Formale Verifikation der Korrektheit sicherheitskritischer Java Anwendungen

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg
im Jahr 2008 von

Holger Grandy

Universität Augsburg

Amtierender Dekan:	Prof. Dr. Wolfgang Reif
Gutachter:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
Tag der Prüfung:	2. Juni 2008
Prüfer	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
	Prof. Dr. Alexander Knapp

Für Vincent und Elias

Kurzzusammenfassung

Sei es im Internet beim Homebanking, bei Chipkarten wie der Geldkarte oder bei der kommenden Gesundheitskarte - Sicherheit ist einer der entscheidenden Faktoren für die Akzeptanz und den Erfolg kommunizierender Anwendungen. Hohe Summen werden investiert, um Sicherheitsziele wie Vertraulichkeit oder Authentizität von Daten bei Entwurf und Implementierung solcher Anwendungen zu garantieren. Dennoch finden sich fast täglich Meldungen über Sicherheitslücken von Computersystemen in den Medien. Der Einsatz formaler Methoden bietet die derzeit maximal möglichen Garantien für die Verlässlichkeit von Computersystemen. Bisherige Ansätze zur formalen Behandlung von kommunizierenden Anwendungen erlauben allerdings keine verlässliche Aussage über die Sicherheit einer tatsächlichen Implementierung, sondern beschränken sich meist auf die Verifikation von Modellen. Diese Arbeit führt - basierend auf Theorien zur formalen Verfeinerung von Systemen - bisherige Verifikationsansätze bis zur Codeebene fort. Hauptergebnis der Arbeit ist eine Spezifikations- und Verifikationsmethodik, die es erlaubt, formale Beweise für die Sicherheit von Anwendungen auf einem Modell zu führen und diese dann formal korrekt auf eine tatsächliche Implementierung des Systems zu übertragen. Die Methodik wird an zwei Anwendungen für Mobiltelefone und Chipkarten illustriert.

Danksagung

Ein Fachmann ist ein Mann, der einige der gröbsten Fehler kennt, die man in dem betreffenden Fach machen kann und der sie deshalb zu vermeiden versteht.
Werner Heisenberg

Die vorliegende Arbeit wäre nicht möglich gewesen, wenn nicht viele Fachmänner (und natürlich auch und insbesondere Fachfrauen) intensiv dazu beigetragen hätten. Mein Dank gilt im Besonderen:

meinem Doktorvater **Prof. Dr. Wolfgang Reif** für sein Vertrauen in mich, für die Anleitung zu dieser Dissertation, seine Motivation, für seine fachlichen Anmerkungen und seine Kritik

Dr. Kurt Stenzel für seine Diskussionsbereitschaft, seine endlose Geduld und seine Hilfestellungen, die mich oft wieder in die richtige Richtung geleitet haben, sowie nicht zuletzt auch für die hervorragenden Ergebnisse seiner Dissertation zur Java Verifikation, auf denen ich mit dieser Arbeit aufbauen durfte

Markus Bischof und **Robert Bertossi** für ihr unglaublich hohes Engagement und die daraus resultierenden hervorragenden Ergebnisse bei der Verifikation der Fallstudien

Dr. Gerhard Schellhorn für die Klärung vieler Feinheiten der formalen Verfeinerungstheorie und der Abstract State Machines sowie seine hilfreichen Hinweise zur Beweismethodik bei Verfeinerungen

Nina Moebius für die vielen Diskussionen über Datentypen, ihre Anmerkungen zu dieser Arbeit und nicht zuletzt für die schöne Zeit in einem Büro

Dr. Dominik Haneberg für die Ausarbeitung seiner Spezifikations- und Beweismethodik für abstrakte Kommunikationsprotokolle, die ebenfalls Grundlage dieser Arbeit ist

Florian Nafz für das intensive Korrekturlesen dieser vielen Seiten

meinen weiteren Kollegen **Simon Bäumler, Andriy Dunets, Matthias Güdemann, Alwin Hoffmann, Dr. Frank Ortmeier, Jonathan Schmitt** und **Hella Seebach** am Lehrstuhl für Softwaretechnik für das gute Arbeitsklima und den Spaß, den ich mit ihnen haben durfte

meiner Frau Diana für ihre Liebe, ihre Unterstützung und ihr Verständnis in den vielen schwierigen und auch den vielen schönen Momenten in den letzten vier Jahren

Holger Grandy

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Aut non tentaris, aut perfice.
Ovid

1.1 Motivation

Elektronische Zahlungssysteme stellen hohe Anforderungen an die Sicherheit der verwendeten Technologien. Ein Beispiel für diese Systeme ist die auf Chipkartentechnologie basierende Mondex Applikation [89] von Mastercard, die eine elektronische Geldbörse realisiert. Sicherheitslücken können die Akzeptanz solcher Systeme sehr negativ beeinflussen, sie sichern teils große monetäre Werte.

Die Grundidee von Mondex ist der Ersatz von Bargeld. Hauptfunktionalität der Karten ist das Übertragen von Geld von einer Karte auf eine andere. Es ergeben sich sofort wesentliche Fragen hinsichtlich der Sicherheit der Applikation. Ein Angreifer darf nicht in der Lage sein, das System so auszunutzen, dass Kunden oder Banken dadurch einen Nachteil und/oder der Angreifer einen Vorteil hat. Ein Beispiel für ein solches Angriffsszenario wäre z.B. der Versuch, mit gefälschten Nachrichten auf einer echten Karte Geld zu erzeugen.

Es stellt sich somit die Frage, wie die Sicherheit solcher Anwendungen sichergestellt werden kann. Abstrakt betrachtet handelt es sich z.B. bei Geldkarten wie Mondex um Systeme, die bei Übertragung von Geld auf einer Karte Geld abziehen und gleichzeitig auf einer anderen Karte Geld gutschreiben. Real ist dies natürlich nicht so einfach umzusetzen, hier muss eine Kommunikation zwischen den Karten stattfinden, die Abbuchung und Gutschrift von Geld auslöst. Hier kann nun allerdings ein Angreifer intervenieren, was den Einsatz eines passenden Kommunikationsprotokolls unter Benutzung von kryptographischen Primitiven wie digitalen Signaturen, Verschlüsselung und Hashing erfordert. Der Entwurf solcher Protokolle ist dabei inhärent schwierig, da die Möglichkeiten eines Angreifers zum Entwurfszeitpunkt nur sehr schwer von Menschen zu erfassen sind und die Protokolle üblicherweise schnell komplex werden. Der bekannteste Beleg für diesen Sachverhalt ist der über 10 Jahre verborgen gebliebene Fehler im Needham-Schroeder Authentifizierungsprotokoll [118].

Neben der Korrektheit und Sicherheit des Kommunikationsprotokolls ist eine wesentliche wei-

tere Frage, ob eine tatsächliche Implementierung in einer Programmiersprache die Vorgaben des Protokolls korrekt umsetzt. Ein System, das durch Implementierungslücken seine (wenn auch vielleicht sichere) Spezifikation nicht korrekt umsetzt, kann dennoch unsicher sein.

Darüber hinaus gibt es noch weitere Quellen für Unsicherheiten in den Anwendungen. Diese können durch die verwendete Hardware der Komponenten entstehen wie z.B. durch Side-Channel-Angriffe auf Chipkarten mittels Differential-Power-Analysis. Daneben können die verwendeten kryptographischen Primitive unsicher sein (oder mit gewachsener Rechnergeschwindigkeit über die Zeit unsicher geworden sein). Beispiele sind etwa die bekannten Unsicherheiten in der Wireless LAN Verschlüsselung WEP oder die Möglichkeit im Hashing-Algorithmus MD5 gezielt Kollisionen zu erzeugen. All diesen Angriffen ist allerdings gemein, dass sie isoliert gelöst werden können: es kann ein besserer Kryptoalgorithmus verwendet werden oder die Hardware entsprechend besser gegen physikalische Angriffe abgesichert werden. Die Protokolle und ihre korrekte Implementierung sind allerdings applikationsabhängig und die Sicherheit ist nicht durch einen generischen Ansatz für alle weiteren Anwendungen erzwingbar. Diese Arbeit erläutert eine Methodik, wie solche kommunizierenden Computersysteme entworfen werden können, wie ihre Sicherheit auf der Ebene des Protokolls sichergestellt werden kann und wie insbesondere die Implementierung als korrekt gegenüber ihrem Protokoll (und damit ebenso als sicher) nachgewiesen werden kann.

Die Informatik hat dabei unzählige Methoden und Tools hervorgebracht, um Computersysteme vor und nach der eigentlichen Umsetzung auf Korrektheit überprüfen zu können. Die formale Verifikation von Computersystemen soll die Wahrscheinlichkeit für Fehlverhalten der Systeme minimieren. Mittels streng mathematischer Formalismen, den *formalen Methoden*, soll ermöglicht werden, Verhalten von Systemen am Modell zu verifizieren und somit Fehler im realen System auszuschließen.

Vielen dieser Ansätze ist gemein, dass die verifizierten Eigenschaften lediglich auf starken Abstraktionen der realen Systeme beruhen (und auch nur beruhen können, da die Formalismen selbst bereits lediglich Abstraktionen als Modelle zulassen). Damit existiert eine semantische Lücke zur tatsächlichen Implementierung. Dies trifft in dieser Allgemeinheit auf fast alle Spezifikationssprachen wie z.B. algebraische Spezifikationen [186] [116], Z [171], B [2], ASM [27], CSP [80] oder Temporale Logik [119] zu. Aussagen, die in solchen Formalismen über Systeme gezeigt werden, können also letztendlich nur Hinweise auf das Verhalten (oder das Fehlverhalten) der Implementierung sein, aber keine Garantien abgeben. Auf der anderen Seite ist die in den Formalismen erreichte Abstraktion auch eine große Hilfe bei der Verifikation, weil sie es ermöglicht, den Kern der Systeme zu betrachten, statt sich mit technischen (und teils unwichtigen) Details zu beschäftigen.

Auch speziell für den Bereich kryptographischer Protokolle hat die aktuelle Forschung viele Herangehensweisen entwickelt, um Sicherheitseigenschaften der Protokolle bereits am Modell überprüfen zu können [139] [118] [41] [34] [101]. Insbesondere hier stellt sich auch die Frage, ob eine lauffähige Implementierung eines Protokolls tatsächlich korrekt bezüglich seiner Spezifikation ist und damit auch die gewünschten Sicherheitseigenschaften in der Realität erfüllt.

Viele aktuelle Forschungsvorhaben haben - um die genannte semantische Lücke zu verkleinern - die Verifikation tatsächlich eingesetzter Programmiersprachen zum Ziel. Beispiele für Programmiersprachenverifikation sind etwa die diversen Kalküle und Tools für die Verifikation von Java [61] in [19] [55] [40] [30] [183] [125] [120] [185] [31] [172] oder C# [15] [126] die in den letzten Jahren in den Fokus der Forschung gerückt sind. Hier wird tatsächlich der Code verifi-

ziert, der im späteren System real eingesetzt wird. Zwar besitzen auch diese Ansätze noch eine gewisse Lücke zur Realität - es ist keine aktuelle und realistische Laufzeitumgebung für Code (Compiler, eine eventuelle virtuelle Maschine sowie Betriebssystem und Hardware) vollständig und durchgängig verifiziert worden. Dennoch verkleinert sich die semantische Lücke zum realen System bei der Verifikation von konkreten (lauffähigen) Implementierungen sehr. Diese Nähe zur Realität ist jedoch gleichzeitig auch das Problem dieser Ansätze bei der tatsächlichen Verifikation: Eine Abstraktion bzw. Modellbildung zur Erleichterung der ohnehin meist schwierigen Verifikation ist hier oft praktisch nicht möglich, da die Beweissysteme und Logiken keine Abstraktionsmöglichkeiten vorsehen. Die Konkretisierung der Verifikationsmethoden ist somit ein zweischneidiges Schwert.

In dieser Arbeit wird eine Methodik entwickelt, die die Frage der Korrektheit einer Implementierung gegenüber einer abstrakten Spezifikation beantworten kann. Insbesondere der Erhalt von Eigenschaften der Spezifikation in der Implementierung ist dabei sichergestellt. Die Verifikation funktionaler Eigenschaften sowie von Sicherheitseigenschaften eines Protokolls erfolgt auf einer abstrakteren Spezifikationsebene, die dem PROSECCO Spezifikations- und Verifikationsansatz aus der Dissertation von Haneberg [71] folgt. Ein weiterer Verifikationsschritt überträgt die gezeigten Eigenschaften uniform auf eine Java Implementierung. Die Methodik basiert auf formaler Verfeinerung.

1.2 Überblick über die Arbeit

Abb. 1.1 gibt einen Überblick über die Zusammenhänge zwischen den Kapiteln dieser Arbeit. Es sind auch die jeweils wichtigsten Inhalte der Kapitel genannt. Kapitel für Fallstudien oder mit starker Praxisorientierung sind gestrichelt dargestellt.

Die Arbeit unterteilt sich demnach grob in drei verschiedene Bereiche: die Beispiele, den Kern der Spezifikationsmethodik sowie einige Erweiterungen.

Zu Beginn stellt Kapitel 2 zunächst die beiden in dieser Arbeit verwendeten praktischen Beispiele vor - die Handy Anwendung Cindy und die Chipkarten Anwendung Mondex.

Diese Arbeit beruht insbesondere auf den Ergebnissen der drei Dissertationen zu ASM Verfeinerung von Schellhorn [155], zu Java Verifikation von Stenzel [173] und der Protokollspezifikationsmethodik PROSECCO von Haneberg [71]. Zusätzlich wird das KIV System [149] [148] [150] als Spezifikations- und Beweissystem verwendet. Die theoretischen Grundlagen und die Ergebnisse der genannten Dissertationen, die für diese Arbeit wichtig sind, werden in Kapitel 3 zusammengefasst.

Kapitel 4 vergleicht den Ansatz mit verwandten Arbeiten. Hier werden grundlegende Methodiken anderer Arbeitsgruppen zur Verifikation von objektorientiertem Code und von Kommunikationsprotokollen gezeigt. Ein technischerer Detailvergleich, insbesondere am Beispiel der Mondex Applikation, findet sich später in der Arbeit in Kap. 12.8.

Der Kern der Spezifikationsmethodik für die Protokollverfeinerung zu Java Code findet sich in Kapitel 5 und 6. Während Kapitel 5 die Verfeinerungstheorie dieser Arbeit und grundlegende Beweisverpflichtungen einführt, erklärt Kapitel 6 die Datentypen, auf denen die abstrakte Spezifikation und die Implementierung beruhen. Ein Hauptproblem bei der Verfeinerung zu Code ist das Mapping dieser Datentypen.

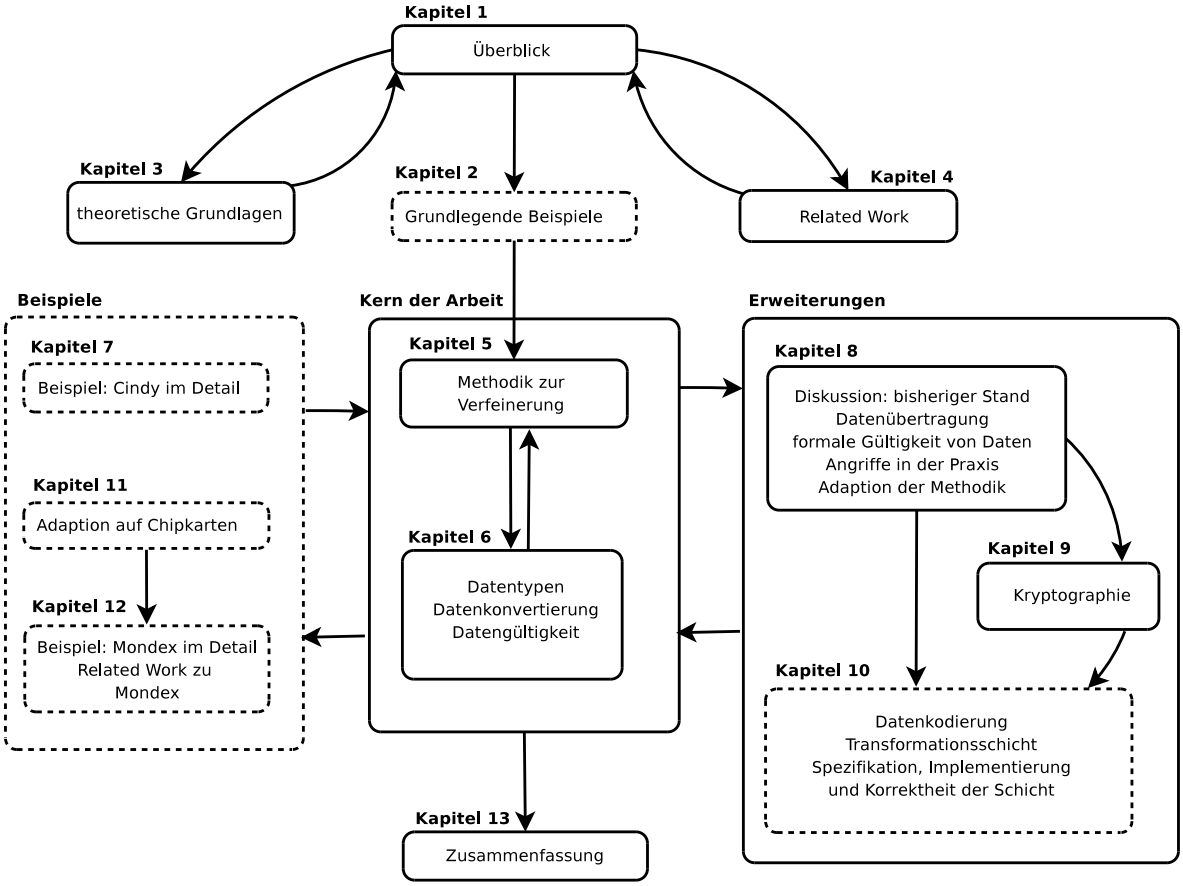


Abbildung 1.1: Übergänge zwischen den Kapiteln dieser Arbeit

Zu Beginn von Kapitel 8 wird der bisherige Stand der Arbeit diskutiert. Dabei wird aufgezeigt, dass die Betrachtungen der vorangehenden Kapitel erweitert werden müssen, um verlässliche Korrektheitsaussagen durch Beweise der Verfeinerungskorrektheit für die Implementierung machen zu können. Bedingt durch die Tatsache, dass reale Datentypen (wie z.B. Pointerstrukturen) wesentlich mehr Fehlerquellen beinhalten können als ihre abstrakten Entsprechungen, muss das Verfeinerungskonzept erweitert werden und auch diese möglichen neuen Fehlerquellen betrachtet werden. Dazu muss die Verfeinerungsmethodik entsprechend angepasst werden. Kapitel 9 erweitert zusätzlich den bisherigen Ansatz um die Betrachtung kryptographischer Operationen.

Schließlich wird in einer zweiten Erweiterung in Kapitel 10 eine generische Datentransformationsschicht eingeführt, die in der Lage ist, die in Kapitel 8 eingeführten neuen Fehlerquellen generisch zu beseitigen. Am Ende des Kapitels wird gezeigt, dass die Datentransformationsschicht tatsächlich genau die Eigenschaften erfüllt, die bereits in Kapitel 6 und 8 für die Verfeinerungsbeweise als Annahmen verwendet wurden.

Dabei wird die Arbeit durch die Betrachtung der Details der beiden Beispielanwendungen komplettiert. Kapitel 7 zeigt die Spezifikation, Implementierung und Verifikation der Cindy

Anwendung, die noch keine kryptographischen Operationen verwendet und relativ einfache Protokolle besitzt. Nach Vorstellung der Erweiterungen des Ansatzes wird in Kapitel 11 eine Erweiterung der bisherigen Methodik für Chipkartenanwendungen vorgestellt. Darauf aufbauend wird in Kapitel 12 die Verifikation der Mondex Anwendung erläutert.

Kapitel 13 fasst die Ergebnisse schließlich kurz zusammen und gibt einen Erfahrungsbericht sowie einen Ausblick auf weitere Schritte.

Hinweis: Die vorliegende Arbeit beschreibt und/oder referenziert oftmals formale Beweise, die mit dem KIV System geführt wurden. Diese Beweise können nicht ausführlich in der schriftlichen Arbeit erläutert werden, da sie wesentlich zu umfangreich sind. Statt dessen wurden alle Beweise aus KIV exportiert und sind im Web unter [49] aufrufbar.

1.3 Erreichte Ergebnisse

Die folgenden Punkte fassen in aller Kürze die Kernergebnisse dieser Dissertation zusammen:

- **Generische Definition eines Refinement Frameworks für die Verifikation von Kommunikationsprotokollen**
  Es wird ein allgemeines Spezifikations- und Beweisframework für den Nachweis der Korrektheit einer Verfeinerung von Kommunikationsprotokollen zu Code eingeführt. Das Framework basiert auf Abstract State Machines auf der abstrakten Ebene und Java Implementierungen auf der konkreten Ebene. Der vorgestellte Ansatz ist generisch für Kommunikationsprotokolle und lässt sich auch einfach auf die Verifikation in anderen Anwendungsgebieten übertragen. Eine Anwendung einer formalen Verfeinerungstheorie bei der Verifikation von tatsächlich lauffähigem Quellcode ist nach bestem Wissen des Autors bisher noch nie durchgeführt worden. Eine formale Verfeinerungstheorie zum Beweis von Sicherheitseigenschaften auf Quellcodeebene ist ebenso bisher noch nicht angewendet worden.

- **Definition einer generischen Datentypabbildung von abstrakter Protokollebene hin zu Java Klassen**
  Es wird eine Abbildung der in Prosecco Spezifikationen verwendeten abstrakten Protokolldatentypen in Java Klassen definiert. Die Abbildung ist generisch für alle Prosecco Spezifikationen und erleichtert damit die Implementierung und auch die Verifikation der Protokolle durch eine standardisierte Datentypkonversion. Die Datentypabbildung ist in allen Prosecco Fallstudien wiederverwendbar.

- **Definition zweier Abstraktionsebenen für die Strukturierung des Refinement-Beweises**
  In der Arbeit wird der Beweis der Verfeinerungskorrektheit einer Protokollimplementierung in zwei Abstraktionsebenen zerlegt: die Ebene der konkreten Nachrichtenübertragung und die Ebene der Protokollimplementierung. Auf der Ebene der Nachrichtenübertragung wird ein Übertragungsformat definiert, das für die reale Kommunikation, z.B. in einem Netzwerk, verwendet werden kann. Das Übertragungsformat stellt eine Kodierung der Datentypen der Protokollimplementierung dar. Die Implementierung der Kodierung kann für verschiedene Protokollimplementierungen wiederverwendet werden und sorgt ihrerseits bereits für den Ausschluss bestimmter Angriffe auf die Protokolle.

- **Korrektheitsnachweis der Implementierung für die Nachrichtenübertragungsebene**
  Die Implementierung der Nachrichtenübertragungsebene wurde als korrekt bezüglich einer Spezifikation nachgewiesen. Damit wird die Wiederverwendung der Eigenschaften der Übertragungsebene in Form von Lemmata in Korrektheitsbeweisen für Protokollimplementierungen möglich.

