellung von Proxies in die virtuelle Maschine einbauen. Ansonsten muß man diesen Mehraufwand akzeptieren. Wenn man keine extremen Effizienzanforderungen hat, ist die Geschwindigkeit auch dann immer noch ausreichend. Wenn man beispielsweise, wie in Abschnitt 4.3.3, eine Liste mit transitiven Capabilities schützt, wird bei jedem Zugriff ein SMO angeheftet. Wenn ein Applikationsteil sich durch Listenzugriff Referenzen auf 1000 Objekte der Liste besorgt, dauert dies im Mittel kürzer als eine halbe Sekunde, was für viele Applikationen sicherlich schnell genug ist.

Bei verteilter Programmierung ist der Methodenaufruf über den SMO-Mechanismus ziemlich langsam; dies liegt jedoch nicht an den SMOs, sondern an der Verschlüsselung. Man könnte (ähnlich wie bei Corba) Domänen definieren, in denen Aufrufe unverschlüsselt erfolgen. Wenn man jedoch Verschlüsselung für Fernaufrufe benötigt, besteht keine Möglichkeit, den Mehraufwand zu umgehen; jedes andere System ist ähnlich langsam – nur durch Wahl und Implementation des Verschlüsselungsalgorithmus läßt sich noch optimieren.

6.8 Zusammenfassung

Der Prototyp implementiert mit wenigen Einschränkungen das gesamte Sicherheitsmodell, und zwar sowohl lokal als auch verteilt. Man erreicht eine nahezu völlige Transparenz für den Benutzer: Er muß sich weder um Verteilung noch um besondere Authentifizierungsmechanismen oder Zugriffsschutzmechanismen für den verteilten Fall kümmern – er kann für seine SMOs einfach authentifizierungsfähige Identitäten oder direkt entsprechende SMOs aus einer Klassenbibliothek verwenden. Alle Komponenten des Modells wie SMOs und Identitäten lassen sich unabhängig voneinander auf abstrakter Ebene implementieren und können später beliebig kombiniert werden.

Die Effizienz des Prototyps ist für viele Applikationen ausreichend. Bei Verwendung von optimierten Verschlüsselungsbibliotheken und Optimierung der SMO-Klassen ist allerdings noch eine erhebliche Effizienzsteigerung möglich.

Die wenigen Einschränkungen des Prototyps beruhen überwiegend auf dem starren und teilweise inkonsistenten Typkonzept sowie der unvollkommenen Konfigurierbarkeit von Java. Bei einem entsprechend modifizierten Java-System ließe sich die Prototypimplementation vervollständigen. Durch die dann entstehende Inkompatibilität ginge dabei aber die Plattformunabhängigkeit verloren.

7 Zusammenfassung und Ausblick

Mit der Nutzung der Objektorientierung in verteilten Systemen entstanden neue, zunächst unlösbar scheinende Probleme: Objektorientierung beinhaltet Abstraktion von den konkret ablaufenden Aktionen im System, Sicherheit hingegen bedeutet, genau zu wissen, was im System abläuft, um erlaubte und unerlaubte Aktionen unterscheiden zu können. In dieser Arbeit wurde gezeigt, daß man die Objektorientierung auch als Vorteil für die Konfiguration der Sicherheit nutzen kann: Auf dem gleichen, hohen Abstraktionsniveau, auf dem die Objekte interagieren, kann auch die Sicherheitsstrategie festgelegt werden – und dies völlig getrennt von der Applikation. Zusätzlich läßt sich die Objektorientierung für die Definition der Sicherheitsstrategie nutzen: Durch Vererbung können Sicherheitsstrategien modifiziert werden; gleiche Strategieteile können wiederverwendet werden.