- **Nachweis der Anwendbarkeit des Ansatzes für Fallstudien realistischer Größe**
  Die Verfeinerungsmethodik wurde auf eine Implementierung der Mondex Geldkarte [89] sowie auf eine Implementierung einer Ticketing-Applikation für Mobiltelefone angewendet. Insbesondere die Mondex Verifikation ist wohl die größte bisher mit dem KIV System durchgeführte Fallstudie.

- **Erweiterung und Verbesserung des Beweissupports für Java im KIV Beweissystem**
  Bei der Anwendung auf Fallstudien ergaben sich viele offene Fragen bezüglich des Beweissupports für Java im KIV Beweissystem. Hier wurden, insbesondere im Bereich der Verifikation von Systemen, die mit komplexen Pointerstrukturen arbeiten, etliche Verbesserungen sowohl in die Basisspezifikationen für den Java Kalkül als auch im Bereich der Axiomengenerierung vorgenommen. Schließlich wurde als Ausblick eine erfolgversprechende neue Axiomatisierung des Java-Speichermodells für Kalkül und Semantik implementiert.

Kapitel 2

Zwei praktische Beispiele

...the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of seemingly great generality is in essence the same as a small and concrete special case.
Paul Halmos, I want to be a Mathematician

2.1 Cindy

Cindy[1] ist eine J2ME Anwendung für Java-fähige Mobiltelefone, die zum Kauf von Kinokarten mit dem Handy verwendet werden kann. Abb. 2.1 zeigt den Ablauf bei Benutzung der Applikation.

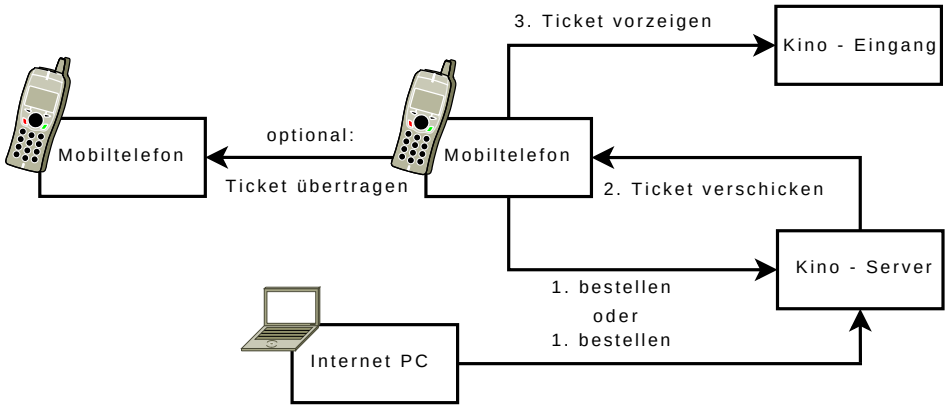


Abbildung 2.1: Die Cindy Applikation

Der Anwender benutzt das Telefon oder einen beliebigen Web Browser an einem Internet PC, um beim Kino ein Ticket zu bestellen. Daraufhin schickt der Kino-Server eine Multimedia Messaging System (MMS) Nachricht an die dem Anwender zugeordnete Mobiltelefonnummer. Die

[1] Cinema Handy

Abrechnung erfolgt entweder über die Telefonrechnung als Premium MMS oder beim Internet-Kauf über eine ebenso dem User zugeordnete Kreditkartennummer. In der versendeten MMS ist als Binärinhalt ein 2D Datamatrix Barcode enthalten, der einen eindeutigen, nicht erratbaren Identifier für das Kinoticket enthält. Zusätzlich sind noch die Ticketinformationen wie z.B. die Sitzplatznummer im Ticket enthalten. Dieser Datamatrix Code dient nun als digitale Eintrittskarte für das Kino. Der Benutzer muss lediglich mit dem Datamatrix Code auf dem Display des Mobiltelefons an einen Barcode Scanner im Kino kommen. Daraufhin kann der Code direkt vom Display eingescannt werden und ein Drehkreuz für den Einlass geöffnet werden. Der selbe Code wird dabei nur einmal am Eingang zum Kino akzeptiert. Zusätzlich zum Kauf ist auch die Übertragung von Tickets von einem Handy zum anderen vorgesehen.

Ein langwieriges Warten an der Kinokasse ist mit Cindy nicht mehr notwendig. Die Anwendung wurde vollständig in Java und J2ME implementiert und ist als Prototyp auf echten Java-fähigen Telefonen und mit einem Datamatrix Code Scanner lauffähig. Die selbe Art Applikation wurde kurz nach der hier vorgestellten Untersuchung der Fallstudie auch in den Niederlanden von der Firma beep [179] und nach und nach auch bei anderen Kinos und anderen Unternehmen wie der Deutschen Bahn [48] oder - entwickelt von der Firma MindMatics [128] - auch für T-Mobile oder S.Oliver implementiert. Ein auf ähnlicher Technik basierendes System zum Fahrkartenkauf wurde kürzlich von der Deutschen Bahn eingeführt [48].

Die Verhinderung von Betrug ist im Rahmen der Cindy Anwendung natürlich die zentrale Fragestellung. So sollen z.B. nur genau diejenigen Tickets, die ordnungsgemäß bezahlt wurden, am Kinoeingang auch akzeptiert werden. Des Weiteren soll, falls ein Ticket bestellt (und damit bezahlt) wurde, dieses auch tatsächlich an den Benutzer ausgeliefert werden. Diese Ziele können mit dem Prosecco Ansatz (Kap. 3.5) für eine abstrakte Spezifikation von Cindy gezeigt werden [63]. Diese Arbeit beschäftigt sich nun mit der Fragestellung, wie nachgewiesen werden kann, dass solche Eigenschaften auch für die tatsächlich lauffähige Implementierung gelten.

Das Kommunikationsprotokoll von Cindy ist sehr einfach (s. Kap. 5.1) und die Fähigkeiten des Angreifers sind recht beschränkt (so kann er z.B. keine MMS Nachrichten abhören oder ändern). Cindy wird in dieser Arbeit daher als einfaches Beispiel für die Vorstellung des Verfeinerungsansatzes in Kap. 5.3 und im Detail in Kap. 7 verwendet.

2.2 Mondex

Die Mondex Fallstudie dieser Arbeit ist eine Implementierung der realen Mondex Chipkartenanwendung von MasterCard [89]. Die Funktionalität der Mondex Chipkarten ist die Übertragung von elektronischem Geld von einer Karte auf eine andere. Dabei sind die Karten als Ersatz für Bargeld anzusehen. Zwei Karten (auch *Purses* genannt) kommunizieren mittels eines Chipkartenlesegeräts mit zwei Kartenslots (auch Terminal oder *Wallet* genannt). Sie führen dabei ein mehrschrittiges Protokoll aus, das insbesondere gegen vorzeitige Unterbrechung der Kommunikation oder Fälschen von Nachrichten sicher sein soll. So soll es z.B. nicht möglich sein, Geld aus dem Nichts zu generieren, indem etwa einer Chipkarte eine Kommunikation mit einer gültigen Partnerkarte vorgetäuscht wird. Genauso sollen Kommunikationsabbrüche oder Angriffe nicht dazu führen, dass Geld verloren geht. Details zum Mondex Protokoll und seiner Implementierung finden sich in Kap. 12.

Schematisch ist die Anwendung in Abb. 2.2 gezeigt.

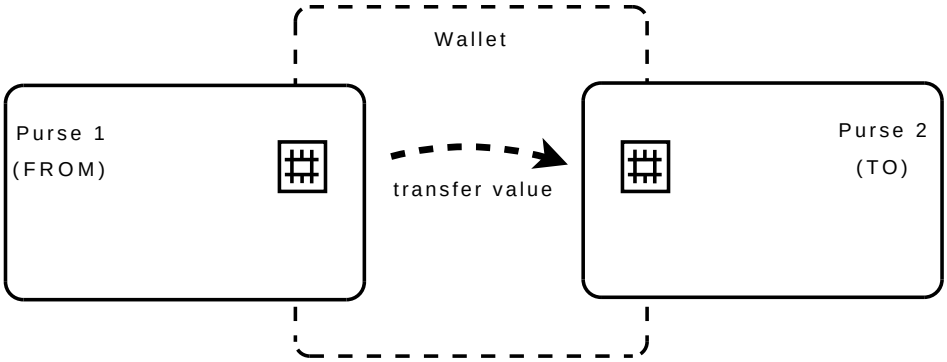


Abbildung 2.2: Die Mondex Applikation

Mondex wurde als Fallstudie insbesondere dadurch bekannt, weil es sich hierbei um die erste Anwendung handelt, die nach dem Zertifizierungslevel ITSEC E6 zertifiziert wurde. Dazu ist auch die Anwendung formaler Methoden vorgeschrieben. Die entsprechende formale Untersuchung der Mondex Fallstudie findet sich in der Arbeit von Stephney, Cooper und Woodcock [174]. Bei dieser in Z [171] durchgeführten Formalisierung wurden keine rechnergestützten Beweissysteme verwendet, alle Beweise wurden auf Papier geführt.

Kürzlich erfuhr die Mondex Fallstudie neue Beachtung, weil ihre maschinengestützte Verifikation als Teil der Grand Challenges in Software Verification [82] als Challenge für Theorembeweissysteme ausgerufen wurde [99] [187]. Einige Gruppen beteiligten sich dabei mit großem Erfolg an der Verifikation der Fallstudie, so z.B. [146] [13] [78] [188] [109] [165]. Auch die Gruppe des Autors führte die Originalbeweise mittels des KIV Systems durch [161]. Wir entdeckten dabei einige kleinere Fehler der originalen Verifikation und änderten in einem zweiten Schritt auch das Kommunikationsprotokoll der Fallstudie leicht ab, um einen vorher nicht beachteten Angriff zu vermeiden [160].

Des Weiteren wurden in der Gruppe des Autors zwei Erweiterungen der Fallstudie durchgeführt: Zum Einen erweiterten wir diese um den Entwurf und Sicherheitsnachweis eines kryptographischen Protokolls [73]. Spezifikation und Verifikation fand hier anhand der Prosecco Methodik (s. Kap. 3.5) statt. Zum Anderen erweiterten wir die Fallstudie im Rahmen dieser Dissertation um die Verifikation einer tatsächlichen Implementierung der Fallstudie.

Mondex dient in dieser Arbeit als Beispiel für die Anwendung des Ansatzes auf Java Chipkarten und JavaCard. Dies wird in Kap. 11 und 12 beschrieben.

Kapitel 3

Grundlagen

Dieses Kapitel erläutert kurz alle Grundlagen, die für die folgenden Kapitel benötigt werden. Dies umfasst dynamische Logik und algebraische Spezifikationen und ihre Implementierung im KIV System, dem in dieser Arbeit verwendeten Beweissystem. Ebenso wird der Prosecco Ansatz zur Spezifikation und Verifikation von kryptographischen Protokollen vorgestellt, auf dem sich das Verfeinerungskonzept abstützt. Schließlich werden die Grundlagen des in KIV implementierten Java Kalküls sowie die nutzbaren Verfeinerungstheorien Data Refinement und ASM Invarianten Refinement vorgestellt.

3.1 Dynamische Higher Order Logik und der Sequenzenkalkül

3.1.1 Syntax und Semantik

Dynamische Higher Order Logik (DL) [76] [77] erweitert Higher Order Prädikatenlogik (s. z.B. [170]) um Programmformeln. Programme in DL bestehen aus imperativen Programmkonstrukten, wie sie etwa auch in Pascal vorkommen. Insbesondere ist lokale Variablendeklaration mit Initialisierung **var** x = y **in** α, Zuweisung x := y, bedingte Verzweigung **if** φ **then** α **else** β, sequentielle Komposition $\alpha;\beta$ und die Schleife **while** φ α **else** β enthalten. Prozedurdeklaration P#(x; y) { α } und Prozeduraufruf P#(x; y) mit Eingabeparametern x und Ein-/Ausgabeparametern y sind möglich. Aus theoretischen Gründen gibt es noch das leere Programm **skip** und das nicht-terminierende Programm **abort** . Mehrere Zuweisungen können in der Logik auch parallel ausgeführt werden. Sie werden dann durch Kommata getrennt (x := y, z := w). Dies ist allerdings nur möglich, wenn dadurch keine Konflikte entstehen können (wie bspw. x := y, z := x).

Programmformeln in DL umfassen die Modaloperatoren Box $[\alpha]\ \varphi$, Diamond $\langle\alpha\rangle\ \varphi$ und Strong-Diamond $\langle\!|\alpha|\!\rangle\ \varphi$, wobei α eine Programmformel und φ eine Formel ist. Dabei bedeutet

$[\alpha\ ]\ \varphi$	wenn α terminiert gilt danach die Formel φ
$\langle\alpha\ \rangle\ \varphi$	es gibt einen Ablauf von α, der terminiert und danach gilt die Formel φ
$\langle\!\|\alpha\ \|\!\rangle\ \varphi$	α terminiert immer und danach gilt die Formel φ

Der Unterschied zwischen Strong Diamond und Diamond kommt nur bei indeterministischen Programmen zum Tragen. Für diese Programme ist das **choose** Konstrukt integriert. Dabei belegt **choose** x **with** φ **in** α **else** β die Variable x mit einem indeterministischen Wert, für den φ gelten muss, und führt anschließend α aus. Wenn kein solcher Wert für x existiert wird β ausgeführt. Der **else** Fall ist dabei optional und ersetzt ein ansonsten implizites **abort**. Als Spezialfall von **choose** existiert die Zufallszuweisung $\mathsf{x} := [?]$, die äquivalent zu **choose** x **with true** ist.

Im Weiteren beinhaltet die hier verwendete Logik Higher Order Konstrukte, also insbesondere lambda-Terme $\lambda x.y$, Funktionsvariablen $f : T_1 \times \ldots \times T_n \to T$ und Funktionsupdates. Ein Funktionsupdate $f[x_1, \ldots, x_n, y]$ ändert dabei den Wert der n-stelligen Funktion f an der Stelle $x_1, \ldots, x_n$ auf y. Es gilt

$$\mathsf{f}[\underline{\mathsf{x}}, \mathsf{y}] = \lambda\, \underline{\mathsf{z}}.\ \underline{\mathsf{z}} = \underline{\mathsf{x}} \supset \mathsf{y}\ ;\ \mathsf{f}(\underline{\mathsf{z}})$$

Für Updates im ersten Argument einer Funktion kann vereinfachend $f := f[x_1, \ldots, x_n, y]$ auch als $f(x_1, \ldots, x_n) := y$ geschrieben werden.

Die Syntax und Semantik der DL findet sich in [11].

3.1.2 Beweise

Beweise in Dynamischer Higher Order Logik werden in einem auf Gentzen zurückgehenden Sequenzenkalkül [52] geführt. Dabei ist eine Sequenz

$$\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$$

eine Abkürzung für die Formel

$$\varphi_1 \wedge \ \ldots\ \wedge \varphi_n \to \psi_1 \vee\ \ldots\ \vee \psi_m$$

$\varphi_1, \ldots, \varphi_n$ heißt dabei der Antezedent, $\psi_1, \ldots, \psi_m$ der Sukzedent der Sequenz.

Eine Kalkülregel des Sequenzenkalküls hat die Form

$$\frac{\mathsf{P}_1,\ \ldots,\ \mathsf{P}_n}{\mathsf{C}}$$

Dabei sind $P_1, \ldots, P_n$ (genannt die Prämissen) als auch C (die Konklusion) Sequenzen. Die Semantik einer Sequenzenregel ist: Wenn die Prämissen gelten, gilt auch die Konklusion. Regeln ohne Prämissen heißen Axiome. Beweise im Sequenzenkalkül sind dabei iterierte Anwendungen von Deduktionsregeln bis zur Reduktion auf Axiome.

Als Beispiel sei die Regel für eine Konjunktion im Sukzedent angegeben. Wir verwenden hier Γ und Δ als Abkürzung für beliebige Formellisten.

$$\text{conjunction right} \qquad \frac{\Gamma \vdash \varphi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}$$

Die Basisregeln für die Prädikatenlogik werden erweitert um Regeln für die Verifikation von Programmen. Als Beispiel seien die beiden Regeln für das **if** Konstrukt angegeben:

$$\frac{\Gamma, \varepsilon \vdash \langle \alpha \rangle\ \varphi, \Delta \qquad \Gamma, \neg\, \varepsilon \vdash \langle \beta \rangle\ \varphi, \Delta}{\Gamma \vdash \langle \textsf{if } \varepsilon \textsf{ then } \alpha \textsf{ else } \beta \rangle\ \varphi, \Delta} \qquad \frac{\varepsilon, \langle \alpha \rangle\ \varphi, \Gamma \vdash \Delta \qquad \neg\, \varepsilon, \langle \beta \rangle\ \varphi, \Gamma \vdash \Delta}{\langle \textsf{if } \varepsilon \textsf{ then } \alpha \textsf{ else } \beta \rangle\ \varphi, \Gamma \vdash \Delta}$$

3.2 Algebraische Spezifikationen und das KIV System

Das KIV System [12] [75] [149] [148] [150] [70] wird in dieser Arbeit als Spezifikations- und Beweissystem verwendet. Das System erlaubt es, strukturierte algebraische Spezifikationen zu erstellen und darauf aufbauend interaktiv u.a. mit dem Sequenzenkalkül Beweise zu führen. Dabei stehen die für Spezifikationshierarchien bekannten Operationen wie enrichment, union, parameterization und actualization zur Verfügung [9]. Frei erzeugte Datentypen können durch spezielle data specifications spezifiziert werden, für die die entsprechenden Axiome für den Datentyp generiert werden. Als Axiome können neben prädikatenlogischen Formeln auch Programmformeln verwendet werden. Die Syntax der Spezifikationen - soweit in dieser Arbeit verwendet - sollte selbsterklärend sein. Die Semantik einer Spezifikation ist die Menge aller Modelle (lose Semantik).

Neben Prädikatenlogik und dynamischer Higher Order Logik wurde im KIV System auch Unterstützung für Temporallogik [10] und Statecharts [178] [16] implementiert. In dieser Arbeit wird insbesondere der ebenso im KIV System implementierte Java Kalkül (s. Kap. 3.6 sowie [172] [173]) und die Einbettung von Abstract State Machines in DL nach dem Ansatz von [155] verwendet.

KIV stellt sehr guten Beweissupport für die genannten Logiken zur Verfügung. Das System wurde bereits für zahlreiche große Fallstudien eingesetzt [162] [137] [57] [159] und kontinuierlich erweitert und verbessert. Insbesondere existiert heute eine sehr große Standardbibliothek für Datentypen wie natürliche Zahlen, Integers, Listen oder Mengen mit tausenden Axiomen und Theoremen. Diese Datentypen bilden die Grundlage für die in dieser Arbeit verwendeten Spezifikationen. Viele dieser Spezifikationen können im Internet eingesehen werden [104].

3.3 Abstract State Machines

Abstrakte Zustandsmaschinen [69] [27] (ASM) sind ein Spezifikationsformalismus basierend auf der Idee von Zustandsübergangssystemen. Dabei ist ein Zustandsübergangssystem in seiner allgemeinsten Form ein Quadrupel $\mathsf{Sys} = (\mathsf{S}, \mathsf{IN}, \mathsf{OUT}, \mathsf{RULE})$ mit einer Zustandsmenge S, Startzuständen $\mathsf{IN} \subseteq \mathsf{S}$, Endzuständen $\mathsf{OUT} \subseteq \mathsf{S}$ und einer Zustandsübergangsrelation $\mathsf{RULE} \subseteq \mathsf{S} \times \mathsf{S}$.

ASMs sind eine sehr allgemeine Form von Zustandsübergangssystemen, da ihre Zustandsmenge S eine Menge von *Algebren* über einer gegebenen Signatur ist. Die Zustandsübergänge (genannt ASM Regeln oder kurz Regeln) sind demnach Modifikationen der Algebra des derzeitigen Zustands. Diese Modifikationen werden in einer programmiersprachlichen Notation angegeben. Dabei ist die einfachste und grundlegendste Modifikation einer Algebra die Modifikation einer Funktion (Funktionsupdate) dieser Algebra. Die einfachste Form einer ASM Regel R ist demnach

$$\mathsf{R} \equiv \mathsf{f}(\underline{\mathsf{p}}) := \mathsf{v}$$

wobei $\underline{p}$ und v variablenfreie Terme sind. Die Semantik eines solchen Funktionsupdates ausgeführt in einer Algebra $\mathcal{A}$ ist die Berechnung einer neuen Algebra $\mathcal{B}$, die zu A identisch ist, mit der Ausnahme dass die Funktion f_A so modifiziert wurde, dass $\mathsf{f}_\mathcal{B}$ an der Stelle $[\![\underline{p}]\!]_\mathcal{A}$ zu $[\![v]\!]_\mathcal{A}$ auswertet. Nullstellige Funktionen (die in der dynamischen Logik Konstanten sind) werden damit in ASMs zu *Variablen*.

Zusätzlich zu dieser Grundregel existieren noch viele weitere Sprachkonstrukte in ASMs. In dieser Arbeit verwenden wir lediglich

- bedingte Verzweigung **if** φ **then** R_0 **else** R_1
- lokale Variablendeklaration mit Initialisierung **let** x = y **in** R
- parallele Ausführung $\begin{matrix} R_0 \\ R_1 \end{matrix}$
- indeterministische Auswahl **choose** x **with** φ **in** R
- Makrodeklaration $R\{...\}$

Diese Sprachkonstrukte entsprechen denen, die bereits in der dynamischen Logik des KIV Systems zur Verfügung stehen. ASM Makros können damit durch DL Prozeduren abgebildet werden. Die gesamte Einbettung beruht dabei auf der Einbettung von ASMs in DL aus der Arbeit von Schellhorn [155]. Dabei werden innerhalb paralleler Ausführung nur konfliktfreie Funktionsupdates erlaubt, keine komplizierteren Sprachkonstrukte.

Initialzustände der ASM werden in KIV durch ein $\mathsf{init} : \mathsf{S}$ Prädikat angegeben, Finalzustände dementsprechend durch $\mathsf{final} : \mathsf{S}$. Die iterierte Anwendung der Zustandsübergangsrelation RULE bis zum Erreichen eines Endzustands wird durch eine DL Prozedur abgebildet, die folgende Form hat:

```
ASM(input; state) {
   while (¬ final(state)) RULE(input; state)
}
```

Dabei ist der optionale input eine Eingabe an das System. In unserem Fall wird input nicht benötigt.

3.4 Theorien zur Verfeinerung

Verfeinerung (Refinement) ist eine Technik zum Korrektheitsnachweis von Systemen. Ganz allgemein bezeichnet man dabei ein konkretes System als Verfeinerung eines abstrakten Systems, wenn jeder Zustandswechsel des konkreten Systems auch im abstrakten System möglich ist.

In der aktuellen Forschung wurden eine Vielzahl von verschiedenen Verfeinerungstheorien entwickelt, z.B. [79], [44], [45], [25], [158]. In dieser Arbeit werden die zwei Verfeinerungstheorien *Data Refinement* [79] und *ASM Refinement für Invarianten* [157] verwendet. Während aus theoretischer Sicht das einfachere Data Refinement für den hier vorgestellten Verfeinerungsansatz bereits ausreichend wäre, ist doch ASM Refinement für einen durchgängigen Ansatz bei der Betrachtung von Spezifikationen mit Abstract State Machines die natürlichere Wahl als Verfeinerungstheorie. Daher werden beide Theorien im Folgenden kurz vorgestellt.

3.4.1 Data Refinement

Data Refinement ist eine grundlegende Verfeinerungstheorie basierend auf den Arbeiten von He, Hoare und Sanders [79]. Die Verfeinerungsbeziehung wird dabei für zwei Datentypen ADT und CDT definiert. Wir verwenden folgende Definitionen:

Definition 3.4.1 ***Datentyp*** *Ein Datentyp* DT *ist ein Quintupel* $(\mathsf{GS}, \mathsf{S}, \mathsf{INIT}, \{\mathsf{OP}_i\}_{i\in I}, \mathsf{FIN})$ *mit einer Menge globaler Zustände* GS *und einer Menge lokaler Zustände* S. $\mathsf{INIT} \subseteq \mathsf{GS} \times \mathsf{S}$ *ist die Initialisierungsrelation,* $\mathsf{FIN} \subseteq \mathsf{S} \times \mathsf{GS}$ *ist die Finalisierungsrelation für den Datentyp. Die* $\mathsf{OP}_i \subseteq \mathsf{S} \times \mathsf{S}$ *sind die auf dem Datentyp möglichen Operationen.* I *ist eine Menge von Indizes für die Operationen auf dem Datentyp. Alle betrachteten Relationen sind total.*

Wir benutzen $\mathsf{ADT} = (\mathsf{GS}, \mathsf{AS}, \mathsf{AINIT}, \{\mathsf{AOP}_i\}_{i\in I}, \mathsf{AFIN})$ und $\mathsf{CDT} = (\mathsf{GS}, \mathsf{CS}, \mathsf{CINIT}, \{\mathsf{COP}_i\}_{i\in I}, \mathsf{CFIN})$ für die abstrakte bzw. die konkrete Ebene.

Die möglichen Abfolgen von Operationen auf einem Datentyp ergeben nun eine Relation von globalen Zuständen, die alle Zustandsübergänge des Datentyps angibt. Diese Relation nennen wir PROG (APROG bzw. CPROG). Mit ; als Relationskomposition und $\{\mathsf{OP}_i\}^*$ für die beliebige iterierte Ausführung beliebiger OP_i ist PROG definiert als:

Definition 3.4.2 ***Programm*** *Die Relationskomposition* $PROG := INIT; \{OP_i\}^*; FIN$ *über einem Datentyp* DT *wird sein Programm genannt.*

Ein korrektes Data Refinement soll nun garantieren, dass CPROG das durch APROG vorgegebene Verhalten erhält. Dazu soll als Verfeinerungsbegriff gelten:

Definition 3.4.3 ***Korrekte Verfeinerung*** *Ein konkretes Programm* CPROG *verfeinert ein abstraktes Programm* APROG *gdw. CPROG* $\subseteq$ *APROG.*

Ein solcher Verfeinerungsbegriff garantiert nun trivial, dass Eigenschaften $\varphi(gs, gs')$, die für alle (gs, gs') $\in$ APROG gezeigt wurden (unter APROG also invariant sind), auch für alle (gs, gs') $\in$ CPROG gelten. Später in dieser Arbeit wird APROG eine Protokollspezifikation und CPROG eine Implementierung sein und es wird damit die Vererbung einer (invarianten) Sicherheitseigenschaft φ auf die Implementierung gezeigt.

Der Refinementbegriff wird gezeigt, indem für jede Einzeloperation von CDT gezeigt wird, dass es eine entsprechende Operation von ADT mit "ähnlichem" Verhalten gibt. Um Ähnlichkeit zwischen der konkreten und abstrakten Ebene auszudrücken, wird im Data Refinement eine Simulationsrelation R $\subseteq$ AS $\times$ CS definiert, die den Zusammenhang zwischen beiden Welten angibt.

Dies kann man sich vorstellen wie in Abb. 3.1 gezeigt.

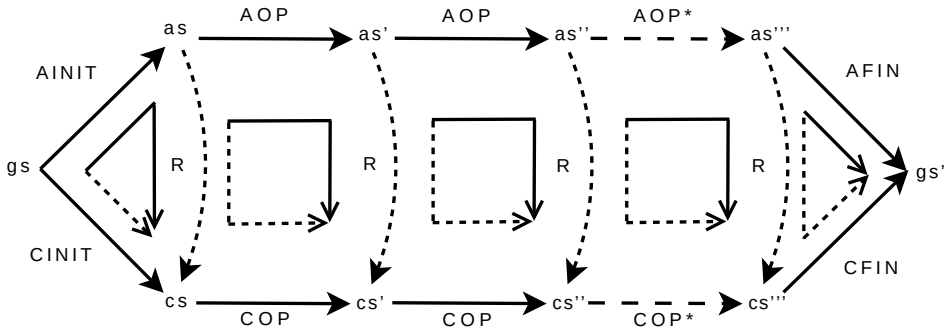


Abbildung 3.1: Data Refinement

Daraus resultieren die drei Beweisverpflichtungen des Data Refinements:

Lemma 3.4.1 *Ein Programm* CPROG *verfeinert ein Programm* APROG *wenn folgende Bedingungen gelten:*

- *Initialisierung:* CINIT(gs, cs) $\rightarrow$ AINIT(gs, as) $\wedge$ R(as, cs)
- *Korrektheit:* $\forall$ i $\in$ I. R(as,cs) $\wedge$ COP_i(cs,cs') $\rightarrow$ $\exists$ as'. AOP_i(as,as') $\wedge$ R(as', cs')
- *Finalisierung:* R(as, cs) $\wedge$ CFIN(cs, gs) $\rightarrow$ AFIN(as, gs)

Beweis: einfach per Induktion.

3.4.2 ASM Refinement für Invarianten

ASM Refinement wird in [155] [25] als Verfeinerungsbegriff für Abstract State Machines vorgestellt. Die zugrundeliegende Idee ist es, den traditionellen Refinementbegriff zu generalisieren. Eine entsprechende Generalisierungsbeziehung wird in [158] gezeigt. Wir wiederholen hier die wichtigsten Definitionen aus [156] und [157] und passen diese an die hier verwendete Notation an. Insbesondere gilt, dass eine Verfeinerung, die nach der Data Refinement Theorie korrekt ist, auch eine korrekte Verfeinerung im Sinne des ASM Refinements für Invarianten ist.

Verfeinerungsbegriffe

Der Verfeinerungsbegriff von ASM Refinement für zwei ASMs AM = (AS, AIN, AOUT, ARULE) und CM = (CS, CIN, COUT, CRULE) basiert zunächst auf zwei Relationen IR $\subseteq$ AIN $\times$ CIN und OR $\subseteq$ AOUT $\times$ COUT. Diese geben den gewünschten Zusammenhang zwischen initialen und finalen Zuständen der beiden Maschinen an. Ein Ablauf σ_a (und analog σ_c) ist dabei eine (potentiell unendliche) Folge von Zuständen, welche in einem Zustand aus AIN beginnt, deren aufeinanderfolgende Zustände jeweils durch Anwendung von ARULE entstanden sind und im Falle endlicher Abläufe der letzte Zustand aus AOUT stammt. $\sigma(j)$ bezeichnet den j-ten Zustand des Ablaufs σ.

Damit ist ein erster Verfeinerungsbegriff:

Definition 3.4.4 ***Erhalt partieller und totaler Korrektheit*** *Eine Verfeinerung von einer ASM* AM *zu einer ASM* CM *erhält partielle Korrektheit bzgl.* IR *und* OR*, wenn für jeden endlichen Ablauf σ_c von CM ein endlicher Ablauf von σ_a von AM existiert, deren jeweilige Anfangszustände über* IR *und deren Endzustände über* OR *in Relation stehen. Sie erhält totale Korrektheit, wenn sie partielle Korrektheit erhält und zu jedem unendlichen Ablauf von* CM *ein unendlicher Ablauf von* AM *existiert, deren Anfangszustände über* IR *in Relation stehen.*

Dies ist allerdings noch nicht genug, um einen aussagekräftigen Korrektheitsbegriff für reaktive Systeme zu spezifizieren, da hier nicht nur Anfangs- und Endzustände wichtig sind, sondern auch die Zwischenzustände. Daher definiert man darauf aufbauend partiellen und totalen Erhalt von Abläufen bzgl. einer Simulationsrelation IO:

Definition 3.4.5 ***partieller und totaler Erhalt von Abläufen*** *Eine Verfeinerung von einer ASM* AM *zu einer ASM* CM *erhält total Abläufe bzgl. einer Simulationsrelation* IO *wenn sie totale Korrektheit bzgl. $IR := IO \cap (AIN \times CIN)$ und $OR := IO \cap (AOUT \times COUT)$ erhält und für jeden unendlichen Ablauf σ_c ein unendlicher Ablauf σ_a und zwei streng monotone Folgen $(i_0 < i_1 < \ldots)$ und $(j_0 < j_1 < \ldots)$ existieren, so dass für jedes k $IO(\sigma_a(i_k), \sigma_c(j_k))$ gilt. Für partiellen Erhalt von Abläufen muss die Verfeinerung partielle Korrektheit erhalten und die Sequenz $(i_0 < i_1 < \ldots)$ muss lediglich monoton sein.*

Dieser Verfeinerungsbegriff besagt intuitiv, dass eine Simulationsrelation in allen *betrachteten* Zuständen gelten muss. Gerade für die hier angestrebte Vererbung von Sicherheitseigenschaften der abstrakten Ebene ist dies jedoch zu allgemein. Die hier betrachteten Sicherheitseigenschaften sind meist Invarianten. Eine Invariante ist hierbei eine Eigenschaft, für die gilt:

Definition 3.4.6 ***Invariante***
Eine Eigenschaft $\mathsf{I} : \mathsf{S} \times \mathsf{S} \rightarrow \mathsf{bool}$ *ist eine Invariante einer ASM* AM = (S, IN, OUT, RULE) *falls*

- I(s,s) *für alle* s ∈ IN *gilt*
- *falls* I(s,s') *und* (s',s") ∈ RULE, *gilt auch* I(s,s")

Als Besonderheit erlauben wir in unserer Definition von Invarianten die Referenzierung des Anfangszustandes der ASM. Dies ist später für die Ausdrückbarkeit von Sicherheitseigenschaften wichtig, z.B. wenn die Unveränderheit eines Datums über einen Protokolllauf ausgedrückt werden soll.

Betrachtet man nun nicht mehr *alle* Zustände in der konkreten Ebene der Verfeinerung, kann es sein, dass die Korrektheit der Verfeinerung nicht mehr garantiert, dass eine entsprechende abstrakte Invariante auch auf der konkreten Ebene eine Invariante ist. Eine konkrete Invariante CINV zu einer abstrakten Invariante AINV ist dabei definiert als

$$\mathsf{CINV(cs, cs')} \leftrightarrow \exists\ \mathsf{as.\ IR(as, cs)} \wedge \mathsf{IO(as', cs')} \wedge \mathsf{AINV(as, as')}$$

Daher beschränken wir den Verfeinerungsbegriff:

Definition 3.4.7 ***partieller und totaler Erhalt von Invarianten*** *Eine Verfeinerung von einer ASM* AM *zu einer ASM* CM *erhält total Invarianten bzgl. einer Simulationsrelation* IO *wenn sie totale Korrektheit bzgl.* $IR := IO \cap (AIN \times CIN)$ *und* $OR := IO \cap (AOUT \times COUT)$ *erhält und für jeden unendlichen Ablauf* σ_c *ein unendlicher Ablauf* σ_a *und eine streng monotone Folge* $(i_0 < i_1 < \ldots)$ *existieren, so dass für jedes* k $IO(\sigma_a(i_k), \sigma_c(k))$ *gilt. Für partiellen Erhalt von Invarianten muss die Verfeinerung partielle Korrektheit erhalten und die Sequenz* $(i_0 < i_1 < \ldots)$ *muss lediglich monoton sein.*

Ein solcher Verfeinerungsbegriff vererbt nun Invarianten von AM auf CM (s. auch [157]), eine Invariante AINV nach obiger Definition gilt damit als CINV (ebenfalls nach obiger Definition) in der konkreten ASM.

Beweisstrategie

Zum Beweis einer korrekten Verfeinerung und des Erhalts von Invarianten werden bei der ASM Verfeinerungstheorie nun beliebige m:0..1 Beziehungen zwischen den Operationen von AM und CM erlaubt, während Data Refinement eine 1:1 Beziehung zwischen abstrakten und konkreten Operationen vorschreibt.

Im Weiteren verwendet man zum Beweis eine stärkere Kopplungsrelation INV zwischen abstrakter und konkreter Welt, die zusätzlich neben dem Mapping der Zustände auch noch Invarianten über beide Ebenen enthalten kann. Auch ASM Refinement folgt hierbei der Beweisidee

des "Zusammensetzens" von kommutierenden Diagrammen, ähnlich dem Data Refinement. Es sind nun neben 1:1 auch m:0..1 Diagramme möglich. Abb. 3.2 stellt dies allgemein dar.

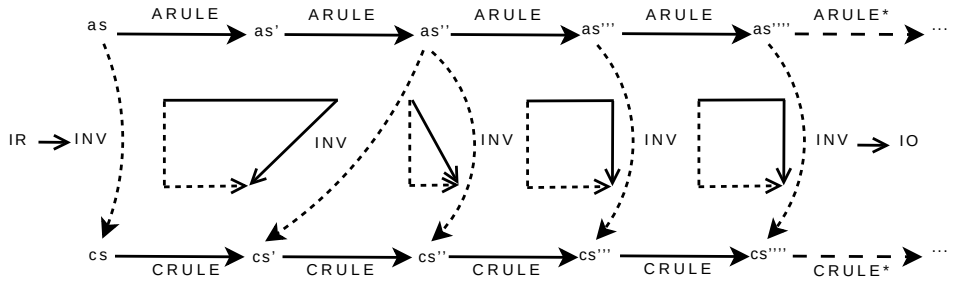


Abbildung 3.2: ASM Refinement für Invarianten

Die zu zeigenden Beweisverpflichtungen ergeben sich hierbei (mit der Einbettung von ASMs in DL aus Kap. 3.3) wie folgt:

Lemma 3.4.2 *Eine ASM* CM *erhält partiell Invarianten bzgl. einer ASM* AM *und einer Simulationsrelation* IO*, wenn folgende Eigenschaften gelten:*

Initialisierung:
$\mathsf{cs} \in \mathsf{CIN} \rightarrow \exists\, \mathsf{as}.\ \mathsf{IR}(\mathsf{as}, \mathsf{cs}) \wedge \mathsf{as} \in \mathsf{AIN}$

Herstellen der Invariante:
$\mathsf{IR}(\mathsf{as}, \mathsf{cs}) \rightarrow \mathsf{INV}(\mathsf{as},\mathsf{cs})$

Simulationsrelation:
$\mathsf{INV}(\mathsf{as}, \mathsf{cs}) \rightarrow \mathsf{IO}(\mathsf{as},\mathsf{cs})$

Korrektheit:
$$\begin{array}{ll} & \mathsf{INV}(\mathsf{as},\mathsf{cs}) \wedge \mathsf{as} = \mathsf{as}_0 \wedge \neg\, \mathsf{cs} \in \mathsf{COUT} \\ \rightarrow & (\quad \neg\, \mathsf{as} \in \mathsf{AOUT} \\ & \quad \wedge \exists\, \mathsf{i}.\ \langle \mathsf{ARULE}(\mathsf{as})^{i+1} \rangle\ (\mathsf{INV}(\mathsf{as},\mathsf{cs}) \wedge (\mathsf{as},\mathsf{cs}) <_{m0} (\mathsf{as}_0,\ \mathsf{cs}))) \\ & \vee\ \langle\!|\mathsf{CRULE}(\mathsf{cs})|\!\rangle\ \exists\, \mathsf{i}.\ \langle \mathsf{ARULE}(\mathsf{as})^{i} \rangle\ \mathsf{INV}(\mathsf{as},\mathsf{cs}) \end{array}$$

Die Hauptbeweisverpflichtung *Korrektheit* lässt hierbei für nicht-Endzustände der konkreten Ebene $\neg\, \mathsf{cs} \in \mathsf{COUT}$ entweder m:0 Diagramme (erstes Disjunktionsglied rechts, (i+1)-fache Iteration von ARULE) oder m:1 Diagramme (zweites Disjunktionsglied, erst einmalige Iteration von CRULE und anschließende i-fache Iteration von ARULE) zu. Im Falle von m:0 Diagrammen müssen die Zustände noch bezüglich einer noetherschen Ordnung $<_{m0}$ kleiner werden, damit sichergestellt ist, dass nicht unendliche abstrakte Abläufe durch endliche konkrete verfeinert werden können.

Beweisidee von Lemma 3.4.2: Zusammensetzen kommutierender Diagramme aus Abb. 3.2,

Schluss per Induktion, s. dazu auch [157].

3.4.3 Umsetzung in KIV und Spezifikationsinstantiierung

Wichtig für diese Arbeit ist insbesondere die im KIV System implementierte Möglichkeit zur Spezifikationsinstantiierung. Dabei wird eine Subspezifikation PS (der Parameter) einer generischen Spezifikation GS mittels einer Spezifikation AS (actual specification) instantiiert. Dazu wird ein Isomorphismus σ zwischen PS und AS definiert, der die Signatursymbole von PS auf solche von AS abbildet. Anschließend müssen die Axiome von $\sigma(PS)$ als Beweisverpflichtungen mittels der Axiome von AS bewiesen werden. Kann dies gezeigt werden, kann man *AS union* $\sigma(GS)$ verwenden und es gelten somit insbesondere alle Theoreme aus GS unter dem Isomorphismus. Instantiierung wird in dieser Arbeit insbesondere dazu verwendet, die generischen Verfeinerungstheorien Data Refinement und ASM Invarianten Refinement aus den vorherigen beiden Abschnitten, die bereits im KIV System als generische Bibliothek spezifiziert sind, auf die konkreten hier behandelten Fallstudien anzuwenden. Generisch wurden dabei die Beweisverpflichtungen der Verfeinerungstheorie spezifiziert und einmal gezeigt, dass aus ihnen ihr jeweiliger Verfeinerungsbegriff folgt. Als Ergebnis der konkreten Spezifikationsinstantiierung müssen nun die Beweisverpflichtungen der Verfeinerungstheorie (PS) für die jeweilige Fallstudie (AS) gezeigt werden. Als Ergebnis erhält man den Verfeinerungsbegriff (GS) für die jeweilige Fallstudie.

3.5 Prosecco

Die Prosecco (**Pro**tocols for **Sec**ure **Co**mmunication) Methodik [72] für die Entwicklung von Kommunikationsprotokollen basiert auf der Dissertation von Haneberg [71]. Sie stellt gleichzeitig die Grundlage für den Verfeinerungsbegriff dieser Arbeit dar. Im Folgenden wird daher auf die grundlegenden Eigenschaften von Prosecco eingegangen, die in dieser Arbeit von Bedeutung sind.

3.5.1 Überblick

Prosecco beschreibt insbesondere einen formalen Spezifikationsrahmen für Kommunikationsprotokolle. Dabei gliedert Prosecco die Spezifikation des Protokollverhaltens nach den am Protokoll beteiligten Teilnehmern. Die Protokollteilnehmer werden auch *Agenten* genannt. Agenten können mit anderen Agenten kommunizieren, indem sie über Verbindungen Nachrichten austauschen. Jeder Agent hat hierzu eine bestimmte Anzahl an Kommunikationsports, die seine Ein-/Ausgabeschnittstelle darstellen. Eine Verbindung ist damit ein Tupel $(agent_1 \times port_1) \times (agent_2 \times port_2)$. Über diese Verbindungen werden nun Protokollnachrichten versendet. Dazu wird jedem Port eine Eingabequeue zugeordnet, die die an diesem Port zugestellten Nachrichten zwischenspeichert. Damit entsteht eine Kommunikationsstruktur, wie sie schematisch in Abb. 3.3 gezeigt ist.

Ziel von Prosecco ist die Ermöglichung der formalen Verifikation von Sicherheitseigenschaften auf dem spezifizierten formalen Modell. Damit dies realistisch möglich ist, muss neben ehrlichen Protokollteilnehmern (also Teilnehmern, die sich an feste Regeln zur Abarbeitung von Protokollschritten halten) auch ein Angreifer betrachtet werden. Ein Angreifer ist dabei

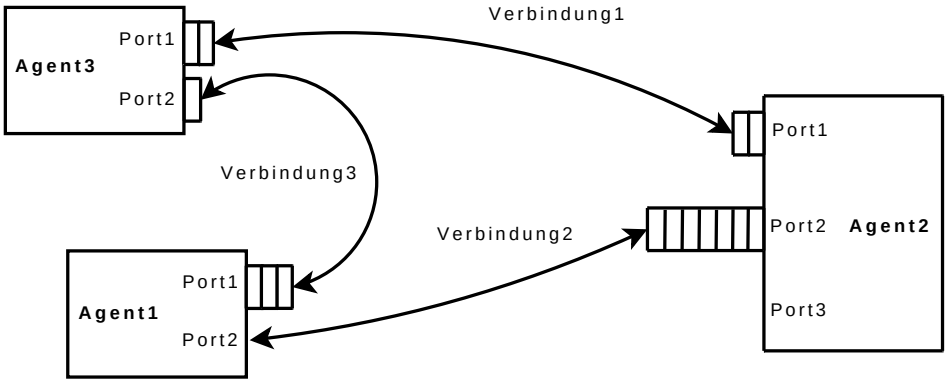


Abbildung 3.3: Kommunikationsstruktur bei PROSECCO

ein Agent, der sich nicht unbedingt so verhält, wie es das Protokoll fordert, sondern auch Nachrichten abhören, ändern, unterdrücken sowie neue Nachrichten anhand der bisher abgehörten Nachrichten erfinden kann. Der Angreifer ist dabei in PROSECCO zunächst dem allgemeinen Dolev-Yao Angreifermodell [50] nachempfunden. Bei diesem Modell besitzt der Angreifer uneingeschränkte Fähigkeiten zur Manipulation von Nachrichten, lediglich das Brechen kryptographisch gesicherter Nachrichten ist ihm hier nicht erlaubt. Neben diesem allgemeinen Modell ist es in PROSECCO auch erlaubt, das Angreifermodell abzuschwächen, falls die Domäne der betrachteten Anwendung stärkere Einschränkungen erlaubt. So kann z.B. bei einem Bankautomaten davon ausgegangen werden, dass das Gehäuse des Automaten manipulationssicher ist und somit die intern stattfindende Kommunikation zwischen Bankchipkarte und Automat nicht unterdrückt oder verändert werden kann. Dies kann unter Umständen einfachere Protokolle erlauben als bei Betrachtung des stärkeren Dolev-Yao Angreifers.