7.1 Zusammenfassung der Arbeit

In Kapitel 2 wurden die Grundkonzepte von Sicherheitsmodellen erläutert. Die auch heute noch verwendeten Basismechanismen wie Zugriffsmatrix, Zugriffskontrollliste und Capabilities wurden erklärt. Zusätzlich wurden rollenbasierte Zugriffskontrolle und das Bell-La Padula-Modell dargestellt. Kapitel 3 behandelte darauf aufbauend reale verteilte Systeme. Hier standen besonders Systeme im Vordergrund, die spezielle Maßnahmen für Sicherheit anbieten wie Hydra und Amoeba, und objektorientierte oder objektbasierte Systeme wie Corba und DCOM. Die von solchen Systemen angebotenen Mechanismen (Capabilities, Zugriffslisten, Revokation, Expiration, Zugriffsbeschränkung, etc.) wurden herausgearbeitet. Der Schutz läßt sich bei den betrachteten objektbasierten Systemen pro Objektreferenz, pro Zielobjekt oder pro Domäne separat konfigurieren. Es stellte sich bei der Betrachtung der Systeme jedoch heraus, daß diese Verfahren für objektorientierte Applikationen nicht geeignet sind. Objekt- und objektreferenzbasierte Festlegung ist wegen der großen Zahl an Objekten in einer objektorientierten Anwendung äußerst schwierig. Domänenbasierte Festlegung ist wiederum zu grobgranular.

In Kapitel 4 wurde das in dieser Arbeit entwickelte Sicherheitsmodell vorgestellt. Sicherheitsmetaobjekte realisieren die klassischen Schutzmechanismen Capabilities und Zugriffslisten. Zusätzlich können sie diese Mechanismen auch transitiv realisieren. Dies ist für objektorientier-

te Programmierung entscheidend, da hierdurch Verbreitungskontrolle von Capabilities und transitive Zugriffslisten implementiert werden können – Strategien, die sich mit keinem der klassischen Systeme realisieren lassen. Mit Sicherheitsmetaobjekten läßt sich jedoch nicht nur Zugriffsschutz implementieren. Man kann zusätzlich die Wahl der Identität, mit der Aufrufe durchgeführt werden, realisieren. Dies erlaubt die Strategie "rollenbasierte Identitäten", die verschiedene Identitäten auf Objektreferenz-Basis zur Interaktion mit verschiedenen Kommunikationspartnern verwendet. Das Problem des böswilligen Unterschiebens von Referenzen, das mit keinem herkömmlichen Sicherheitsmodell in den Griff zu bekommen ist, kann hiermit auf generische Weise gelöst werden. Um die Flexibilität und Vollständigkeit des Sicherheitsmodells unter Beweis zu stellen, wurde gezeigt, daß man alle für objektorientierte Systeme relevanten, in anderen Systemen vorhandenen Sicherheitsstrategien mit Sicherheitsmetaobjekten realisieren kann.

Das entstandene Modell besitzt als Basismechanismus nur die Sicherheitsmetaobjekte. Dadurch war eine Formalisierung von Teilaspekten verhältnismäßig einfach. In Kapitel 5 wurden die Reduktion von redundanten SMO-Anheftungen und die rollenbasierten Identitäten formalisiert. Für die Reduktion von redundanten SMO-Anheftungen wurden die SMOs kategorisiert. Dann wurden für die verschiedenen Kategorien von SMOs Reduktionsgesetze formuliert und diese bewiesen. Die Formalisierung von rollenbasierten Identitäten erforderte zunächst die Definition der statischen Eigenschaften des Systems: die Definition von SMOs, virtuellen Domänen und SMO-Anheftungen. Dann wurde ein momentaner Zustand des Systems und der Zustandsübergang mittels Methodenaufrufen formalisiert. Es mußte nun zunächst bewiesen werden, daß das entstehende System konsistent ist, daß der Zustandsübergang also beispielsweise nicht die virtuellen Domänen zerstört. Dann wurden die in Kapitel 4 vorgestellten Beispiele formalisiert und die dort erwähnten Sicherheitseigenschaften formal nachgewiesen. Dabei wurden die Beispiele verallgemeinert, so daß die Beweise für viele Konfigurationen rollenbasierter Identitäten gültig sind.