Zur formalen Spezifikation werden in PROSECCO Abstract State Machines in Kombination mit algebraischen Spezifikationen als Spezifikationssprache verwendet.

3.5.2 Algebraische Grundspezifikationen in PROSECCO

Jeder Agent besitzt einen internen Zustand, der, wie in ASMs üblich, durch dynamische Funktionen angegeben wird. Durch die Verwendung des KIV Systems kann dabei bereits auf eine große Basisbibliothek zurückgegriffen werden. Diese umfasst z.B. Datentypen wie Listen und Mengen oder auch natürliche Zahlen und Integers mit den üblichen Basisoperationen. Darauf aufbauend bietet PROSECCO eine Bibliothek für die Darstellung von Protokollnachrichten, insbesondere den Datentyp Document. Der Document Typ wird benutzt um alle Nachrichten darzustellen, die von den Agenten verschickt werden. Wenn Nachrichteninhalte im Zustand eines Agenten zwischengespeichert werden sollen, wird auch hier der Document Typ verwendet. Seine formale Spezifikation als data specification ist:

SPEZIFIKATION:

```
Document =
        IntDoc(value : int) |
```

```
                NonceDoc(nonce : Nonce) |
                SecretDoc(secret : Secret) |
                KeyDoc(key : Key) |
                HashDoc(hash : Document) |
                EncDoc(key : Key, doc : Document) |
                SigDoc(key : Key, doc : Document) |
                Doclist(docs : Documentlist)
Documentlist =
   [ ] | . + . (.first : Document, .rest : Documentlist)
Nonce = mkNonce(value : int)
Secret = mkSecret(value : int)
Key = mkKey(value : int)
```

Ein Document kann ein IntDoc sein, das einen Zahlenwert value speichert. Es kann auch ein NonceDoc sein, das eine Zufallszahl darstellt, oder ein SecretDoc, das ein nicht ratbares Geheimnis wie z.B. eine PIN ist. Ein KeyDoc ist ein Dokument, das einen kryptographischen Schlüssel speichert. Zusätzlich gibt es die Document Typen, die Ergebnisse kryptographischer Operationen vertreten. Dies sind HashDoc für Hashing-Operationen eines Document, EncDoc für die Verschlüsselung eines Document doc mit einem Schlüssel key und SigDoc für die entsprechende Signatur. Komposition von mehreren Documents erfolgt mittels dem Doclist Typ, der eine Liste von Document (Documentlist) enthält. Eine Documentlist wiederrum ist entweder die leere Liste [ ] oder eine zusammengesetzte Liste doc + docs. Im Weiteren definieren wir die Typen Nonce, Secret und Key als Inhalte der entsprechenden Dokumente.

Die Unterscheidung in diese verschiedenen Dokumenttypen ist insbesondere für die Spezifikation des Angreifermodells wichtig. Schließlich will man z.B. ausdrücken, dass ein Angreifer die Inhalte verschlüsselter Dokumente nur mit dem entsprechend passenden Schlüssel einsehen kann. Genauso ist es z.B. wichtig, dass ein Angreifer Zufallszahlen nicht erraten kann, beliebige Integers dagegen erzeugen kann. Diese Unterscheidungen werden im PROSECCO Angreifermodell über den Typ der Dokumente festgelegt. Die genaue Spezifikation dieses Angreifermodells ist für diese Arbeit nicht relevant. Sie findet sich in [71] [63]. Hier relevant ist lediglich, dass dazu das Wissen des Angreifers modelliert werden muss (intuitiv ist dies die Menge der Dokumente, die der Angreifer kennt). Dieses Wissen wird in der PROSECCO ASM als dynamische 0-stellige Zustandsfunktion attacker-known modelliert.

Ebenso lässt sich mittels der Dokumente der interne Zustand der Agenten beschreiben. So ist z.B. bei der Cindy Anwendung die Liste der derzeit auf dem Telefon gespeicherten Tickets wichtig. Die Tickets selbst werden als Doclist repräsentiert (u.a. ist hier die Nonce für den Datamatrix Code enthalten). Mehrere Tickets sind damit also eine Documentlist von Doclists. Die Tickets jedes Telefons werden somit spezifiziert als eine dynamische Zustandsfunktion tickets:

```
tickets : agent → Documentlist
```

Zusätzlich zu diesem agentenlokalen Teil des Zustands gibt es in einer PROSECCO Protokoll ASM auch einen Infrastrukturanteil, der die Kommunikationsstruktur und die versendeten Nachrichten enthält. Zur Modellierung oben vorgestellter Verbindungen wird der Datentyp Connection verwendet, der eine Verbindung zwischen zwei Endpunkten (Agent und Port, letzterer repräsentiert als natürliche Zahl) darstellt:

SPEZIFIKATION:

```
connection = mkcon(.endpoint1 : endpoint; .endpoint2 : endpoint)
endpoint = . × . (.agent : agent; .port : nat)
```

Per Lifting auf Mengen erhalten wir so eine dynamische 0-stellige Zustandsfunktion connections : connectionset für die ASM. Im Weiteren müssen noch die Eingabequeues der beteiligten Agenten modelliert werden. Dazu verwenden wir eine zweistellige Zustandsfunktion inputs : agent → nat → Documentlist. Die Funktion bildet somit Agent und Port auf eine Liste von Dokumenten ab, die sich derzeit in der Eingabequeue des Agenten befinden.

Die Vereinigung aller dieser dynamischen Funktionen der Agenten und der dynamischen Funktionen, die die Kommunikationsinfrastruktur darstellen, ergeben die Zustandsfunktionen der Protokoll ASM.

3.5.3 Spezifikation des Protokollverhaltens in PROSECCO

Das Verhalten des Gesamtsystems wird durch eine ASM Regel angegeben, die bei jeder Ausführung einen Agenten des Protokolls auswählt und seine jeweilige Funktionalität durchführt. Die Struktur der Regel für diese Protokoll-ASM folgt dabei folgendem Schema:

```
RULE(;state)
INIT(;state);
while(¬ stop) {
      STEP(;state);
      stop := [?]
}
```

Zunächst wird die ASM initialisiert, d.h. der Zustand der ASM wird auf einen definierten Ausgangszustand gesetzt. Hier werden z.B. kryptographische asymmetrische Schlüsselpaare eines Agenten mit den richtigen (zusammenpassenden) Schlüsselwerten initialisiert.

Im Anschluss wird die Hauptschleife der Protokoll-ASM ausgeführt, die in jeder Iteration die STEP Routine ausführt, welche dann die eigentliche Protokoll-Funktionalität enthält. Durch die indeterministische Zuweisung stop := [?] am Ende der Schleife und die Iteration der while Schleife bis zur Gültigkeit von stop erhält man insgesamt alle endlichen Präfixe aller möglichen Protokollläufe.

Die Protokollschrittregel der Protokoll-ASM selbst ist nun schematisch wie folgt aufgebaut:

```
STEP(;state)
choose agenttype with ∃ agent. agentok(agent, agenttype) in
    if is-agenttype1(agenttype) choose agent with agentok(agent, agenttype1) in
        AGENT1(agent;state)
    else
    if is-agenttype2(agenttype) choose agent with agentok(agent, agenttype2) in
        AGENT2(agent;state)
    else if is-attacker(agenttype) ATTACKER(agent;state)
    else ...
```

Zunächst wird ein Agententyp ausgewählt, der in dieser Iteration der Hauptschleife einen Protokollschritt ausführen soll. Dies kann ein beliebiger im Szenario existierender Agententyp sein. Die Einschränkung agentok(...) drückt aus, dass ein konkreter Vertreter dieses Agententyps existieren muss, der nun (technisch) einen Schritt ausführen kann (als reale Analogie z.B.: nur eingeschaltete Geräte können Nachrichten empfangen). Anschließend wird die Protokollfunktionalität mit einem dieser konkreten Vertreter des Typs ausgeführt.

Da die Auswahl des Agenten jeweils indeterministisch erfolgt, erhält man durch Iteration dieser ASM Regel alle möglichen Protokollabläufe (also alle möglichen Folgen von Einzelschritten).

Beispiele für Prosecco Spezifikationen finden sich in den Kapiteln zu den Fallstudien Cindy (Kap. 7) und Mondex (Kap. 13).

3.6 Java Verifikation mit KIV

Die Verifikation von Implementierungen in dieser Arbeit stützt sich auf den in KIV implementierten Java Kalkül für sequentielles Java aus der Arbeit von Stenzel [173] [172]. Der Kalkül ist in der Lage, den vollen sequentiellen Sprachumfang von Java 1.4 [100] formal zu behandeln. Es kann direkt der lauffähige Code ohne wesentliche syntaktische Umwandlungen verifiziert werden. Dazu ist es nötig, das Java Speichermodell zu axiomatisieren und darauf aufbauend Kalkülregeln für die Java Sprachkonstrukte zu definieren.

3.6.1 Das Java Speichermodell

Der Speicher (auch Heap genannt) der Java Virtual Machine (JVM) wird in KIV formal durch einen Datentyp store abgebildet. Ein store ist dabei im Wesentlichen eine Assoziationsliste, die Speicheradressen auf Java-Werte abbildet.

Die Grundidee ist dabei, Objekte und Arrays durch Referenzen eindeutig zu identifizieren. Die Inhalte der Objekte (die Felder) und die Inhalte der Arrays werden dann im store gespeichert. Die Adressierung der Inhalte erfolgt durch Kombination der Referenz des Objekts bzw. Arrays mit dem Feldnamen bzw. Arrayindex.

Formal definiert der store ein Mapping von refkey auf javavalue. Ein refkey rk = r – sk besteht dabei aus einer reference r (einem Pointer auf ein Objekt oder Array) und einem storekey sk. Ein storekey kann eine fieldspec fs (einem Bezeichner für ein Feld eines Objekts) oder ein indexkey i (ein Index in einem Array) sein. Damit bezeichnet st[r – .fs] den Speicherzugriff auf den Wert, der im Speicher st in Feld fs von Referenz r gespeichert ist. Analog ist st[r – i] ein Arrayzugriff auf Index i von Array r. Abkürzend schreiben wir später einfach st[r.fs] und st[r[i]]. Es ist zu beachten, dass die Schreibweise .fs für den Feldzugriff eine Abkürzung darstellt. Formal wird ein Feld erst eindeutig durch die Kombination von Feldname, Feldtyp und Typ der beinhaltenden Klasse charakterisiert. Der Typ fieldspec ist daher auch ein Summentyp mit genau diesen Inhalten. Aus Gründen der Lesbarkeit schreiben wir dennoch .fs mit lediglich dem Feldnamen fs, da dies in realen Beispielen meist doch ausreichend eindeutig ist. Für Java Arrayzugriffe `r[i]` verwenden wir - um den `[ . ]` Operator nicht zu sehr zu überladen - später auch die Schreibweise st[r.i], sofern eindeutig klar ist, was gemeint ist.

Updates auf dem Speicher werden mittels st[rk, val] angegeben. Dabei wird der Speicher st an refkey rk auf Wert val geändert. Es gilt:

$$st[rk, val][rk0] = (rk = rk0) \supseteq val\ ;\ st[rk0]$$

Werte im Speicher orientieren sich dabei am Typsystem von Java. Ein Speicherzugriff liefert dabei einen javavalue zurück. Formal ist dies wiederum ein Produkttyp, der z.B. ein `int` (intval(i)), ein `byte` (byteval(by)) oder ein `short` (shortval(sho)) sein kann. Dementsprechend haben wir z.B. bei einem Feld ifield mit int Typ st[r.ifield] = intval(i). Direkt auf die enthaltenen Werte greifen wir dann mittels Selektorfunktionen .intval, .byteval usw. zu. Es ist also z.B. st[r.ifield].intval = i.

Bei Pointerstrukturen werden wieder Referenzen in den Feldern der Objekte gespeichert (refval(r)). Dadurch mögliche verkettete Feldzugriffe `r.fs.fs0.fs1` können abkürzend durch Listenschreibweise dargestellt werden (st[r – (fs + fs0 + fs1)]) oder auch verkürzt als st[r.fs.fs0.fs1].

Jedes Objekt und Array besitzt darüber hinaus ein Spezialfeld `.type`, das den Typ des Objekts angibt. Bei einem Objekt mit Referenz r vom Typ `String` wäre dann st[r – .type] = typeval(mkclasstype("java.lang.String")). Bei Arrays gibt es zusätzlich noch das Feld `.length`, das die Länge des Arrays als intval enthält.

Statische Felder werden als Felder der Spezialreferenz jvmref im Speicher abgelegt. So bezeichnet st[jvmref – .statfield] den Zugriff auf das statische Feld `statfield`. Abkürzend schreiben wir hier einfach st[.statfield]. Die Spezialreferenz jvmref ist hierbei auch der Stellvertreter des Java Wertes `null`.

Schließlich beinhaltet der Store auch noch das spezielle statische Feld .mode, das angibt, wie der derzeitige Ausführungskontext für das nächste Sprachkonstrukt ist. Dies kann normale Ausführung (normal) oder ein Sprung etwa nach einem break, return oder throw sein. Des Weiteren gibt es für jede Klasse noch ein statisches Feld initstate, das den Zustand der statischen Initialisierung der Klasse angibt (initdone, initundone oder initerror nach einer Exception im statischen Initialisierer).

3.6.2 Kalkül

Der KIV Java Kalkül definiert darauf aufbauend für jedes Statement und jede Expression der Java Sprache eine Kalkülregel. Dabei werden die Modaloperatoren Box und Diamond um einen store für die Repräsentation des Kontextes des jeweiligen Programms erweitert. Es gilt somit:

$[st;\alpha]\ \varphi$	wenn α im Kontext von st terminiert gilt danach die Formel φ
$\langle st;\alpha\rangle\ \varphi$	α terminiert im Kontext von st und danach gilt die Formel φ

Ein Strong Diamond ist für Java Programme nicht notwendig, da hier kein Indeterminismus auftreten kann.

Die Kalkülregeln basieren nun auf der Idee der symbolischen Ausführung (und bilden damit die Ausführungsreihenfolge in einer JVM nach). Grundidee ist die schrittweise Reduktion des Programms durch Behandlung des jeweils nächsten in der Ausführungsreihenfolge kommenden Sprachkonstrukts. Beispielsweise gilt für das Ausführen einer `for` Schleife:

$$\frac{\begin{array}{ll}1. & \Gamma, st[.mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[.mode] = normal \vdash \langle st; \texttt{if}\ (e)\ \{\alpha\ es;\ \texttt{for}(e;es)\ \alpha\}\ \texttt{else}\ \{\}\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st; \texttt{for}(e;es)\ \alpha\rangle\ \varphi, \Delta}$$

Zunächst wird die Bedingung der `for` Schleife `e` durch Reduktion auf ein `if` getestet[1]. Bei positivem Testergebnis wird der Schleifenrumpf α sowie danach die Updates der `for` Schleife `es` ausgeführt. Anschließend wird wieder die `for` Schleife in der nächsten Iteration ausgeführt. Beides wird durch Einführung des neuen `{...}` Blocks sichergestellt. Anschließend muss nun die Nachbedingung φ gelten. Dies gilt so nur, wenn der derzeitige Ausführungsmodus normal ist (Prämisse 2). Sollte dies nicht der Fall sein, wird die Schleife einfach übersprungen (Prämisse 1) und die Nachbedingung φ muss bereits ohne die Schleife gelten.

Als Beispiel für eine Kalkülregel, die die Objektorientiertheit der Java Programmiersprache berücksichtigen muss und auch den Speicher beachten muss, betrachten wir den Methodenaufruf einer Instanzenmethode auf einem Objekt. Folgende Regel betrachtet den Aufruf der Methode `m(z)` (mit Parametervariablen $\underline{z}$) auf der Invoking Expression `e`. Die tatsächlichen Parameterwerte sind dabei die Expressions `es` (hier ist zu beachten, dass der Kalkül durch Flattening dafür sorgt, dass zum Zeitpunkt der Regelanwendung für den Methodenaufruf die Parameter und die Invoking Expression bereits vollständig ausgewertet sind). Die Regel ist damit:

$$\frac{\begin{array}{ll}1. & \Gamma, st[.mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, e = null, st[.mode] = normal \vdash \\ & \langle st; \texttt{throw new NullpointerException()}\rangle\ \varphi, \Delta \\ 3. & \Gamma, e \neq null, mode(st) = normal, \neg\ st[e.type] \in \{c_1, \ldots, c_n\} \vdash \Delta \\ 4. & \Gamma, e \neq null, mode(st) = normal, st[e.type] = c_1, \\ & this = e, \underline{z} = es \vdash \langle st; \alpha_2'\rangle\ \langle st; \texttt{targetexpr(x)}\rangle\ \varphi, \Delta \\ \vdots & \\ n+3. & \Gamma, e \neq null, mode(st) = normal, classOf(e, st) = c_n, \\ & this = e, \underline{z} = es \vdash \langle st; \alpha_3'\rangle\ \langle st; \texttt{targetexpr(x)}\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st; \texttt{x = e.m(es);}\rangle\ \varphi, \Delta}$$

[1] `for` Schleifen werden im KIV Kalkül normalisiert, d.h. die Initialisierung wird vor die Schleife verlagert

Prämisse 1 besagt, dass die gewünschte Nachbedingung φ bereits ohne Auswertung des Methodenaufrufs gelten muss, falls der aktuelle Ausführungszustand .mode nicht normal ist (denn dann wird - wenn etwa eine Exception vorher geworfen wurde - der Aufruf einfach übersprungen). Die zweite Prämisse sorgt für das Werfen einer `NullpointerException`, falls die Invoking Expression zu `null` auswertet. Die restlichen Prämissen geben die Auswertung des Methodenrumpfes an. Da in Java der Methodenaufruf dynamisch ist, kommen hier verschiedene Methodenrümpfe abhängig vom Laufzeittyp von e in Frage. Der passende Methodenrumpf für m kann dabei in einer Superklasse des statischen Typs der Invoking Expression im Quellcode definiert sein. Dann hat der Typ von e die Methode geerbt. Die Methode kann aber auch erst in diesem Typ überschrieben oder definiert worden sein. Schließlich kann der Laufzeittyp von e auch eine Subklasse des statischen Typs sein, in der wiederum die Methode überschrieben worden sein könnte. Insgesamt ergibt sich so eine Menge von Typdeklarationen mit Typen $\{c_1, \ldots, c_n\}$. Je nach tatsächlichem Laufzeittyp st[e.type] wählen nun die Prämissen 4 bis n + 3 den jeweils passenden Methodenrumpf α_i und führen ihm aus. Die `this` Variable wird dabei jeweils mit der Invoking Expression belegt, die Parametervariablen z mit ihren tatsächlichen Werten es. Das extra danach eingefügte Sprachkonstrukt `targetexpr` am Schluss des Rumpfes sorgt dafür, dass nach Rückgabe eines Wertes durch den Methodenrumpf der st[.mode] wieder von return auf normal zurückgesetzt wird und die Rückgabevariable x mit dem passenden zurückgegebenen Wert belegt wird. Prämisse 3 sorgt schließlich dafür, dass die Programmformel, sollte st[e.type] nicht einer der beschriebenen sein (und das Programm damit nicht typkorrekt), aus der Sequenz eliminiert wird.

Eine vollständige Übersicht über alle Kalkülregeln und auch über die (für diese Arbeit nicht direkt relevante) Semantik gibt die Dissertation von Stenzel [173].

3.6.3 Einige Eigenschaften von Java Strukturen

Der Speicher definiert lediglich ein reines Mapping von Speicheradressen auf Werte. Nur durch seine Definition sind noch keine Wohlgeformtheitskriterien an den Speicher gegeben. Dazu zählt insbesondere die Typkorrektheit des Speichers. In einem Feld, das im Quellcode als `int` getyptes Feld deklariert wurde, sollte zur Laufzeit auch nur ein intval(i) gespeichert sein (und nicht etwa ein refval mit einer Referenz auf ein anderes Objekt). Allein durch die Definition des Speichers ergibt sich dies noch nicht. Kalkül und Semantik sind dabei so formuliert, dass sie von der Wohlgeformtheit des Speichers ausgehen (offensive Semantik). Dies ist auch legitim, da wohlgeformte Java Programme nur wohlgeformte Speicher produzieren können.

Da die reine Axiomatisierung des Speichers noch keine Wohlgeformtheit automatisch mit sich bringt, werden Prädikate definiert, die die Wohlgeformtheit des Speichers beschreiben. Dies geschieht dabei stets für einzelne Referenzen. Auf eine ausführliche Erläuterung der Axiomatisierung dieser Prädikate wird an dieser Stelle verzichtet. Sie können in der Web Präsentation der Dissertation im Projekt javalib nachgelesen werden [49]. Da die Prädikate jedoch im späteren Verlauf häufig als Vorbedingungen für Lemmata oder in Axiomatisierungen auftauchen werden, geben wir hier eine Übersicht über die durch sie sichergestellten Eigenschaften. Wichtig ist hier noch die Tatsache, dass die Java Typdeklarationen in KIV ähnlich zu Axiomen in Spezifikationen in verschiedenen Ebenen einer Spezifikationshierarchie deklariert werden können. Damit wird es möglich, die Typdeklarationen einer low-level Spezifikationsebene (z.B. einer Bibliothek) später auf einer high-level Spezifikationsebene zu erweitern (z.B. einer konkreten Anwendung, die die Bibliothek benutzt). Dies bringt es mit sich, dass Begriffe wie z.B.

Typkorrektheit von der Ebene der derzeitigen Spezifikation abhängig werden. Betrachtet man z.B. eine Zeigerstruktur, die Referenzen auf nicht existierende Typen beinhaltet, so würde man diese als nicht typkorrekt ansehen. Werden auf einer späteren Ebene diese Typen jedoch nachträglich definiert, ist die selbe Struktur plötzlich typkorrekt. Damit werden alle Prädikate, die sich mit Eigenschaften von Pointerstrukturen beschäftigen, zusätzlich mit dem jeweilig interessanten Spezifikationslevel als String parametrisiert. Dabei gilt natürlich, dass sich z.B. die Typkorrektheit von einer niedrigen auf eine hohe Spezifikationsebene vererbt.

Das wichtigste verwendete Prädikat ist validref : string × reference × javatype × store. Es definiert die grundlegende Typkorrektheit einer Referenz.