Die Frage nach der Implementierbarkeit des Modells behandelte Kapitel 6. Es wurde ein Prototyp in Java implementiert, der nahezu das komplette Modell realisiert, und zwar lokal durch Verwendung der in Java vorhandenen, auf Objektorientierung basierenden Sicherheitsmechanismen und verteilt durch Verwendung von Kryptographie. Sicherheitsstrategie und Applikationscode konnten auch in der Implementation nahezu vollständig (entsprechend dem Modell) getrennt werden. Es zeigte sich, daß sogar die Verteilung orthogonal zu Applikationscode und Sicherheitsstrategie erfolgen kann. Nicht einmal die Sicherheitsmetaobjekte, die Authentifizierung mit Identitäten implementieren, brauchen zu wissen, ob ein Aufruf lokal abgehandelt wird (und damit die Authentifizierung über die virtuelle Maschine erfolgt) oder über Netzwerk ein Objekt auf einem anderen Rechner anspricht (dann erfolgt die Authentifizierung mittels Signierung des Aufrufs).

Das Ziel der vorliegenden Arbeit war die Auflösung des scheinbaren Gegensatzes zwischen Objektorientierung und Sicherheit. In klassischen verteilten Systemen wird bei sicherheitskritischen Applikationen durch die Verwendung von Objektorientierung der Applikationscode einfacher, jedoch die Sicherheitsstrategie komplexer. In dieser Arbeit wurde ein Sicherheitsmodell entwickelt, das diese Anomalie nicht besitzt. Es erlaubt vielmehr die Konfiguration der Sicher-

heit auf der gleichen hohen Abstraktionsebene, auf der die Objekte miteinander interagieren. Ein Anwendungsentwickler oder -administrator kann, ohne die Interna der Applikationsklassen zu kennen, Sicherheitsmetaobjekte konfigurieren, die den für seine Bedürfnisse nötigen Schutz realisieren. Viele in objektorientierten Systemen oft auftretende Situationen lassen sich durch generische Sicherheitsmetaobjekte behandeln. Diese können in Klassenbibliotheken abgelegt werden, so daß der Implementationsaufwand für die Sicherheitsstrategie auf ein Minimum reduziert werden kann.

7.2 Weiterführende Arbeiten

Einige Aspekte des entwickelten Sicherheitsmodells wurden im Rahmen dieser Arbeit nicht oder nur teilweise betrachtet. Sie sollen hier kurz dargestellt werden, um weiterführende oder auf dieser Arbeit aufbauende Untersuchungen zu motivieren.

- Die Konfiguration der Sicherheitsstrategie sollte völlig unabhängig von der Applikation erfolgen können. Dazu könnte man eine Registratur verwenden, die verschiedenen Applikationsteilen oder Applikationen SMOs zuordnet. Es entsteht das Problem der Zuordnung von SMOs auf Initialwerte der Applikation: Wie konfiguriert man in der Registratur ein SMO für bestimmte Referenzen?
- Identitäten wurden nicht genauer definiert. Es wurden zwar Szenarien angegeben, wo Identitäten beispielsweise mit Benutzern identifiziert wurden. Wie man jedoch am besten eine Benutzerverwaltung implementieren kann wurde nicht betrachtet. Man könnte natürlich eine klassische Benutzerverwaltung wie etwa bei Unix auch für SMOs verwenden – alle dazu nötigen Mechanismen werden von SMOs bereitgestellt. Eventuell gibt es jedoch eine bessere Möglichkeit für eine Benutzerverwaltung, die optimal auf die speziellen Fähigkeiten von SMOs abgestimmt ist.
- Die Namensauflösung wurde nur ansatzweise betrachtet. Es müßte genauer überlegt werden, welche Lokalität Namen besitzen, d.h. ob ein Programm normalerweise einen einzigen globalen Namensraum besitzt und ob Namen auch in größeren Kontexten lokal sein können (z.B. rechnerlokale Namen).
- Objektmigration und Aufruf mit Übergabe als Wert wurden nicht betrachtet. Ein Objekt, das auf einen anderen Rechner migriert, muß dem Zielsystem (Interpreter und Rechner) vertrauen. Sonst kann es nicht sicherstellen, daß es nach der Migration seine Tätigkeit weiter korrekt verrichten kann. Dazu benötigt man zusätzliche Mechanismen, die Objektmigration und Objektkopie erlauben bzw. verbieten können. Die Implementation gestaltet sich ebenfalls schwieriger: Ein System, das Objektmigration unterstützt, muß auch Objektsuche implementieren.
- Es wurde zwar ein Prototyp implementiert, dieser wurde jedoch im wesentlichen auf den Aspekt der Realisierbarkeit des Modells hin implementiert. Der Effizienzaspekt stand im Hintergrund. Für die Verwendung des Modells ist die Effizienz jedoch – zumindest für ei-

nige Anwendungen – ein wichtiger Faktor. Es stellt sich daher die Frage, wie effizient Sicherheitsmetaobjekte – eventuell durch Verwendung eines optimierenden Compilers – realisiert werden können.