In der Beschreibung von validref kommt der Begriff der Erreichbarkeit eines Wertes von einer Referenz vor. Erreichbar ist ein Wert val dabei von r, falls val = refval(r) selbst oder falls eine Folge von storekeys sk_0 + sk_1 + ... + sk_n existiert, so dass st[r − (sk_0 + sk_1 + ... + sk_n)] = val und die Folge mit den Typdeklarationen verträglich ist. Letzteres bedeutet, dass sk_0 ein deklariertes Feld von st[r − .type] ist, sk_1 ein deklariertes Feld von st[r − (sk_0 + .type)], usw.. Intuitiv kann man sich vorstellen, dass es einen Weg durch die von r aufgespannte Zeigerstruktur gibt, der bei r beginnt und bei val endet.

validref(str, r, ty, st) beschreibt nun mit diesem Erreichbarkeitsbegriff:

1. r ist `null` oder r besitzt ein Typfeld und st[r − .type] ist Subtyp von ty.
2. Falls st[r − .type] eine Klasse ist: Alle deklarierten Felder der Klasse sind in st enthalten und die Werte passen zu dem Typ der Felder (d.h. in primitiven Feldern stehen kompatible primitive Werte, in Klassen- bzw. Array-getypten Feldern stehen Referenzwerte).
3. Falls st[r − .type] ein Array ist: Alle indexkeys von 0 bis st[r.length] der Referenz r sind im store enthalten und die Werte passen (analog zu 2) zum Typ des Arrays.
4. Alle von r aus erreichbaren Referenzen in st sind `null` oder sie sind ungleich `null` und sind dann im Speicher enthalten und besitzen ein Typfeld .type, das einen existierenden Typ auf Spezifikationslevel str enthält.
5. Alle von r aus erreichbaren Referenzen ungleich `null` in st besitzen ein Typfeld .type, das zum Typ des die Referenz enthaltenden Feldes passt (also ein Subtyp des jeweiligen Feldtyps). Beispiel: Jede Referenz `r` auf eine Klasse `C` mit einem Feld `f` vom Typ `D` mit Wert st[r.f] ungleich `null` muss im Speicher an der Stelle st[r.f.type] ein Typwert `D` oder einen Subtyp von `D` enthalten.
6. Analog müssen alle von r aus erreichbaren primitiven Werte in Feldern oder Arrays von dem selben primitiven Typ enthalten sein (z.B. `byte[]` Arrays enthalten nur `byte`).
7. Alle von r aus erreichbaren Referenzen mit Array-Typ in st haben ein .length Feld mit einem Integerwert größer oder gleich Null und die entsprechenden Werte sind wiederum im Speicher enthalten.

Dazu existiert noch das Prädikat validrefnotnull(str, r, ty, st). Dieses Prädikat besagt zusätzlich, dass r ungleich `null` ist.

Für Referenzwerte r_1 und r_2 ist die Erreichbarkeit generisch durch das Prädikat path∃ (str, r_1, r_2, st) definiert. path∃ (str, r_1, r_2, st) bedeutet, dass es einen typkorrekten Weg durch

die Zeigerstruktur unter r_1 gibt, der bei r_2 endet. Analog kann das Auftreten von Sharing innerhalb einer Struktur definiert werden: hasSharing(str, r, st) gilt, wenn es zwei verschiedene Wege (Folgen von storekeys) gibt, die zum selben von r erreichbaren Referenzwert führen.

Schließlich benötigen wir noch die Zyklizität von Zeigerstrukturen. Diese wird für eine Referenz r generisch durch das Prädikat cyclic(str, r, st) ausgedrückt. Dabei ist eine Zeigerstruktur beginnend bei r zyklisch, wenn es von r aus eine Folge von Referenzen gibt, die jeweils voneinander erreichbar sind und die Duplikate enthält. Insbesondere bedeutet dies, dass der Zyklus nicht unbedingt die Referenz r selbst enthalten muss, sondern dieser eventuell erst später in der Struktur auftritt.

Kapitel 4

Der GoCard Entwicklungsprozess und verwandte Arbeiten

Dieses Kapitel ordnet die Arbeit in den Entwicklungsprozess für Protokollimplementierungen ein, der im Rahmen des GoCard Projekts entwickelt wurde. Dazu wird zunächst ein Überblick über die bisherigen Arbeiten im Projekt gegeben.

Im Weiteren behandelt dieses Kapitel verwandte Arbeiten anderer Gruppen auf dem Gebiet der Protokoll- und Code-Verifikation.

4.1 Der GoCard Entwicklungsprozess

In dem von der Deutschen Forschungsgemeinschaft geförderten Projekt GoCard [143] [88], dem auch der Autor angehörte, wurde die systematische toolgestützte Entwicklung von Anwendungen erforscht, die mittels kryptographischer Protokolle Sicherheitseigenschaften garantieren. Bekannte Vertreter dieser Anwendungskategorie sind Chipkarten Anwendungen. Diese Anwendungen stehen im Fokus des Interesses des Projekts. Zwei bekannte Vertreter von Chipkartenanwendungen sind etwa die deutsche Geldkarte [59] oder auch die in jedem Mobiltelefon zu findende GSM[1] SIM[2]-Karte [90]. Ebenfalls in diese Kategorie fällt der europäische elektronische Reisepass [46]. Im GoCard Projekt wurde für Anwendungen dieser Art ein Entwicklungsprozess definiert, der speziell auf deren charakteristische Anforderungen zugeschnitten ist.

Charakteristisch für Chipkartenanwendungen ist insbesondere die große Bedeutung eines speziellen kryptographischen Protokolls. So wird z.B. die deutsche Geldkarte erst durch Einsatz eines geeigneten Protokolls zu einer sicheren und benutzbaren Anwendung. Dem gegenüber ist ebenso charakteristisch, dass die restliche Funktionalität der Anwendungen meist sehr klein und einfach ist. Dagegen gehört die korrekte Entwicklung eines kryptographischen Protokolls zu den schwierigsten und fehleranfälligsten Bereichen der aktuellen Softwareentwicklung, wie

[1] Global System for Mobile communications
[2] Subscriber Identity Module

die bekannten Fehler in wichtigen Protokollen wie z.B. dem Needham-Schroeder Protokoll zur wechselseitigen Authentifizierung [118] belegen.

Damit sind herkömmliche moderne Software Entwicklungsparadigmen wie z.B. der Unified Process [98] für Chipkarten nicht adäquat, da diese ihren Fokus auf Domänenmodellierung oder auf Modellierung von algorithmischem Interaktionsverhalten zwischen Komponenten legen - beides Aspekte, die bei Chipkarten nur sehr begrenzt wichtig sind.

Darüber hinaus bieten moderne Software Entwicklungsprozesse nur sehr begrenzte Möglichkeiten, die Ergebnisse des Softwareentwurfs und der Modellierung noch vor der Implementierung zu testen oder zu verifizieren - geschweige denn eine Korrektheitsaussage über den entstandenen Quellcode abgeben zu können. Gerade bei Chipkarten ist es jedoch sehr erstrebenswert, bereits über dem Modell Korrektheitsaussagen bezüglich dem Kommunikationsprotokoll treffen zu können. Schließlich liegt eine erste Schwierigkeit bei diesen Anwendungen bereits in dem korrekten Design der Protokolle.

Das GoCard Projekt begegnet diesen Unzulänglichkeiten aktueller Software Entwicklungsparadigmen mit starker Formalisierung und Modellbildung während des Entwurfs bei gleichzeitigem Einsatz von Standard-Diagrammen der Unified Modelling Language (UML, [177] [154]). Einen Überblick über den Ansatz gibt Abb. 4.1.

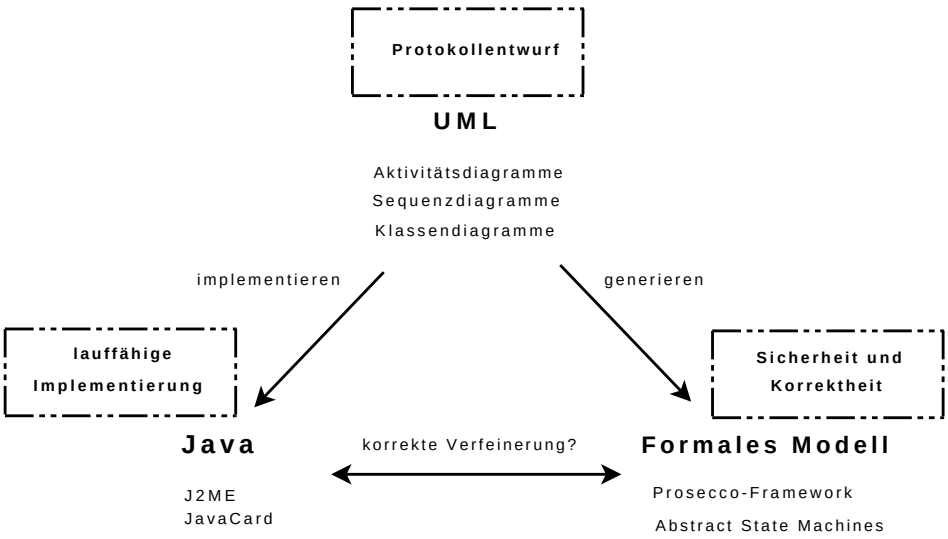


Abbildung 4.1: Der GoCard Entwicklungsprozess

Der Protokollentwurf wird hierbei mittels UML Diagrammen durchgeführt. Insbesondere finden hier UML Sequenzdiagramme, Klassendiagramme, Use Cases, Deployment Diagramme und Aktivitätsdiagramme Anwendung. Ziel auf dieser Ebene ist es, eine verständliche und einfache Notation speziell für Kommunikationsprotokolle zur Verfügung zu stellen, um den Protokollentwurf möglichst optimal zu unterstützen. Erreicht wird dieses Ziel durch Definition von UML Stereotypen und Tagged Values. Eine genauere Beschreibung des Modellierungsansatzes am Beispiel der Mondex Anwendung findet sich in [129].

Ausgehend vom Ergebnis der semiformalen Modellierung wird nun ein modellgetriebener Ansatz verwendet, um ein formales Modell zu generieren. Das formale Modell verwendet den

in Kap. 3.5 beschriebenen Ansatz zur Spezifikation der Protokollfunktionalität und des Angreifermodells mittels Abstract State Machines. Auf dieser Spezifikationsebene können nun Sicherheits- und Korrektheitseigenschaften für das Protokoll formal nachgewiesen werden.

Schließlich wird in einem dritten Schritt das Protokoll implementiert. Dazu wird im GoCard Projekt die Programmiersprache Java verwendet. Damit steht, neben der Eignung für eine Vielzahl an Geräten von der Chipkarte in JavaCard oder dem Mobiltelefon in J2ME, auch eine volle Verifikationsunterstützung im KIV System (s. Kap. 3.6) zur Verfügung. Mittels dem dort definierten Kalkül kann nun in einem letzten Arbeitsschritt der implementierte und auf echten Geräten lauffähige Quellcode formal als korrekt nachgewiesen werden. Dieser letzte Schritt ist der Schwerpunkt dieser Arbeit.

Exemplarisch verfolgt das GoCard Projekt damit die Beantwortung von Fragen der folgenden Art:

- Ist mein Protokoll sicher gegen Replay-Attacken?
- Kann ein Angreifer Nachrichten so fälschen, dass er Vorteile daraus ziehen kann?
- Können nur ehrliche Benutzer eine erfolgreiche Authentifizierung durchführen?
- Kann ein Angreifer Nachrichten so abändern, dass er einen Vorteil daraus ziehen kann?
- Setzt eine Implementierung das Design richtig um?
- Was bedeutet überhaupt Korrektheit bezüglich einer abstrakten Spezifikation eines Protokolls?
- Führen mögliche Programmierfehler zu neuen Sicherheitslücken, die nur in der konkreten Implementierung existieren und im abstrakten Modell nicht auftreten können?

Insbesondere liegen dagegen folgende Fragen *nicht* im Interesse des Projektes:

- Kann ein Virus auf dem Endgerät meiner Anwendung schaden? (s. dazu z.B. [176])
- Führt meine Hardware meine Programme korrekt aus? (s. dazu z.B. [110])
- Wie entwickelt man eine gute Bedienoberfläche für sicherheitskritische Anwendungen? (s. dazu z.B. [17] [181])
- Welche kryptographischen Algorithmen sollte ich für mein Protokoll einsetzen und wie sicher sind diese? (s. dazu z.B. [166])

Die vorliegende Arbeit verfolgt dabei nun zwei wesentliche Ziele:

1. Nachweis der funktionalen Korrektheit einer Protokollimplementierung in Java bzgl. einer abstrakten Spezifikation
2. Vererbung der Gültigkeit von Sicherheitseigenschaften auf die Implementierungsebene

Das erste Ziel bedeutet intuitiv, dass eine Implementierung das gleiche Ein-/Ausgabe Verhalten zeigt wie die abstrakte Spezifikation. Zusätzlich müssen die internen Zustandübergänge ebenfalls im Wesentlichen gleich sein. Das zweite Ziel geht einen Schritt weiter: Sicherheitseigenschaften sollen bereits auf abstrakter Ebene als korrekt nachgewiesen werden und sich dann automatisch (bzw. mit geringem Verifikationsaufwand) auf die Implementierung übertragen lassen. Erst zweiteres Ziel garantiert formal die Sicherheit einer Implementierung.

4.2 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten anderer Arbeitsgruppen vorgestellt, die zu den hier verwendeten Ansätzen verwandt sind. Dies sind zum Einen die verschiedenen Verifikationssysteme für objektorientierte Programmiersprachen im Allgemeinen. Diese werden in Kap. 4.2.1 beschrieben. Im Anschluss geht Kap. 4.2.2 auf Methoden zur Überprüfung von Protokollimplementierungen ein.

Da technischere Vergleiche bzw. fallstudienspezifische Vergleiche der vorherigen genaueren Vorstellung eigener Arbeiten bedürfen, finden sich weitere Vergleiche zu internationalen Vorarbeiten auch später in Kap. 12.3.1 und Kap. 12.8.

Da die Verfeinerungstheorien in der Literatur eher auf theoretischer Ebene betrachtet werden und eine Anwendung auf tatsächlich lauffähigen Quellcode, wie sie mit dieser Arbeit vorgestellt wird, bisher in der Literatur nicht vorhanden ist, wird auf einen theoretischen Vergleich verschiedener Verfeinerungstheorien an dieser Stelle verzichtet. Die verwendeten Theorien werden in Kap. 3.4 vorgestellt. Ein detaillierter Vergleich von Theorien für Data Refinement mit ASM Refinement findet sich insbesondere in [158].

Die vorliegende Arbeit verfolgt zwar das Ziel, Sicherheitseigenschaften für ein Kommunikationsprotokoll durch Verfeinerung auch für eine Implementierung zu zeigen, der eigentliche Beweis der Sicherheit des Protokolls ist dagegen nicht Gegenstand der Arbeit. Die Dissertation von Haneberg [71] vergleicht den hier zugrunde liegenden Ansatz im Detail mit verwandten Ansätzen zur Protokollverifikation wie etwa den Arbeiten von Paulson [139], Basin [41], Burrows, Abadi und Needham [33], Meadows [122], Roscoe [152] oder Jürjens [101]. Daher wird hier nicht auf Ansätze zur Verifikation der Sicherheit von Protokollen auf abstrakter Ebene eingegangen.

Eigene Publikationen des Autors zu den einzelnen Kapiteln dieser Arbeit werden jeweils am Anfang der späteren Kapitel in den initialen Zusammenfassungen genannt.

4.2.1 Objektorientierte Verifikation

Ein ausführlicher Vergleich des Java Kalküls in KIV zu anderen objektorientierten Verifikationsansätzen findet sich bereits in der Dissertation von Stenzel [173]. Im Folgenden wollen wir einige Ansätze in Bezug auf die Verifikation von Fallstudien vorstellen und - soweit möglich - mit den hier verwendeten Techniken vergleichen.

Ein gegenwärtig sehr beliebter Trend ist der Einsatz von annotationsbasierten Werkzeugen und Methodiken. Bei diesen Ansätzen besteht die Grundidee darin, die Programme mit Annotationen z.B. für Methodenkontrakte und Klasseninvarianten zu erweitern. Die Annotationen

ergeben eine Spezifikation für den Quellcode. Damit folgen diese Ansätze dem Prinzip des Design by Contract [124].

Wichtigster Vertreter dieser Annotationssprachen für Java ist die Java Modelling Language von Leavens (JML, [113] [114]). JML unterstützt insbesondere Methodenspezifikationen mittels Kontrakten. Dabei sind etwa Vorbedingungen und Nachbedingungen für Methoden sowie Angaben zum Verhalten im Falle von Ausnahmen möglich. Innerhalb der Annotationen wird im Wesentlichen die Java Syntax verwendet. Zusätzlich sind noch weitere Konstrukte wie z.B. Allquantifikation möglich. Neben Methodenkontrakten können auch Klasseninvarianten angegeben werden. JML selbst bringt dabei bereits eine Tool Suite mit, die z.B. einen Typchecker für die Annotationen und einen Compiler enthält, der die Methodenkontrakte in *assertions* übersetzt. Es sind allerdings noch keine Tools zur statischen Überprüfung bzw. Verifikation der Annotationen enthalten.

Die JML Sprache wurde von diversen Forschergruppen aufgegriffen und für Fallstudien und in diversen Tools verwendet. Das Tool ESC/Java2 (Extended Static Checker for Java [40]) ist ein auf JML basierender statischer Programm Checker. Das Tool übersetzt das Java Programm samt JML Annotationen zunächst in eine einfachere Sprache, ähnlich zu Dijkstras Guarded Commands. Daraus werden dann Beweisverpflichtungen für den automatischen Theorembeweiser Simplify [47] generiert. Das Tool ist durch die komplett automatische Analyse weder vollständig noch korrekt - eine Auflistung von Quellen für beide Fälle gibt z.B. [103]. Das Ziel ist vielmehr eine Unterstützung bei der Implementierung und insbesondere eine effiziente Möglichkeit, einen Entwickler auf mögliche Fehler hinzuweisen. Eine der ersten größeren Fallstudien mit dem Tool war die Implementierung einer E-Voting Lösung in Holland [102]. Ebenso wurde ESC/Java (die Vorgängerversion zu ESC/Java2 von Compaq Research [55]) für den Beweis eines JavaCard Applets für eine elektronische Geldbörse verwendet [38]. Ein Vergleich findet sich in Kap. 4.2.2.

Ähnlich zu ESC/Java2 basiert auch das LOOP Tool [183] auf JML. LOOP hat seine Ursprünge in der Formalisierung einer denotationalen Semantik für Java [97] [85] im interaktiven Beweissystem PVS [138] [167]. Hierzu wurde mit dem LOOP Tool ein Übersetzer entwickelt, der Java Klassen in PVS Theorien übersetzen kann. Auch für JML wurde eine solche Übersetzung implementiert, was schließlich die Generierung von Beweisverpflichtungen für PVS erlaubt. Diese sind Hoare Tripeln ähnlich und können mittels einer Java Hoare Logik [94] oder einem weakest-precondition Kalkül [95] bewiesen werden. Mittels dem LOOP Tool und PVS wurden u.a. eine Implementierung für Dezimalzahlen [28] auf Chipkarten und eine Implementierung für eine einfache Bezahlkarte [93] verifiziert. Auf Letztere wird im nächsten Abschnitt nochmals vergleichend zur vorliegenden Arbeit eingegangen. Erstgenannte Dezimal-Implementierung verfolgt genau die beschriebene Vorgehensweise: JML Spezifikation, Übersetzung und Beweis mittels PVS. Das Vorgehen ist dem in KIV sehr ähnlich. Als weitere Fallstudie ist die Verifikation einer Java `Collection` Klasse durch Huisman in [86] zu nennen. Hier wird statt PVS das Isabelle Beweissystem [133] verwendet. Die Beweise selbst sind methodisch (modulo verwendeter Logik usw.) zu dem Vorgehen in KIV recht ähnlich.

Als weiterer Vertreter von JML-basierten Tools ist JACK (Java Applet Correctness Kit [32]) zu nennen. Das anfangs vom Chipkartenhersteller Gemplus entwickelte Werkzeug wird nun von der französischen Forschungseinrichtung INRIA weiter gepflegt. Im Gegensatz zu LOOP liegt hier der Fokus auf vollautomatischer Analyse mittels einem weakest-precondition Kalkül. Analog zu LOOP wird eine Übersetzung der JML annotierten Programme in die Eingabespra-

chen verschiedener Beweissysteme durchgeführt. Das Ziel ist (ähnlich zu ESC/Java2) die Unterstützung normaler Softwareentwickler ohne Kenntnisse in formaler Verifikation. Dies zeigt sich auch durch die Integration in Eclipse. JACK basiert dabei nicht (wie LOOP) auf einer formalen Java Semantik. Dennoch wird neben vollautomatischer Verifikation der entstehenden Beweisverpflichtungen (ebenso mittels Simplify oder dem Beweiser des Atelier B Toolkits [2]) auch bei JACK eine Übersetzung in PVS angeboten.

Das KRAKATOA Werkzeug [54] ist ebenfalls ein JML basiertes Werkzeug. Es bietet selbst keinen Verifikationssupport, sondern wird als reiner Übersetzer verwendet. JML annotierte Java Quelldateien werden mittels Krakatoa in die Eingabesprache des Why Tools [53] [54] übersetzt, welches daraus Beweisverpflichtungen für das Verifikationssystem Coq [23] generiert. In [121] zeigen Marche et al. exemplarisch die Verifikation von Dijkstras Dutch National Flag Algorithmus mittels Krakatoa. Das Tool bietet im Gegensatz zu KIV keine Unterstützung für das Überschreiben von Methoden in Java Programmen, den Überlauf von Integers oder für den Beweis der Terminierung von rekursiven Programmen.

Als Grenzfall zwischen annotationsbasierten und nicht annotationsbasierten Ansätzen ist das KeY Tool [19] [6] anzusehen. KeY ist dabei wohl dem KIV System in Bezug auf zu Grunde liegende Logik und Methodik am ähnlichsten. Ebenso wie KIV basiert KeY auf einer dynamischen Logik zur Programmverifikation. Dementsprechend finden sich Operatoren wie Box und Diamond und auch die Sequenzen und Sequenzenkalkülregeln im KeY System wieder. Ein Hauptunterschied auf Kalkülebene ist die Handhabung von Exceptions: In KeY bedeutet das Auftreten einer Exception Nicht-Terminierung. Anders als KIV ist KeY ein hauptsächlich auf Java ausgerichtetes Werkzeug (daneben existieren auch Arbeiten zur Verifikation eines C Fragments mittels KeY [131]). Insbesondere fehlt der verwendeten Logik die Möglichkeit zur Definition von abstrakten Datentypen neben den in Java existenten Typen. Weitere Logiken (wie in KIV z.B. für temporale Logik [8] oder ASMs [156] realisiert) sind nicht vorgesehen. Dies zeigt sich insbesondere darin, dass die Typen der Java Klassendeklarationen direkt als Typen der Logik von KeY verwendet werden. Eine explizite Speicherrepräsentation wie in KIV ist nicht vorgesehen, statt dessen sind die Java Typen und z.B. die Felder von Klassen direkt in die Logik integriert. Dies schafft zum Einen eine etwas intuitivere Repräsentation von Speicherinhalten, auf der anderen Seite ist eine Abstraktion - und insbesondere formale Verfeinerung wie in dieser Arbeit beschrieben - mit dem KeY System derzeit nicht möglich. Im KeY System können Beweisverpflichtungen ähnlich zu KIV direkt mittels dynamischer Logik und Sequenzen formuliert werden. Daneben bietet KeY jedoch auch die Möglichkeit zur Übersetzung von OCL Constraints [134] oder JML Annotationen an. Aus beiden können automatisch Beweisverpflichtungen in dynamischer Logik generiert werden. Derzeit wurden zwei große Fallstudien mit dem KeY System verifiziert: Mostowski zeigt in [130] eine Verifikation der JavaCard API. Es handelt sich hierbei um die erste vollständige Verifikation dieser Laufzeitumgebung. Sämtliche Methodenkontrakte wurden dabei mit der dynamischen Logik des KeY Systems spezifiziert. Es fand noch keine Anwendung von JML oder OCL statt. Eine JML Spezifikation der API von Meijer und Poll findet sich vergleichend in [123]. Demgegenüber spezifizieren und verifizieren Tonin und Schmitt in [165] die Mondex Fallstudie mittels KeY. Hier werden alle Beweisverpflichtungen mittels JML angegeben. Ein ausführlicher Vergleich dieses Ansatzes zu den hier vorgestellten Arbeiten findet sich am Ende dieser Arbeit in Kap. 12.8. Zusammenfassend lässt sich sagen, dass das KeY System im Vergleich zu KIV spezialisierter auf Java ausgelegt ist und einen stärkeren Fokus auf automatischere Beweise hat. Demgegenüber wurde mittels KIV eine formale Semantik und ein dazu korrekter Kalkül

für Java umgesetzt und das KIV System besitzt insbesondere die Möglichkeit zur weiteren Abstraktion, was Arbeiten wie die vorliegende überhaupt erst möglich macht.