- Interessant wäre eine Verwendung von Sicherheitsmetaobjekten in realen Anwendungen zur Evaluierung des Implementationsaufwandes. Obwohl einige Szenarien untersucht wurden, wurden keine kompletten, bereits existierenden Anwendungen mit Sicherheitsmetaobjekten versehen.

Bei einer Evaluierung einer realen Applikation könnten sich weitere Szenarien für Sicherheitsstrategien herauskristallisieren. Dank der hohen Konfigurierbarkeit werden sich nahezu alle Szenarien mit Sicherheitsmetaobjekten realisieren lassen, es stellt sich nur die Frage, wieviel Aufwand dies macht und ob man die Sicherheitsmetaobjekte generisch implementieren kann, so daß sie in ähnlichen Szenarien wiederverwendbar sind.

8 Literatur

BBN96 Benantar, M.; Blakley, B.; Nadalin, A.: Approach to Object Security in Distributed SOM, *IBM Systems Journal*, Vol. 35 No. 2, 1996, New York.

BlK97 Blakeley, B.; Kienzle, M. D.: Some Weeknesses of the TCB Model, Discussion paper in: *1997 IEEE Symposium on Security and Privacy*, Oakland, California, 1997.

Boo94 Booch, Grady: *Object Oriented Analysis and Design*. Redwood City, Kalifornien, USA, 2. Auflage, 1994.

Bro89 Brockhaus AG (Bearb.: Claus, Volker; Schwill, Andreas): *Duden der Informatik*, Dudenverlag, Wien, 1988.

Bro98 Brose, G.: Reflection in Java, CORBA und JacORB, *JIT'98 - Java Informations-Tage 1998*, Springer, 1998.

CQL+96 Campell, R.; Qian, T.; Liao, W.; Liu, Z.: Active Capability: *A unified Security Model for Supporting Mobile, Dynamic and Application Specific Delegation*. White Paper. University of Illinois, 1996.

Cus93 Custer, Helen: *Inside Windows NT.* Washington, Microsoft Press, 1993.

DeH66 Dennis, Jack B., Van Horn, Earl C.: Programming Semantics for Multiprogrammed Computations, In: *Communications of the ACM*, März 1966.

Dod83 Department of defense: *Trusted Computer System Evaluation Criteria* (Orange Book), Department of defense Computer Security Center, 1983.

FCK95 Ferraiolo, D.; Cugini, J.; Kuhn, R.: Role Based Access Control: Features and Motivations, *Proceedings of Annual Computer Security Applications Conference*, IEEE Computer Society Press, Baltimore, 1995.

FeK92 Ferraiolo, D.; Kuhn, R.: Role-based Access Control, In: *15th NIST-NCSC National Computer Security Conference*, p. 554-563, IEEE Computer Society Press, Baltimore, Oktober 1992.

Fla96 Flanagan, David: *Java in a Nutshell*, O'Reilly & Associates, 1st edition, Februar 1996.

Gar95 Garfinkel, Simson: *PGP: Pretty Good Privacy.* Sebastopol, Kalifornien, O'Reilly, Januar 1995.

Ger98 Gerdom, Michael: *Erweiterung eines Prototypen für ein metaobjektbasiertes Sicherheitskonzept in Java*. Studienarbeit am Institut für Mathematische Maschinen und Datenverarbeitung, Lehrstuhl für Betriebssysteme. SA-I4-98-09, 1998, Betreuer Prof. F. Hofmann, T. Riechmann.

Gon98 Gong, L.: *Java Security Architecture (JDK1.2)*, White Paper, Sun Microsystems, Palo Alto, 1998.

GoR89 Goldberg, A.; Robson, D.: *Smalltalk-80 The Language*. Addison Wesley, Reading, Massachusetts, 1989.