Nicht für Java, sondern für C#, wurde von Microsoft Research das Annotations- und Verifikationstoolkit rund um die Sprache Spec# [126] entwickelt. Dabei handelt es sich um eine Erweiterung von C# um Annotationen analog zu JML. Die dahinter liegende Toolsuite funktioniert ähnlich zur Vorgehensweise von ESC/Java2 oder JACK: Es werden mittels Übersetzung in die Zwischensprache BoogiePL und einem weakest precondition Kalkül Beweisverpflichtungen generiert, die anschließend von einem automatischen Beweissystem verifiziert werden. In früheren Versionen beruhte auch Spec# auf dem Simplify Beweissystem, inzwischen wurde mit Z3 [43] ein eigener SMT (Satisfiability Modulo Theories) Solver entwickelt, der z.B. lineare Arithmetik unterstützt. Die Spec# Sprache wurde daneben auch in anderen Tools von Microsoft Research verwendet, so z.B. in SpecExplorer als Grundlage für das modellbasierte Testen [36]. SpecExplorer wurde auch zum Testen von kryptographischen Protokollen eingesetzt, insbesondere zeigt [153], dass so z.B. auch die bekannte Schwachstelle im originalen Needham-Schroeder Protokoll gefunden werden konnte.

Auch im Isabelle Beweissystem wurde insbesondere durch die Arbeiten durch von Oheimb [185] [184] [136] im Bali Projekt eine Semantik sowie eine Hoare Logik für eine Teilmenge der Java Sprache definiert. Im Nachfolgeprojekt VerifiCard wurde insbesondere auch Byte Code Verifikation betrachtet [105] [107] [106] sowie die Analyse spezifischer Java Sprachkonstrukte wie Definite Assignment [163] und den Package Konzepten [164]. Ebenso auf Isabelle bzw. PVS als Beweissystem beruhend wurde mit dem Jive System [125] eine weitere Hoare Logik für (eine Teilmenge von) JavaCard definiert.

Zusammenfassend lässt sich zu den bisher genannten Ansätzen vergleichend mit der vorliegenden Arbeit festhalten: Keiner der genannten Ansätze wurde verwendet, um eine auf Verfeinerung basierende Verifikation durchzuführen: Keiner der Ansätze verfolgt das Ziel, eine abstraktere Sichtweise auf z.B. Kommunikationsprotokoll zu erlauben und Korrektheit bezüglich einer solchen Spezifikation zu zeigen. Insbesondere bei den meisten annotationsbasierten Ansätzen ist dies technisch gar nicht möglich. Die Annotationen in JML oder Spec# geben eine Spezifikation auf der niedrigen Abstraktionsebene des vorliegenden Quellcodes vor. Insbesondere eine Verifikation von High-Level Eigenschaften (wie z.B. korrekte Authentisierung bei Kommunikationsprotokollen, bei der mehrere Protokollteilnehmer, deren Kommunikationsverhalten und ein eventueller Angreifer betrachtet werden müssen) ist so nicht gut möglich. Die Spezifikationen in JML erlauben das Betrachten der Umgebung eines Programms (wie z.B. andere Programme oder Nachrichtenkommunikation) nicht. Dies zeigt im Details auch der Vergleich der vorliegenden Arbeit mit Arbeiten mittels dem KeY System in Kap. 12.8.

Besondere Beachtung verdient aktuell das Verisoft Projekt [144] (und sein aktuelles Nachfolgeprojekt VerisoftXT). Hier ist der Anspruch, eine vollständige Verifikationskette beginnend bei verifizierter Hardware hinweg über einen verifizierten Compiler, verifiziertem Betriebssystem, verifizierten virtuellen Maschinen sowie Kommunikationsprotokollen hin zu verifiziertem Quellcode zu schaffen. Sämtliche Programme werden in der eigenen Sprache C0 (einer eingeschränkten Variante von C) implementiert. Als Fallstudien werden verschiedene Anwendungen herangezogen: Biometrische Identifikation mit Chipkarten, eine Notrufanwendung aus dem Automotive Bereich, ein Mikrocontroller, sowie ein System zum sicheren Versenden von EMails. Insbesondere letzteres soll den Gesamtansatz illustrieren. Zur Verifikation kommt VSE [87] und ebenso Isabelle [133] zu Einsatz. Das Projekt hat in den einzelnen Bereichen bereits ei-

ne große Anzahl von verschiedensten Ergebnissen geliefert. Weiter verwandt im Umfeld der vorliegenden Arbeit sind z.B. die Arbeiten von Beckert et al. [18], [20], [21] [22]. Dennoch handelt es sich bei den bisherigen Ergebnissen jeweils um einzelne Teilaspekte des beschriebenen Ganzen, die teils mit unterschiedlichen Methodiken gelöst wurden. Ein vollständig integrativer Ansatz wurde noch nicht entwickelt. Insbesondere die Quellcode Verifikation des beschriebenen EMail-Systems steht noch aus.

Im folgenden Abschnitt werden nun weitere Ansätze, die stärker auf die Behandlung von Protokollen und Implementierungen dieser abzielen und damit zur vorliegenden Arbeit direkter verwandt sind, detaillierter beschrieben und verglichen.

4.2.2 Formale Behandlung von Protokollimplementierungen und Applets

Neben den genannten Methodiken und Tools zur Verifikation objektorientierter Programme sollen in diesem Abschnitt enger mit der vorliegenden Arbeit verwandte Arbeiten aus dem Bereich der Java Applet Verifikation sowie der formalen Behandlung von Protokollimplementierung betrachtet werden.

Mittels ESC/Java wurden bereits mehrere Fallstudien aus den genannten Bereichen verifiziert. In der bereits oben erwähnten Arbeit [38] beschreiben Cataño und Huisman eine Spezifikation einer elektronischen Geldbörse mittels JML und ESC/Java. Die Arbeit verfolgt dabei nicht den Anspruch, eine korrekte Implementierung bezüglich einer Spezifikation zu erstellen (dies ist auf Grund der oben bereits beschriebenen fehlenden Vollständigkeit und Korrektheit von ESC/Java nicht möglich), sondern zu erproben, ob mittels dem Werkzeug Programmierfehler gefunden werden können. Die entstandenen Methoden Kontrakte können trotzdem als Grundlage für die Verifikation mittels anderer Werkzeuge verwendet werden. Es gelingt den Autoren, tatsächlich subtile Programmierfehler im Quellcode zu finden. Kryptographie, die korrekte Implementierung von Kommunikationsprotokollen oder Sicherheitseigenschaften wurden von der Arbeit dagegen überhaupt nicht betrachtet.

Auch mit dem LOOP Tool wurde eine elektronische Geldbörse von Jacobs, Marché und Rauch in [93] verifiziert. Das Kommunikationsprotokoll an sich, das in dieser Geldbörse verwendet wird, genügt nicht den Sicherheitsanforderungen der Realität, da z.B. das Abbuchen von Geld nicht kryptographisch gesichert ist. Das Aufladen von Geld ist mittels einem einfachen Challenge-Response Verfahren gesichert. Die Protokollsicherheit war allerdings auch nicht Intention der Autoren, vielmehr sollte die Verifikationsmethodik mittels LOOP an einem Applet erprobt werden und die funktionale Korrektheit des Codes gezeigt werden: eine typische verifizierte Eigenschaft ist etwa, dass beim Aufladen von Geld auf die Karte deren Kontostand tatsächlich auf den in der Nachricht enthaltenen Wert gesetzt wurde. Die Autoren geben zwar an, dass sie ihr Protokoll zusätzlich mittels Casper/FDR [117] überprüft haben - die Korrektheit ihres Quellcodes bezüglich des dort verwendeten Modells wird allerdings nicht überprüft. Statt dessen spezifizieren sie das intendierte Protokollverhalten als Automat. Die Zustände korrespondieren zu einem Modus-Feld des Applets. Die Zustandsübergänge entsprechen den empfangenen Nachrichten. So können dann schließlich die intendierten Zustandsübergänge (nach Empfang von Nachricht X in Zustand Z muss danach Zustand Y gelten) als JML Annotationen an die `process` Methode annotiert werden und so die Korrektheit des Codes bzgl. dieser Spezifikation gezeigt werden. Im Vergleich zur vorliegenden Arbeit kann so natürlich keine Sicherheitsaussage für den Quellcode getroffen werden. Die gleichen Autoren geben in

[96] nochmals einen Überblick über die Verifikation der selben Fallstudie mittels ESC/Java, Jive, LOOP und Krakatoa.

Einen ähnlichen Ansatz zur Verifikation von Protokollimplementierungen verfolgen Hubbers, Oostdijk und Poll in [84]. Ziel ist hier die Verifikation eines Protokolls zum bilateralen Schlüsselaustausch in JavaCard. Der Anspruch des Papiers ist dabei durchaus mit dem Anspruch dieser Arbeit vergleichbar: Eine Protokollspezifikation auf abstrakter Ebene soll durch mehrere Verfeinerungen zu einer Implementierung werden und schließlich sollen auf der Implementierungsebene die Korrektheits- und Sicherheitseigenschaften der abstrakten Ebene ebenso gelten. Auf abstrakter Ebene wurde die Sicherheit des Protokolls mittels Casper/FDR [117] überprüft. Danach sollte (in obigem Papier als zukünftige Arbeiten angekündigt) z.B. mittels LOOP tatsächlich eine Verifikation durchgeführt werden. Dies wurde allerdings nicht mehr realisiert. Statt dessen zeigen die Autoren, wie das Protokollverhalten (ähnlich zur beschriebenen Arbeit [93]) mittels eines Automaten spezifiziert werden kann. Anschließend werden auch hier JML Annotationen verwendet, um Kontrakte anzugeben, die das Zustandsübergangsverhalten dieses Automaten angeben. Allerdings werden hier keine Nachrichten betrachtet, sondern lediglich der interne Zustand. Damit ist natürlich keine echte Sicherheitsaussage für den Code mehr möglich und nur eine eingeschränkte Aussage für die Korrektheit. In einer neueren Arbeit wird von Schubert und Poll eine ähnliche Methodik für die Verifikation einer OpenSource SSH Implementierung verwendet [142]. Hier wird zusätzlich mittels das Tool AUTOJML verwendet, das Annotationen aus dem Quellcode automatisch generiert und somit ein bereits ein Grundgerüst für die weitere Analyse mittels ESC/Java2 schafft. Ansonsten wurde der gleiche automatenbasierte Ansatz verwendet, Sicherheit für die Implementierung wurde nichtgezeigt. In einem zweiten Schritt wurde zusätzlich noch das Nicht-Auftreten von unerwünschten Runtime Exceptions (Dereferenzierung von `null` u.ä.) gezeigt.

Einen anderen Ansatz verfolgen Affeldt und Kobayashi in [5] mittels dem Coq Verifikationswerkzeug. Das Ziel der Arbeit ist die Formalisierung und Verifikation eines SMTP Mail Servers. Ein Teil dieses in Java implementierten Servers wurde dabei per Hand in die Eingabesprache von Coq übersetzt und es wurde verifiziert, dass sich diese Spezifikation gemäß dem SMTP Protokoll verhält. Der Umfang des so betrachteten Quellcodes ist 700 Zeilen und die Verifikation dauerte mehrere Monate. Damit ist das Projekt mit der Mondex Verifikation in dieser Arbeit vom Umfang her vergleichbar. Sicherheitseigenschaften wurden nicht gezeigt, lediglich Korrektheitseigenschaften. Es bleibt allerdings festzuhalten, dass die händische Transformation des Quellcodes in Coq sehr viel Raum für Fehler lässt und damit eine solche Verifikation nur eine eingeschränkte Aussage für das reale System liefern kann.

Song, Perrig und Phan beschreiben in [169] das Toolkit AGVI: Automatic Generation, Verifikation and Implementation of Security Protocols. Grundidee ist hierbei, lediglich die gewünschten Sicherheitsziele vorzugeben und daraus automatisch ein passendes Protokoll zu generieren. Zusätzlich wird das so entstandene Protokoll mit dem Athena Modelchecker [168] automatisch geprüft und darüber hinaus auch eine Java Implementierung automatisch erzeugt. Der Ansatz konnte, wie in [141] und [140] gezeigt, durchaus elegante und kurze Protokolle finden. Die Codegenerierung ist allerdings nicht formal bewiesen. Damit lässt sich keine verlässliche Korrektheitsaussage für den Quellcode aus dem Ansatz herleiten.

Einen ähnlichen Ansatz verfolgen Tobler und Hutchison in [180] mittels dem Tool spi2java. Zunächst wird hier eine Spezifikation eines Security Protokolls im Spi Kalkül [1] erzeugt, einer auf dem π Kalkül [127] basierenden Prozessalgebra speziell für kryptographische Protokolle.

Auf dieser Ebene können z.B. Sicherheitseigenschaften bewiesen werden. Daraus wird dann in einem Codeerzeugungsschritt eine Java Implementierung erzeugt. Die Codeerzeugung folgt dabei den gleichen Prinzipien wie die vorliegende Arbeit: Protokolldatentypen wie z.B. Nonces oder wie die einzelnen Nachrichten werden auf generische Java Klassentypen abgebildet, die funktionale Spezifikation des Protokolls wird zu Java Code mit Operationen auf diesen Klassen. Dabei bleibt allerdings die Korrektheit der Übersetzung unbewiesen. Auch hier lässt sich damit keine Sicherheits- oder Korrektheitsaussage für den Quellcode ableiten. Eine formale Verfeinerung wird nicht bewiesen. Die automatisch erzeugten Implementierungen sind im Vergleich zu händischen Implementierungen unverhältnismäßig lang, enthalten redundante Teile und sind damit schwer zu verstehen und zu erweitern.

Im französische EVA Project wurde von Marlet und Le Metayer ebenso eine Spezifikationssprache für Security Protokolle sowie eine Verifikationsmethodik entwickelt [112]. Die vorgestellte Spezifikationssprache DEXTRA erlaubt sehr implementierungsnahe Spezifikationen von Smart Card Protokollen. So sind z.B. die Längenangaben von Nachrichtenteilen als Byte Folgen explizit in der Spezifikation enthalten. Daraus kann dann sowohl eine abstraktere Spezifikation zum Nachweis von Sicherheitseigenschaften erzeugt werden als auch mittels einer eigenen Implementierung eine JavaCard Implementierung statisch auf korrekte Implementierung des Protokolls überprüft werden. Eine formale Untermauerung dieser Überprüfungen wurde nicht durchgeführt. Ebenso beschränkt sich die Analyse des Quellcodes natürlich nur auf die einfachen Sprachkonstrukte, die auch direkt mit der Modellierungssprache umsetzbar sind. Eine weitere Entwicklung oder Erprobung des Ansatzes an Fallstudien fand nicht mehr statt.

Kapitel 5

Eine Theorie zur Verfeinerung von Protokollen zu Code

All things are difficult before they are easy.
Thomas Fuller

Dieses Kapitel stellt das Grundgerüst für die Spezifikation der Ebenen der Verfeinerung vor. Insbesondere wird ein generischer Spezifikationsrahmen für die konkrete Ebene zu einer abstrakten Prosecco ASM gegeben. Dazu werden die verschiedenen Bestandteile der Spezifikation erläutert und die resultierenden Beweisverpflichtungen erklärt. Zunächst beginnen wir allerdings mit der Vorstellung eines einfachen Beispiels. Die in diesem Kapitel beschriebenen Arbeiten wurden in [63] und [66] publiziert.

5.1 Beispiel: Die Cindy Spezifikation

Als Beispiel für die Vorstellung des allgemeinen Verfeinerungsrahmens dieser Arbeit dient zunächst die Cindy Anwendung. Für die folgenden Kapitel ist daher die Prosecco Spezifikation der Applikation wichtig. Daher werden im Folgenden deren wichtigste Bestandteile und das Protokoll kurz erläutert.

Das Cindy Protokoll ist dabei sehr einfach und beinhaltet keine explizite Kryptographie. Sämtliche Sicherheitseigenschaften beruhen auf der Nicht-Abhörbarkeit und Manipulationssicherheit der verwendeten GSM Mobilfunkkommunikation. Da dort eine implizite Verschlüsselung und Authentifizierung gegeben ist, muss keine eigene Kryptographie in der Anwendung verwendet werden. Lediglich der eindeutige Identifier für die Tickets, der als Data Matrix Code auf dem Display dargestellt wird, darf nicht von einem Angreifer ratbar sein.

Meist handelt es sich bei den Protokollen nicht um ein mehrschrittiges Austauschen von Nachrichten sondern nur um einfache Nachrichten, die in einem Schritt zwischen den modellierten Agenten verschickt werden. Für die Modellierung der Nachrichten werden die Dokumente aus Kap. 3.5 verwendet. Am wichtigsten ist hierbei die Kommunikation zwischen Kino-Server und

Mobiltelefon. Diese findet mittels MMS statt. Eine MMS wird als Document modelliert als Liste Doclist(IntDoc(phonenumber) + doc), die neben der Telefonnummer des Absenders (phonenumber) den eigentlichen Nachrichteninhalt (doc) beinhaltet. Die Nachrichteninhalte werden wiederum mittels einer doc = Doclist(IntDoc(ins) + doc') modelliert, die eine Instruktion ins und einen Inhalt doc' hat. Diese Dokumentstruktur wird auch CommandDoc genannt, da anhand der Instruktion die Nachrichten den Protokollschritten zugeordnet werden. Eigentlicher Inhalt eines solchen CommandDocs kann nun z.B. ein Ticket sein. Ein Ticket ist wiederum eine Doclist(IntDoc(ticketdata) + NonceDoc(nonce)), das neben den Ticketdaten (modelliert als IntDoc) den eindeutigen DataMatrix Code als NonceDoc enthält. Die Nachrichten, die über das Netzwerk verschickt werden, haben daher im Modell der Cindy Anwendung die Form, die in Abb. 5.1 dargestellt ist.

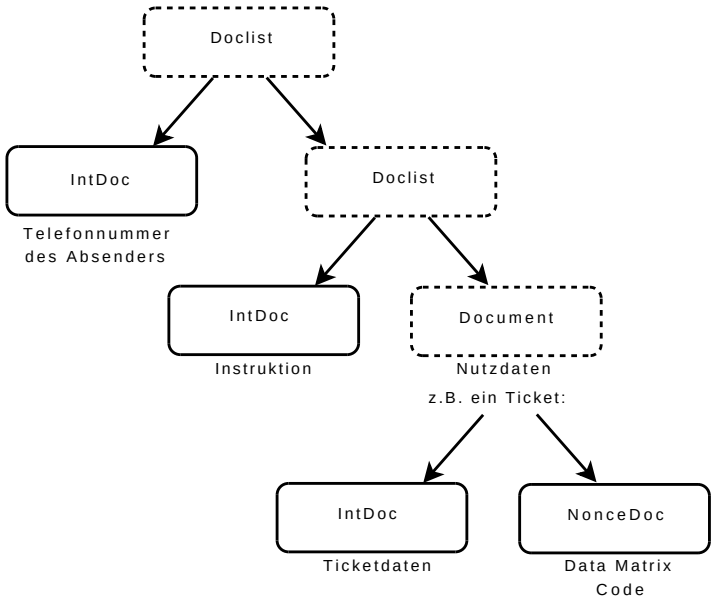


Abbildung 5.1: Nachrichtenstruktur der Cindy Anwendung

Einzige Abweichung von dieser Struktur ist das Vorzeigen des Tickets beim Eintritt ins Kino. In der Realität geschieht dies durch Anzeige des Data Matrix Codes auf dem Display des Telefons. Im Modell wird einfach die gleiche Sende- und Empfangsinfrastruktur verwendet wie auch für MMS Nachrichten. Das gleiche gilt für Eingaben des Benutzers über die graphische Oberfläche der Anwendung. Formal kann man damit das Einschalten der Anwendung auf dem Mobiltelefon als Verbindungsaufbau zwischen dem Benutzer-Agent und dem Telefon-Agent verstehen (und analog das Ausschalten als Verbindungsabbau). Technisch ist dies später als Implementierung mittels dem Command Pattern gelöst, wobei die graphische Oberfläche der Protokollimplementierung Dokumente als Commands übergibt.

Im Weiteren ist die generische (von Smartcards inspirierte) Darstellung der Kommunikationsinfrastruktur mittels Verbindungen bei einer Handy-Anwendung nicht immer adäquat. Insbesondere das Verschicken von MMS Nachrichten ist in der Realität verbindungslos. Daher wird das formale Rahmenmodell an dieser Stelle um die Möglichkeit zur verbindungslosen Kommunikation erweitert. Formal wird dies gelöst, in dem das SEND Makro aus Prosecco

um zwei zusätzliche Parameter, den Empfänger receiver (hier als IntDoc, das eine Telefonnummer darstellt) und den Sende-Modus send-mode (direct oder verbindungsorientiert normal), erweitert.

Cindy stellt darauf aufbauend folgende Protokolle zur Verfügung:

- *Kaufen über das Mobiltelefon* s. Abb. 5.2

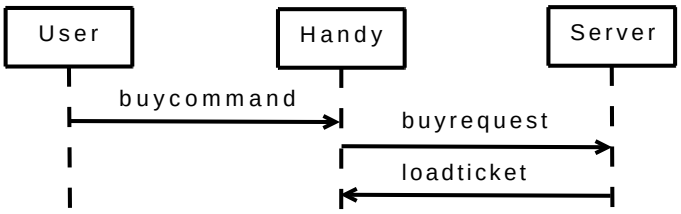


Abbildung 5.2: Kaufprotokoll mit dem Mobiltelefon

- *Kaufen über das Internet mit einem PC* s. Abb. 5.3

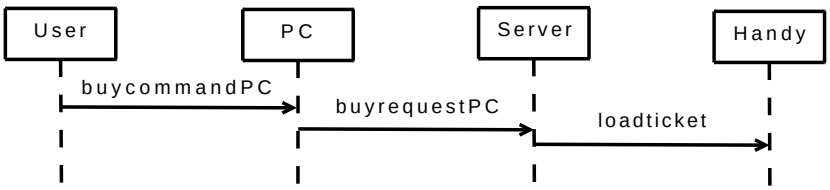


Abbildung 5.3: Kaufprotokoll mit dem Internet PC

- *Übertragung eines Tickets* s. Abb. 5.4

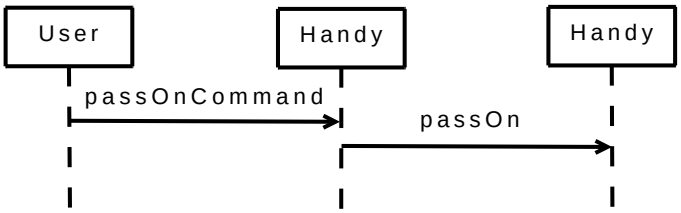


Abbildung 5.4: Übertragung eines Tickets

- *Vorzeigen eines Tickets* s. Abb. 5.5

Das Format der Nachrichten wird durch Tab. 5.1 angegeben.