Hie93 Hieronymus, A.: *Unix*. Braunschweig, Vieweg, 1993.

Hof84 Hofmann, F.: *Betriebssysteme: Grundkonzepte und Modellvorstellungen*. Teubner, Stuttgart, 1984.

HRU76 Harrison, M.; Ruzzo, W.; Ullman, J.: Protection in Operating Systems. In: *Communications of the ACM* (CACM) 19 Nr. 8, 1976.

Ive96 Ivens, Kathy: *Windows NT Workstation - Professional Reference*. New Riders Publishing, Indianapolis, USA, 1996.

KaG98 Kassab, L., Greenwald, S.: Towards Formalizing the Java Security Architecture of JDK 1.2, *Proceedings of the European Symposium on Research In Computer Security*, Louvain-la-Neuve, Belgien, September 1998.

Kle92 Kleinöder, Jürgen: *Objekt- und Speicherverwaltung in einer offenen, objektorientierten Betriebssystemarchitektur.* Dissertation, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), Erlangen, Juli 1992.

KlG96 Kleinöder, Jürgen.; Golm, Michael: "MetaJava: An Efficient Run-Time Meta Architecture for Java", *IWOOOS '96 workshop*, Seattle, 1996.

KoN93 Kohl, J., Neuman, C.: *The Kerberos Network Authentication Service (V5)*, IETF Network Working Group Request for Comments 1510, September 1993.

KPS95 Kaufman, C.; Perlman, R.; Speciner, M.: *Network Security - Private Communication in a Public World*, Prentice Hall Series in Computer Networking and Distributed Systems, Englewood Cliffs, New Jersey, 1995.

Lam73 Lampson, B.: A Note on the Confinement Problem, In: *Communications of the ACM 1973*, pp. 613-615, Oktober, 1973.

Lev84 Levy, H.: *Capability-Based Computer Systems*. Bedford, Mass., Digital Press, 1984.

Luc98 Luckhardt, Norbert: Rückwärtiges Portal für Windows. In: *c't Magazin für Computer Technik 17/1998*, Verlag Heinz Heise GmbH, Hannover, 1998.

Mad96 Madany, Peter W.: *JavaOS: A Standalone Java Environment*, Sun Microsystems Inc. / JavaSoft White Paper, Mountain View, Kalifornien, USA. Mai 1996.

Mae87 Maes, P.: *Computational Reflection*, Ph.D. Thesis, Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

Mcl85 McLean, John : A Comment on the "Basic Security Theorem" of Bell and La Padula. *Information Processing Letters*, Vol. 20, Nr. 2, Februar 1985.

Mcl94 McLean, John: Security Models. In: *Encyclopedia of Software Engineering*, Wiley Press, Februar 1994.

Mcl97 McLean, John: Is the Trusted Computing Base Concept Fundamentally Flawed?, Discussion paper in: *1997 IEEE Symposium on Security and Privacy*, Oakland, California, 1997.

Mey90 Meyer, Bertrand: *Objektorientierte Softwareentwicklung*. Prentice-Hall International Inc., München, 1990.

MGH+94 Mitchell, J.; Gibbons, J.; Hamilton, G. et.al.: An Overview of the Spring System. *Proceedings of the Compcon Spring 1994 (San Francisco)*. Los Alamitos, IEEE, 1994.

Mic96 Microsoft: *Microsoft Windows NT Server - DCOM Technical Overview*. White Paper, Microsoft Corporation, Redmond, USA, 1998.

Mic98 Microsoft: *DCOM Architecture*. White Paper, Microsoft Corporation, Redmond, USA, 1998.

Mon98 Monadjemi, Peter: *Windows 95 - Kompendium*. Neuauflage mit Internet Explorer 4. Markt & Technik, Haar bei Muenchen, 1998.

Neu93 Neumann, C. B.: Proxy-Based Authorization and Accounting for Distributed Systems, In: *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Mai 1993.

OHE96 Orfali, Robert; Harkey, Dan; Edwards, Jeri: *The Essential Distributed Objects Survival Guide*. New York, 1996.

Omg97 Object Management Group: *CORBAservices: Common Object Services Specification*. November 1997.