Für die folgenden Kapitel wird die Cindy Anwendung immer wieder als Beispiel dienen. Während die Spezifikationsteile für den Angreifer, den Kino-Server und den Internet-PC für die

Tabelle 5.1: Nachrichten der Cindy Anwendung

buycommand	Doclist(IntDoc(user) + Doclist(IntDoc(buy) + Doclist(IntDoc(phonenumber) + IntDoc(ticketdata))))
buyrequest	Doclist(IntDoc(phonenumber) + Doclist(IntDoc(buyTicket) + Doclist(IntDoc(phonenumber) + IntDoc(ticketdata))))
loadticket	Doclist(IntDoc(server) + Doclist(IntDoc(loadTicket) + Doclist(IntDoc(ticketdata) + NonceDoc(datamatrixcode))))
buycommandPC	Doclist(IntDoc(PC) + Doclist(IntDoc(buyInternet) + Doclist(IntDoc(phonenumber) + SecretDoc(password) + IntDoc(ticketdata))))
buyrequestPC	Doclist(IntDoc(phonenumber) + Doclist(IntDoc(buyTicket) + Doclist(IntDoc(phonenumber) + SecretDoc(password) + IntDoc(ticketdata))))
passOnCommand	Doclist(IntDoc(user) + Doclist(IntDoc(passOnCmd) + Doclist(IntDoc(ticketindex) + IntDoc(receivernumber))))
passOn	Doclist(IntDoc(phonenumber) + Doclist(IntDoc(loadTicket) + Doclist(IntDoc(ticketdata) + NonceDoc(datamatrixcode))))
showCommand	Doclist(IntDoc(user) + Doclist(IntDoc(present) + IntDoc(ticketindex))
present	Doclist(IntDoc(phonenumber) + NonceDoc(datamatrixcode))

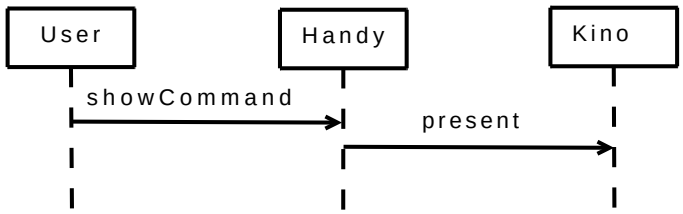


Abbildung 5.5: Vorzeigen eines Tickets

folgenden Erläuterungen unwesentlich sind (diese Agenten bleiben beim Refinement unverändert), ist die Spezifikation des Mobiltelefons wichtig. Diese wird daher (inklusive der schematischen Einbettung in die Haupt-ASM-Regel) im Folgenden im PROSECCO Stil schematisch angegeben[1].

```
CINDY(... ; attacker-known, inputs, connections, ..., tickets, stop) {
   while (¬ (stop ∧ no-tickets-left(...) )) { CINDY-STEP; stop := [?]}
}

CINDY-STEP(... ; attacker-known, inputs, connections, ..., tickets, stop) {
   choose cindy-step with
              (step = cellphone-step → ∃ agent. cellphone?(agent) ∧ ready?(agent, ...))
           ∧ (step = cinema-step →...)
           ∧ (step = attacker-step →...)
           ∧ (step = ... →...)
           ∧ (step = connect →...)
           ∧ (step = disconnect →...)
   in
   {
   if (cindy-step = connect) then CONNECT(...; connections, ...)
   if (cindy-step = ...) then ...
   if (cindy-step = attacker-step) then ATTACKER(attacker-known; inputs, ...)
   if (cindy-step = cellphone-step) then choose agent
         with (cellphone?(agent) ∧ ready?(agent, ...)) in
            CELLPHONE(agent, ...; attacker-known, inputs, tickets, ...)
   if (cindy-step = cinema-step) then choose agent with ... in CINEMA(agent; ...)
   }
}
...
CELLPHONE(agent, ... ; attacker-known, inputs, tickets, ...) {
  let outdoc = ⊥ in
  choose inport with inputs(agent)(inport) ≠ [ ] in
```

[1]Der besseren Lesbarkeit wegen werden hier manche Tests durch Prädikate mit intuitiven Namen abgekürzt. Des Weiteren werden nicht alle Zustandsupdates und Protokollschritte dargestellt, sondern nur jene, die für ein Verständnis der nächsten Kapitel wichtig sind. Eine genauere Betrachtung findet sich später in Kap. 7

```
  let inmsg = inputs(agent)(inport).first in    {
    inputs(agent)(inport) := inputs(agent)(inport).rest;
    let indoc = get-part(inmsg, 2) in
    if is-passon(indoc) then
       ...
    else if is-load(indoc) ∧ # tickets(agent) < MAXTICKETS then
       tickets(agent) :=    tickets(agent)
                          + doclist(get-part(inmsg, 1) + get-part(indoc,2))
    else if is-buy(indoc) then {
       booked(agent) := booked(agent) + get-part(indoc, 2);
       send-mode := direct;
       receiver := CINEMA;
       outport := MMSPORT;
       outdoc := doclist(intdoc(buyTicket) + get-part(indoc, 2)); }
    else if is-present(indoc) then
       ... }
  if outdoc ≠ ⊥ then SEND
}
```

Die Hauptregel CINDY führt bis zum Erreichen der Abbruchbedingung die ASM Regel für den Protokollschritt CINDY-STEP durch. Die Abbruchbedingung stop ist hier erweitert um die zusätzliche Bedingung no-tickets-left. Diese Bedingung besagt, dass keine Tickets mehr auf den Telefonen der Benutzer vorhanden sind, die noch nicht beim Kino vorgezeigt wurden (intuitiv bedeutet dies, dass wir nur Endzustände betrachten, in denen jeder Benutzer seine Tickets auch eingelöst hat - etwa am Ende eines Geschäftstages). Diese zusätzliche Einschränkung macht die Betrachtung von Sicherheitseigenschaften etwas einfacher, s. dazu auch [63].

Die CINDY-STEP Regel wählt nun einen gültigen Protokollteilnehmer (bzw. die generischen Makros CONNECT und DISCONNECT für Verbindungsaufbau und -abbau zwischen zwei Teilnehmern) indeterministisch aus und führt im Anschluss das entsprechende ASM Makro aus. Die Darstellung hier konzentriert sich dabei auf das Makro für das Mobiltelefon CELLPHONE. Dieses Makro wiederum besteht zunächst aus dem Empfangen einer Eingabenachricht über einen Eingabeport (**choose** inport **with** inputs(agent)(inport) ≠ [ ] ...). Die empfangene Nachricht inmsg = inputs(agent)(inport).first ist nun von der Form einer MMS Nachricht (dies wird durch die Kommunikationsinfrastruktur sichergestellt). Wie oben erläutert ist der erste Teil der Nachricht damit der Absender selbiger. Um an den Inhalt der Nachricht zu gelangen, wird folglich der zweite Teil der Nachricht selektiert (indoc = get-part(inmsg, 2)). Dieser Teil kann nun eine der Protokollnachrichten sein. Die folgenden Tests (is-passon, is-load, is-buy, is-present) überprüfen nun die Struktur und den Inhalt des Eingabedokuments und sorgen daraufhin für Abwicklung des richtigen Protokollschritts. Im Fall einer Lade-Nachricht, die ein Ticket für das Mobiltelefon enthält (entweder kommend von einem anderen Mobiltelefon durch Übertragen oder vom Kino-Server durch Kauf), wird z.B. die Liste der derzeit auf dem Telefon gespeicherten Tickets um das empfangene Ticket und dessen Absender erweitert (tickets(agent) := tickets(agent) + doclist(get-part(inmsg, 1) + get-part(indoc,2))). Dies geschieht allerdings nur, wenn noch nicht die maximal auf dem Telefon speicherbare Anzahl von Tickets erreicht ist (# tickets(agent) < MAXTICKETS). Im Fall einer Nachricht, die vom User initiiert wurde und

den Start des Kaufvorgangs für ein Ticket anzeigt (is-buy), wird z.B. eine Nachricht an den Server gesendet, welche die Kaufinformationen (get-part(indoc, 2), bereits durch die Nachricht des Users übermittelt) enthält (outdoc := doclist(intdoc(buyTicket) + get-part(indoc, 2))). Da es sich hier um eine verbindungslose MMS Übertragung handelt, wird der send-mode auf direct gesetzt und der Empfänger direkt angegeben (receiver := CINEMA;). Des Weiteren wird das bestellte Ticket in einer History Variablen booked gemerkt. Diese History Variablen werden später wichtig, um Sicherheitseigenschaften ausdrücken zu können (hier gilt z.B. "Der Kunde zahlt nicht mehr als er bestellt"). Für die anderen Protokollschritte (present, passOn) gelten ähnliche Spezifikationen. Zum Ende des Makros für das Mobiltelefon wird schließlich das SEND Makro aufgerufen, das anhand von send-mode, receiver, outport und outdoc die Zustellung von outdoc an den richtigen Kommunikationspartner übernimmt.

5.2 Ziele und Problemstellungen

Wie in 3.5 beschrieben und gerade am Beispiel illustriert sieht die PROSECCO Methodik die Unterteilung des abstrakten Protokollmodells anhand der am Protokoll beteiligten Agenten vor. So unterscheidet PROSECCO klar zwischen einem ehrlichen Protokollteilnehmer (wie z.B. einem Mobiltelefon oder einer Chipkarte eines ehrlichen Kunden), der sich an feste Protokollregeln hält, und einem böswilligen Angreifer, der durch Senden von falschen Nachrichten oder Unterdrücken bzw. Ändern von Nachrichten versucht, Vorteile aus dem System zu ziehen. Abstrahiert betrachten wir in PROSECCO also Abläufe von Protokollschritten der verschiedenen Agenten, die in dem jeweiligen Szenario interagieren. Zur Kommunikation sieht das PROSECCO Framework dazu eine generische Spezifikation von Nachrichten (Dokumente, s. Kap. 3.5 und 6) und von Verbindungen zwischen Agenten vor. Dabei beginnt der Protokollschritt jedes ehrlichen Agenten mit dem Empfangen einer Nachricht und endet mit dem Senden einer Nachricht (falls vorhanden). Der Schritt des Angreifers besteht aus der Generierung einer Nachricht aus seinem Wissen und dem Senden selbiger. Somit ergeben sich bei Ausführung einer PROSECCO ASM Abläufe nach dem Grundschema von Abb. 5.6.

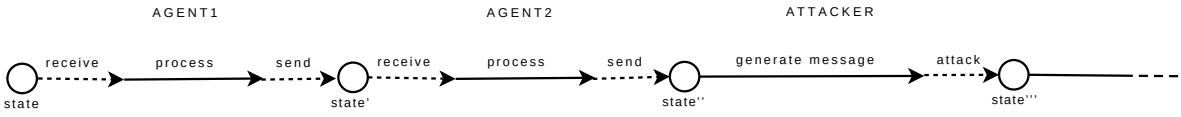


Abbildung 5.6: Schematischer Ablauf einer PROSECCO Spezifikation

Damit stellt sich die Frage, was eine korrekte Implementierung bzgl. einer solchen Spezifikation auszeichnet und welche Bedingungen erfüllt sein müssen, damit eine Implementierung als korrekt angesehen werden kann. Intuitiv soll die Verfeinerung garantieren, dass die Implementierung das gleiche Verhalten wie Ihre Spezifikation zeigt.

Die Unterteilung in Schritte einzelner Agenten in PROSECCO gibt bei der Verfeinerung zu Code dafür zunächst das Grundgerüst vor. Wir werden Schritt für Schritt einzelne Agenten des Modells durch konkrete Java Implementierungen ersetzen. Damit bedeutet korrektes Zustandsübergangsverhalten gleiches Ein-/Ausgabe Verhalten bezüglich der der Implementierung entsprechenden Agenten-Teilspezifikation. Ebenso bedeutet es, dass sich der interne

Zustand (z.B. der aktuell gespeicherte Geldwert auf einer Geldkarte oder die Ticketliste bei Cindy) genauso verändert, wie im abstrakten Modell gefordert.

Die restlichen am Protokoll beteiligten Agenten bleiben im jeweiligen Verfeinerungsschritt unverändert, womit sich ihr Zustandsübergangsverhalten trivial nicht ändert.

Die konkrete Ebene wird wiederum als ASM spezifiziert, wobei ein ASM Makro für einen Agenten durch eine diesem Agenten entsprechende Implementierung ersetzt wird. Insgesamt erhalten wir (bei exemplarischer Verfeinerung von Agent_1) eine Verfeinerungsstruktur wie in Abb. 5.7 gezeigt.

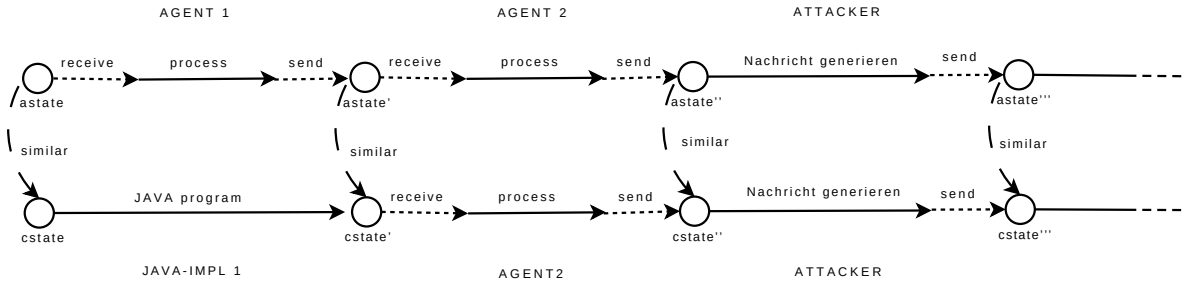


Abbildung 5.7: Schritte bei der Verfeinerung zu Code

Dabei müssen nun folgende Fragestellungen betrachtet werden:

- Wie findet die Einbettung eines Java Programms in eine Prosecco ASM unter Erhalt aller anderen Agentenspezifikationen konzeptionell statt? (Kap. 5.3)
- Welche Teile einer Implementierung eines Protokolls können verifiziert werden? (Kap. 5.3 und 10)
- Wie integriert man technisch ein Java Programm in eine ASM Spezifikation? (Kap. 5.4.2)
- Wie spezifiziert man die Ähnlichkeit (Simulationsrelation similar in Abb. 5.7) ? (Kap. 5.3.3)
- Muss man beim Übergang zu einer Implementierung zusätzliche Angriffe betrachten? (Kap. 8)

5.3 Vorgehen zur Verfeinerung

5.3.1 Die Implementierung

Eine reale Anwendung wie Cindy besteht dabei aus mehreren Komponenten, die verschiedene Aufgaben übernehmen. Typische Bereiche für eigene Softwarekomponenten innerhalb einer Implementierung eines Kommunikationsprotokolls sind z.B.

- eine graphische Oberfläche zur Kommunikation mit dem Benutzer

- ein oder mehrere Implementierungen zur Nutzung der Kommunikationsschnittstellen über das Netzwerk
- eine Implementierung kryptographischer Operationen
- eine Implementierung der eigentlichen Protokollfunktionalität

Zunächst ist es wesentlich, den Teil der Implementierung, der nun verifiziert werden soll, zu bestimmen.

Die ersten drei der obigen Komponenten sind für die Bedienbarkeit und Korrektheit des Kommunikationsprotokolls sicherlich wesentlich. Dennoch ist die formale Verifikation einer Implementierung z.B. einer graphischen Oberfläche nur von sehr beschränktem Nutzen, schließlich kann hier keine Sicherheitseigenschaft des Protokolls verletzt werden. Des Weiteren ist dies auch ein Systembestandteil, der massiv Parallelität und Multithreading zur Realisierung seiner Funktionalität benutzt. Dies schließt eine formale Verifikation mit derzeit bekannten Verifikationstools für z.B. Java auf Grund des fehlenden Beweissupports und der selbst bei Realisierung selbiger immensen entstehenden Komplexität aus.

Eine Implementierung einer Netzwerkkomponente (wie z.B. ein Netzwerk-Socket oder die Implementierung des MMS Zugriffs bei Cindy) ist dagegen zwar funktional für die Applikation sehr wichtig, kann jedoch ebenso wenig mit formalen Methoden für Java verifiziert werden. Dies liegt hier darin begründet, dass die Implementierungen für Netzwerkzugriffe nicht mehr in Java selbst programmiert werden, sondern über plattformabhängige native Implementierungen eingebunden werden. Im Weiteren kann natürlich die rein physikalische Übertragung von Nachrichten nicht mehr verifiziert werden und muss daher als korrekt angenommen werden.

Eine Implementierung kryptographischer Operationen wie Verschlüsselung, Entschlüsselung, digitale Signatur, Zufallszahlenberechnung oder Hashing ist ebenfalls für die Sicherheit der Anwendung unabdingbar. Der hier angestrebte Korrektheitsbegriff fußt jedoch auf einem Angreifermodell der abstrakten Ebene, der auf der Standard Annahme für Sicherheitsbetrachtungen von Kommunikationsprotokollen, der *perfekten Kryptographie*, beruht. Dies bedeutet, dass ein Angreifer in der abstrakten Welt kryptographische Operationen nicht unvorhergesehen wegen Schwächen der verwendeten Verfahren (zu kurze Schlüssellängen, Implementierungsfehler, ...) brechen kann. Es ist natürlich nicht möglich, solche Annahmen an einer realen Implementierung einer Kryptographie-Bibliothek zu verifizieren. Hier kann lediglich Korrektheit der Implementierung eines bestimmten Verfahrens (wie z.B. RSA) nachgewiesen werden. Die eigentliche Sicherheit des Verfahrens beruht in der Implementierung auf probabilistischen Werten basierend auf der algorithmischen Komplexität der Verfahren. Sicherheit bedeutet damit in der Realität lediglich, sich auf das Nicht-Eintreten eines Ereignisses mit einer möglichst kleinen Wahrscheinlichkeit zu verlassen. Für den Verfeinerungsbegriff in dieser Arbeit übernehmen wir daher die Annahme der perfekten Kryptographie auf die Implementierungsebene. Dies ist wesentlich für einen korrekten Verfeinerungsbegriff. Natürlich stellt dies in Bezug auf die Wirklichkeit eine Abstraktion dar. Es handelt sich allerdings um die gleiche Abstraktion, mit der wir reale Kryptoverfahren als "sicher" ansehen. Im Umkehrschluss führt diese Abstraktion dazu, dass natürlich die in der Verfeinerung verwendeten Eigenschaften der kryptographischen Operationen nicht mehr verifizierbar sind (sie stellen die Annahmen perfekter Kryptographie in der Realität dar). Damit ist auch die Implementierung einer solchen Komponente nicht mehr formal behandelbar. Weitere Details zur Behandlung kryptographischer Operationen im Refinement finden sich in Kap. 9.

Schließlich bleibt noch die Implementierung der eigentlichen Protokollfunktionalität, also dem Senden und Empfangen von Nachrichten, dem Verarbeiten dieser Nachrichten und der Abänderung des jeweiligen lokalen Zustands, der über die Reaktion auf weitere Nachrichten entscheidet. Dies ist der eigentlich interessante Teil der Implementierung, da hier die Vorgaben der abstrakten Spezifikation in Code umgesetzt werden. Diese Umsetzung ist dabei sehr fehleranfällig. Daher ist dies die Implementierung, an der wir im Rahmen der Verfeinerung interessiert sind und deren Korrektheit nachgewiesen werden soll.

Es ergibt sich somit eine allgemeine Architektur für unsere betrachteten Anwendungen nach Abb. 5.8.

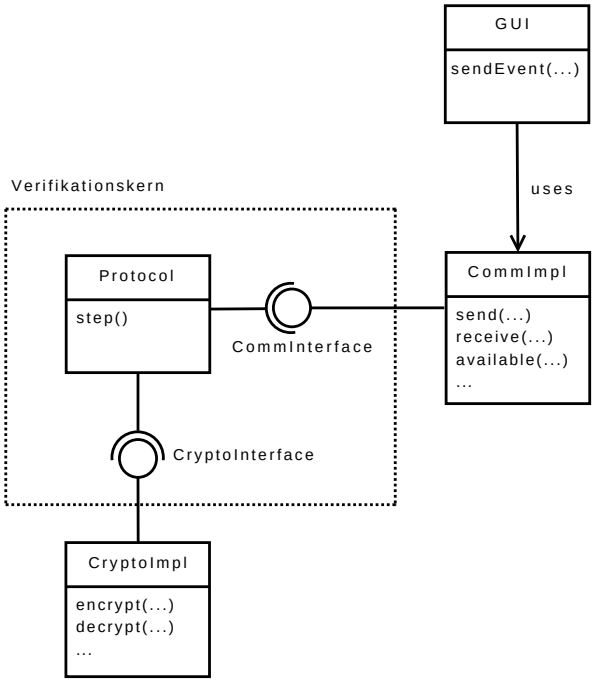


Abbildung 5.8: Typische Architektur betrachteter Anwendungen

Zur Abgrenzung der Komponenten verwenden wir in dieser Arbeit den bereits in [65] entwickelten Ansatz mittels Verifikationskernen. Dazu werden die beteiligten Komponenten untereinander über Interfaces von einander abgegrenzt. Die Protokollimplementierung erhält dabei Interface-getypte Referenzen auf ein Kryptographieinterface (zur Anbindung der kryptographischer Operationen) und ein Kommunikationsinterface (zur Anbindung nachrichtenorientierten Kommunikation). Zur Verifikation werden später lediglich Eigenschaften der Methoden dieser Interfaces verwendet ohne ihre tatsächliche Implementierung betrachten zu müssen. Über diesen Mechanismus können dann auch z.B. Annahmen perfekter Kryptographie für das Kryptographieinterface in der Verifikation realisiert werden.

Die Kommunikation zwischen graphischer Oberfläche und Protokollimplementierung erfolgt hier vereinfachend wiederum über nachrichtenorientierte Kommunikation über das Kommunikationsinterface. Die Oberfläche generiert also Eingabenachrichten als Kommandos nach dem Command Pattern. Diese werden (genauso wie Nachrichten über das Netzwerk) über das Kommunikationsinterface von der Implementierung empfangen. Antworten an die Oberfläche über

dieses Interface werden dann z.B. verwendet, um das aktuelle Ticket als Datamatrix Code auf dem Display anzuzeigen. Für die Cindy Anwendung ist dieses Vorgehen dabei ausreichend. Im Allgemeinen ist dies eine Vereinfachung, kann aber mittels eines weiteren Interfaces zur Kommunikation mit der Oberfläche verallgemeinert werden.