Osf92 Open Software Foundation: *Security in a Distributed Computing Environment*, White Paper, Open Software Foundation, Cambridge, USA, 1992.

Red96 Redlich, J.: *Corba 2.0 - Praktische Einführung für C++ und Java*, Addison Wesley, Bonn, 1996.

RHK98 Riechmann, Thomas; Hauck, Franz J.; Kleinöder, Jürgen: Transitiver Schutz in Java durch Sicherheitsmetaobjekte, In: *JIT'98 - Java Informations-Tage 1998*, Springer, 1998.

RiH97 Riechmann, Thomas; Hauck, Franz J.: "Meta Objects for Access Control: Extending Capability-Based Security", In: *Proc. of the ACM New Security Paradigms Workshop 1997*, Great Langdale, England, Association for Computing Machinery (ACM), September 1997.

RiH98 Riechmann, Thomas; Hauck, Franz J.: "Meta Objects for Access Control: A Formal Model for Role-Based Principals". In: *ACM New Security Paradigms Workshop 1998 Proceedings* (NSPW '98), Charlottesville, USA, Association for Computing Machinery (ACM), 1998.

RiK98 Riechmann, Thomas; Kleinöder, Jürgen: Meta Objects for Access Control: Role-Based Principals. In: *Proceedings of Third Australasian Conference on Information Security and Privacy* (ACISP '98), Brisbane, Australien, Springer, 1998.

RiR99 Riechmann, Gerald; Riechmann, Thomas: Selbstabreißer – Motivkalender für den Active Desktop. In: *c't Magazin für Computer Technik 2/1999*, Verlag Heinz Heise GmbH, Hannover, 1999.

SaS75 Saltzer, Jerome H.; Schroeder, Michael D.: The Protection of Information in Computer Systems. In: *Proceedings of the IEEE*, volume 63, number 9, September 1975.

Sin97 Sinha, Pradeep K.: *Distributed Operating Systems - Concepts and Design*. IEEE Computer Society Press, New York, 1997.

SiS94 Mukesh Singhal, Niranjan G. Shivaratri: *Advanced Concepts in Operating Systems*, McGraw Hill, New York, 1994.

Ste96 Steffen, Jorge: *Referenz Handbuch - HTML 3.2*. Sybex-Verlag, Düsseldorf, 1. Auflage, 1996.

Str92 Stroustrup, Bjarne: *Die C++ Programmiersprache*, Addison Wesley Longman, Bonn 1992.

Sum97 Summers, Rita C.: *Secure Computing - Threats and Safeguards*. Computing McGraw-Hill, New York, 1997.

Sun95 Sun Microsystems Comp. Corp.: *HotJava: The Security Story*, White Paper, 1995.

Sun95a Sun Microsystems Comp. Corp.: S*pring Research Distribution 1.1 Source*, Source CD, 1995.

Sun96 Sun Microsystems Comp. Corp.: *Java Remote Method Invocation Specification*, Revision 12, JDK 1.1 Beta Draft, 1996.

TMR86 Tanenbaum, Andrew S.; Mullender, Sape J.; van Renesse, Robbert: Using sparse capabilities in a distributed operating system. In: *Proceedings of the 6th International Conference on Distributed Computing Systems*, pp. 558-563, IEEE, Cambridge, Mass., Mai 1986.

Tim97 Timmermann, B.: A Security Model for Dynamic Adaptable Traffic Masking, In: *Proceedings of ACM New Security Paradigms Workshop 1997*, Great Langdale, England, Association for Computing Machinery (ACM), September 1997.

WBD+97 Wallach, Dan S.; Balfanz, Dirk; Dean, Drew; Felten, Edward W.: Extensible Security Architecture for Java. *Symposium on Operating Systems Principles 1997*, pp. 116-128, Saint-Malo, Frankreich, Oktober 1997.

WCC+74 Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; Pollack, F.: HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM 17(6)*, pp. 337-344, Juni 1974.

WWK96 Wang, C.; Wulf, W.; Kienzle, D.: A New Model of Security for Distributed Systems, In: *Proceedings of the 1996 ACM New Security Paradigms Workshop*, Lake Arrowhead, Kalifornien, USA, Association for Computing Machinery (ACM), September 1996.