In der Cindy Anwendung (die ja ohne Kryptographie auskommt) ist die Implementierung daher von folgender Gestalt:

```
1
2  public interface CommInterface {
3      public boolean available(int port);
4      public void send(Document d, byte[] receiver,
5                       int port, int mode);
6      public Document receive(int p);
7  }
8
9
10 public class Protocol {
11
12     private CommInterface comm;
13     ...
14
15     public Protocol(CommInterface comm){
16         this.comm = comm;
17         ...
18     }
19
20     public void step(){
21         if(comm.available(USERPORT)){
22             Document inmsg = comm.receive(USERPORT);
23             if(inmsg != null) userStep(inmsg);
24         }
25         else if(comm.available(PHONEPORT)){
26             Document inmsg = comm.receive(PHONEPORT);
27             if(inmsg != null) phoneStep(inmsg);
28         }
29     }
30
31     private void phoneStep(Document inmsg) {
32         ...
33         if(outmsg != null) comm.send(outmsg, ...);
34     }
35
36     private void userStep(Document inmsg) {
37         ...
38         if(outmsg != null) comm.send(outmsg, ...);
39     }
40 }
```

Das Interface `CommInterface` bildet die nachrichtenorientierte Kommunikation ab. Im Vorgriff auf Kap. 6 erkennt man hier bereits die Abbildung der Document Datentypen der abstrakten Spezifikation durch eine entsprechende `Document` Java Klasse. Entsprechend den obigen Erläuterungen realisiert die Cindy Kommunikationsimplementierung dabei zwei verschiedene Kommunikationsports: `USERPORT` für die Kommunikation mit der graphischen Oberfläche und `PHONEPORT` für die Kommunikation mittels MMS Nachrichten. Das Kommunikationsinterface bietet die drei Operationen `available`, `send` und `receive` an. Die `available` Methode testet, ob auf dem jeweiligen Port eine Nachricht für das Protokoll vorliegt. `receive` empfängt daraufhin eine Nachricht vom angegebenen Port. Für die `send` Methode müssen nun (neben der eigentlichen Nachricht `Document d`) die drei Parameter `port` für Angabe des Kommunikationsports zum Senden der Nachricht (sowie bei verbindungsloser Kommunikation der `receiver` für Angabe des Namens des Empfängers und der `send-mode` für die Unterscheidung zwischen verbindungsorientierter und verbindungsloser Kommunikation) angegeben werden. Eine Referenz auf das `CommInterface` wird als Feld der Klasse `Protocol` gespeichert, um darüber die Kommunikation abzuwickeln.

Details zur eigentlichen Implementierung der Kommunikationsfunktionalität werden später in Kap. 10 gegeben.

5.3.2 Veränderungen im Zustand

Bei der Verfeinerung zu einer Implementierung wird ein Teil der ASM Spezifikation durch ein entsprechendes Java Programm ersetzt. Dabei werden die Zustandsfunktionen der konkreten Ebene teilweise durch Java Stores ersetzt, da der vorher dort gespeicherte Zustand jetzt in der Implementierung gespeichert wird.

Die abstrakte Prosecco ASM sieht allgemein Zustandsfunktionen nach folgendem Schema vor[2]:

Definition 5.3.1 Prosecco ***Zustand***

Der Zustand astate *einer* Prosecco *ASM ist das Kreuzprodukt der Zustandsfunktionen für den Kontext der ASM und der agentenlokalen Zustände. Der Kontext besteht aus den Zustandsfunktionen der Infrastruktur* inputs : agent → nat → documentlist *und* connections : connectionset *sowie dem Angreiferwissen* attacker-known : documentset. *Die Zustandsfunktionen der Agenten bilden die agentenlokalen Zustände. Damit ist* astate *definiert als:*

$$\text{astate} = \underbrace{\text{inputs} \times \text{connections} \times \text{attacker-known}}_{\text{acontext}} \times \underbrace{\text{statefun}_1 \times ... \times \text{statefun}_n}_{\text{aagentstate}}$$

Wir unterteilen diesen Zustand nun nach der Zugehörigkeit zu ehrlichen Agenten (diese können potentiell verfeinert werden) und dem Infrastrukturteil sowie dem Angreiferanteil. Die Vereinigung aller Zustandsfunktionen der ehrlichen abstrakten Agenten nennen wir aagentstate,

[2]Wir verwenden wir die Präfixe a und c zur Kennzeichnung der Zustandsfunktionen der abstrakten bzw. konkreten Ebene sowie später auch g für globale Zustände.

Infrastruktur und Angreiferzustand nennen wir acontext. Zum Infrastrukturanteil gehört z.B. die Zustandsfunktion inputs : agent → nat → Documentlist, die die derzeitigen Eingabequeues der Agenten darstellt. Ebenso gehört dazu die Zustandsfunktion connections : connectionset, die die derzeitigen Verbindungen zwischen den Agenten enthält. Beide Funktionen zusammen kodieren die Kommunikationsstruktur und ihren derzeitigen Zustand. Der Angreiferzustand wird durch das Angreiferwissen attacker-known : Documentset angegeben.

Der acontext wird bei einem Verfeinerungsschritt zu Code nicht ersetzt, bleibt also unverändert. Dies hat folgende Gründe:

Zum Einen soll der hier eingeführte Verfeinerungsbegriff die schrittweise Verfeinerung von Agentenspezifikationen aus Prosecco erlauben. So soll es möglich sein, in einem ersten Verfeinerungsschritt den ersten Agententyp (bei Cindy z.B. das Mobiltelefon) und in einem zweiten Verfeinerungsschritt einen zweiten (z.B. den Internet-PC) zu verfeinern. Dies hat den großen Vorteil der Modularisierbarkeit und damit entstehender Komplexitätsreduktion. Eventuell kann es sogar sein, dass die Implementierungen mancher Agententypen der Applikation nicht formal betrachtet werden sollen (z.B. die Datenbank des Kinoservers). Damit bleiben bei jedem Verfeinerungsschritt große Teile der Spezifikation unverändert. Insbesondere stützen sich diese unveränderten Teile des Modells immer noch auf die gegebene Kommunikationsinfrastruktur mit connections und inputs. Diese Teile dürfen also nicht entfernt oder verändert werden.

Darüber hinaus den Angreifer zu verfeinern ergibt in unserem Kontext ebenfalls keinen Sinn. Schließlich soll auf der Implementierungsebene immer noch die Möglichkeit zum Angriff gegeben sein. Eine Verfeinerung des Angreifers würde seine Implementierung bedeuten. Das abstrakte Modell des Angreifers ist mittels einer Menge von bekannten Dokumenten und der Generierung von neuen Dokumenten aus dieser Menge indeterministisch modelliert. Insbesondere ergeben sich durch diese Modellierung erst die Angriffsmöglichkeiten, die im wirklichen System auftreten können und erst dadurch entsteht eine sinnvolle Verifikationsumgebung für Sicherheitseigenschaften. Eine Implementierung z.B. in Java würde diesen Indeterminismus entfernen (insbesondere würde man damit alle Angriffe im vornherein festlegen - und kennen müssen). Das kann nicht sinnvoll sein.

Eine alternative Konkretisierung des Datentyps Document, auf dem das Angreifermodell beruht, kann ebenso nicht sinnvoll sein. Zum Einen soll der Angreifer weiterhin mit den nicht verfeinerten Agenten kommunizieren können und benötigt insbesondere auch dazu die Möglichkeit, Daten vom Typ Document zu erzeugen. Im Weiteren wäre damit eine sinnvolle Spezifikation von kryptographischen Operationen nicht möglich. Im Prosecco Ansatz werden kryptographische Operationen, wie z.B. das Verschlüsseln, als sicher betrachtet und können nur durch Kenntnis des entsprechenden kryptographischen Schlüssels umgekehrt werden. Zur sinnvollen Spezifikation solcher Fakten ist ein getypter Nachrichtendatentyp wie Document unerlässlich.

Im Gegensatz dazu wird bei der Verfeinerung eines Agententyps zu einem Java Programm natürlich dessen interner Zustand auf der konkreten Ebene durch Java Datentypen repräsentiert. Dieser interne Zustand ist dabei ein Teil der Zustandsfunktionen aus aagentstate. Beispielsweise existiert in der Spezifikation der Cindy Applikation eine dynamische Zustandsfunktion tickets : agent → Documentlist, welche die derzeit auf den Mobiltelefonen gespeicherten Tickets darstellt. Diese Liste wird in der Implementierung durch einen Java Datentyp dargestellt. Damit ist die Zustandsfunktion tickets in der konkreten Spezifikationsebene nicht mehr nötig und

kann durch einen Java Heap für jedes Mobiltelefon ersetzt werden (in welchem dann natürlich auch andere Informationen, wie z.B. die Telefonnummer des Telefons, abgelegt sind).

Diese Betrachtung motiviert die Notwendigkeit einer Datentypkonversion zwischen dem abstrakten Zustand der nicht verfeinerten Komponenten und dem konkreten Implementierungszustand der verfeinerten Agenten. Diese Konversion ist dabei an zwei Stellen notwendig: Zum Einen muss der interne abstrakte Zustand des Agenten, der verfeinert werden soll, im Rahmen der Simulationsrelation einem konkreten Java Zustand zugeordnet werden. Zum Anderen muss der verfeinerte Agent auch auf der konkreten Ebene Eingaben von anderen Agenten entgegennehmen und Ausgaben an diese produzieren können. Damit muss auch seine lokale Eingabe und Ausgabe jeweils mittels einer Datentypkonversion von abstrakten Dokumenten nach Java Daten und umgekehrt konvertiert werden.

Es ergeben sich damit folgende generelle Zustandfunktionen für die Spezifikation des Zustands der konkreten Ebene bei Verfeinerung des Agenten agent_j:

Definition 5.3.2 ***Verfeinerter*** PROSECCO ***Zustand bezüglich eines Agenten*** agent_j

Der verfeinerte Zustand cstate *einer* PROSECCO *ASM ist das Kreuzprodukt der Zustandsfunktionen für den Kontext der ASM und der agentenlokalen Zustände. Der Kontext besteht aus den Zustandsfunktionen der Infrastruktur* inputs : agent → nat → documentlist *und* connections : connectionset *sowie dem Angreiferwissen* attacker-known : documentset. *Die Zustandsfunktionen der Agenten bilden die agentenlokalen Zustände. Dabei werden alle Zustandsfunktionen, die in der Spezifikation von* agent_j *vorkommen, durch eine Zustandsfunktion* cstore : agent → store *ersetzt.*

$$\begin{aligned}\mathsf{cstate} = \quad & \underbrace{\mathsf{inputs} \times \mathsf{connections} \times \mathsf{attacker\text{-}known}}_{\mathsf{ccontext}} \\ & \times \underbrace{\mathsf{statefun}_i \times \ldots \times \mathsf{statefun}_n}_{\mathsf{cagentstate}} \\ & \times \underbrace{\mathsf{cstore} : \mathsf{agent} \rightarrow \mathsf{store}}_{\text{Java Stores}}\end{aligned}$$

wobei $\mathsf{statefun}_i \times \ldots \times \mathsf{statefun}_n$ eine Teilmenge der Zustandsfunktionen der abstrakten ASM sind. Dabei handelt es sich genau um die Zustandsfunktionen, die innerhalb des Spezifikationsanteils des verfeinerten Agenten agent_j *nicht* benutzt werden.

Als Spezialfall der Ersetzung von Zustandsfunktionen ist es noch möglich, dass eine Zustandsfunktion der abstrakten Ebene durch mehr als einen Agententyp benutzt wird. Zur Verdeutlichung: Die Zustandsfunktion tickets : agent → documentlist in der Cindy Anwendung ist zwar (aufgrund ihres Funktionstyps) für alle Agenten (also z.B. auch für Kinoserver oder Angreifer) definiert, wird allerdings nur innerhalb des Spezifikationsteils für das Mobiltelefon benutzt und verändert. Die Initialisierung der abstrakten Ebene stellt sicher, dass diese Liste für alle Agenten initial leer ist. Damit bleibt die Liste für alle nicht-Mobiltelefone auch während der Ausführung leer und insbesondere für deren Funktionalität irrelevant. Eine Entfernung bei der Verfeinerung ändert damit keine Funktionalität. Sollte eine Zustandsfunktion jedoch sinn-

voll für mehrere Agententypen benutzt werden, so muss diese bei der Verfeinerung natürlich auch im Zustand der konkreten Ebene erhalten bleiben. Im Folgenden wird o.B.d.A. davon ausgegangen, dass alle Zustandsfunktionen ausschließlich jeweils für einen Agenten verwendet werden. Dies lässt sich durch Duplikation und Umbenennung der Zustandsfunktionen immer erreichen.

Im Folgenden verwenden wir den Begriff "agentenlokale Zustandsfunktionen von Agent a" zur Bezeichnung der Teilmenge aller Zustandsfunktionen, die nur in der Teilspezifikation eines Agenten a verwendet werden.

5.3.3 Simulationsrelation

Im Folgenden verwenden wir für die Darstellung die Data Refinement Theorie.

Aufbauend auf der Definition der Zustände der beiden Ebenen lässt sich die gewünschte Simulationsrelation - und damit insbesondere ein wesentlicher Teil des intendierten Korrektheitsbegriffes - uniform definieren.

Intuitiv soll gelten, dass die konkrete Ebene das gleiche Zustandsübergangsverhalten zeigt wie die abstrakte Ebene. Dies bedeutet formal die Definition einer Extraktionsfunktion, die einen Java Zustand (also eine Objekthierarchie mit Pointerstrukturen) in einen zugehörigen abstrakten Zustand wandelt. Unter der Annahme, dass

$$(\mathsf{statefun}_1 : \mathsf{agent} \rightarrow \mathsf{statetype}_1) \times \ldots \times (\mathsf{statefun}_{i-1} : \mathsf{agent} \rightarrow \mathsf{statetype}_{i-1})$$

die agentenlokalen Zustandsfunktionen des verfeinerten Agenten sind, hat diese Funktion damit folgende Signatur:

$$\mathsf{extract} : \mathsf{store} \rightarrow (\mathsf{statetype}_1 \times \ldots \times \mathsf{statetype}_{i-1})$$

Details zur Spezifikation dieser Funktion werden allgemein in Kapitel 6 und speziell für Cindy in Kapitel 7 gegeben. An dieser Stelle ist lediglich wichtig, dass die extract Funktion den Zusammenhang zwischen abstrakten Datentypen und den Java Typen herstellt.

Im Weiteren sei acontext der Kontext der abstrakten Ebene und ccontext der Kontext der konkreten Ebene. Im Weiteren sei $\mathsf{aagentstate}_{(1..n)} := \mathsf{astatefun}_{(1..i-1)} \times \mathsf{astatefun}_{(i..n)}$ der abstrakte agentenlokale Zustand (bestehend aus dem Zustand $\mathsf{aagentstate}_{(1..i-1)} := \mathsf{astatefun}_1 \times \ldots \times \mathsf{astatefun}_{i-1}$ des verfeinerten Agenten und analog dem Zustand der nicht verfeinerten Agenten $\mathsf{aagentstate}_{(i..n)}$). Ebenso sei cagentstate der konkrete Zustand und cstore die konkrete Zustandsfunktion für die Java Stores. Ebenso sei $\mathsf{astate}_{(1..i-1)}(\mathsf{agent})$ eine Abkürzung für $\mathsf{astatefun}_1(\mathsf{agent}) \times \ldots \times \mathsf{astatefun}_{i-1}(\mathsf{agent})$.

Damit ergibt sich folgendes Grundgerüst für die Definition der Simulationsrelation, und damit auch der Korrektheitsbegriff, der durch ein korrektes Refinement garantiert wird:

$$
\begin{aligned}
&\mathsf{R}(\mathsf{acontext}, \mathsf{aagentstate}_{(1..i-1)}, \mathsf{aagentstate}_{(i..n)}, \mathsf{ccontext}, \mathsf{cagentstate}, \mathsf{cstore}) \\
\leftrightarrow\ & \quad \mathsf{acontext} = \mathsf{ccontext} \\
& \quad \wedge\ (\forall\ \mathsf{agent}.\ \mathsf{aagentstate}_{(1..i-1)}(\mathsf{agent}) = \mathsf{extract}(\mathsf{cstore}(\mathsf{agent}))) \\
& \quad \wedge\ \mathsf{aagentstate}_{(i..n)} = \mathsf{cagentstate} \\
& \quad \wedge\ \mathsf{INV}(\mathsf{cstore}) \\
& \quad \wedge\ \mathsf{INV}(\mathsf{aagentstate}_{(1..n)}) \\
& \quad \wedge\ \mathsf{INV}(\mathsf{acontext})
\end{aligned}
$$

Der Kontext (also Angreiferwissen und Eingabequeues) muss in beiden Ebenen identisch sein. Damit ist durch eine korrekte Verfeinerung auch die Korrektheit bezüglich des Ein-/Ausgabeverhaltens gegeben, modulo der Definition der Datentypumwandlung zwischen abstrakter Welt und Java Datentypen. Des Weiteren muss der abstrakte agentenlokale Zustand des verfeinerten Agenten gleich dem Ergebnis der Anwendung der Extraktionsfunktion auf den Java Store des Agenten sein. Damit wird die Korrektheit des Zustandsübergangsverhaltens der konkreten Ebene modulo der Definition von extract sichergestellt. Um diese Bedingungen zeigen zu können wird in der Praxis eigentlich immer eine zusätzliche Invariante gebraucht (die etwa Wohlgeformtheitsbedingungen auf Nachrichten oder ähnliches enthält). Aus Modularisierungsgründen wird diese Invariante aufgeteilt in drei Invarianten für den Java Anteil (INV(cstore)), den abstrakten Zustand (INV($\mathsf{aagentstate}_{(1..n)}$) und den Kontext (INV(acontext)).

Abschließend sei noch bemerkt, dass die genannte Simulationsrelation sowohl in einem auf Data Refinement als auch in einem auf ASM Refinement basierenden Verfeinerungsansatz verwendet werden kann. In letzterem Fall gibt sie sowohl die Definition für INV als auch für IO an.

5.3.4 Initialisierung

Neben der Korrektheit jedes einzelnen Protokollschritts ist auch die korrekte Initialisierung der Implementierung einer Komponente wichtig. Insbesondere beinhaltet die abstrakte Spezifikationsebene hier ein init(astate) Prädikat, das die Wohlgeformtheit des Initialzustandes der ASM beschreibt. Details hierzu finden sich in [71]. Im Fall der Cindy Anwendung besagt dieses Prädikat etwa, dass alle Eingabequeues leer sein müssen. Ebenso darf der Angreifer initial keine Passwörter der Benutzer oder zukünftige Datamatrix Codes der Tickets kennen. Darüber hinaus müssen auch die agentenlokalen Zustandsfunktionen Restriktionen erfüllen. So muss z.B.

$$\forall\ \mathsf{agent}.\ \mathsf{tickets}(\mathsf{agent}) = [\,]$$

gelten, kein Telefon darf also initial bereits Tickets besitzen. Auch die entsprechenden History Variablen wie booked müssen initial leer sein.

Diese Initialisierungsbedingungen müssen natürlich nun in der Implementierung genauso erhalten bleiben, um eine korrekte Verfeinerung zu erhalten. Dies ist für die Zustandsfunktionen des Kontexts trivial, da diese auf der konkreten Ebene sowieso unverändert bleiben. Lediglich für den agentenlokalen Zustandsteil für das Mobiltelefon muss substantiell etwas gezeigt werden. Am Beispiel bedeutet dies, dass auch die Java Implementierung mit einer leeren Ticketliste starten muss. Die Beweisverpflichtungen für ein korrektes Refinement (s. Kap. 5.4.5) fordern dies dann auch entsprechend.

5.4 Spezifikation der Verfeinerung

Es ist nun möglich, aufbauend auf den Vorgaben der vorangegangenen Kapitel, einen generischen Spezifikationsrahmen für die Verfeinerung von Kommunikationsprotokollen anzugeben und auch die Simulationsrelation zwischen abstrakter und konkreter Ebene generisch zu formulieren. Damit ergibt sich auch ein generischer Korrektheitsbegriff für Protokollimplementierungen.

Der allgemeine Rahmen ist dabei prinzipiell nach Abb. 5.9 aufgebaut. Die Darstellung orientiert sich hierbei an der Data Refinement Verfeinerungstheorie (s. Kap. 3). Bei Einsatz von ASM Refinement fallen die globalen Zustände weg. Die durch die Initialisierungs- und Finalisierungsoperationen ausgedrückten Bedingungen werden hier dann durch IR und OR angegeben. Im Folgenden verbleiben wir bei der Sichtweise des Data Refinements.

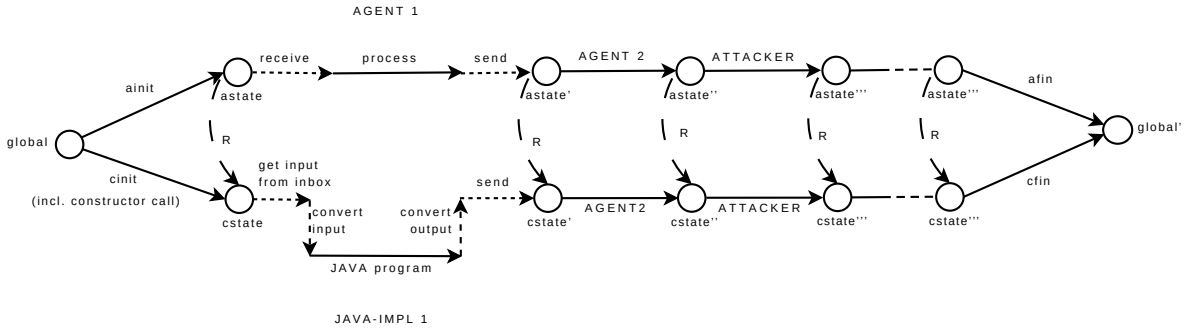


Abbildung 5.9: Struktur der Verfeinerung

Die Signatur der globalen Zustände entspricht der Signatur der abstrakten Ebene.

Die Initialisierung der konkreten Ebene entspricht der Initialisierung der abstrakten Ebene bis auf die Teile zur Initialisierung der agentenlokalen Zustandsfunktionen des verfeinerten Agenten. Diese werden nun im jeweiligen Java Store abgebildet. Zur Initialisierung wird hier der Konstruktor der jeweiligen das Protokoll implementierenden Klasse verwendet. Die Finalisierung wird durch die Identitätsfunktion auf der abstrakten Ebene und durch die bereits in der Simulationsrelation verwendete Extraktionsfunktion extract auf der konkreten Ebene spezifiziert.

Im Folgenden sollen die Einzelschritte und die Initialisierung im generischen Spezifikationsrahmen im Detail betrachtet werden.

5.4.1 Spezifikation der Einzelschritte

Die Schritte des verfeinerten Agenten werden wie in Abb. 5.10 in der konkreten Ebene spezifiziert.

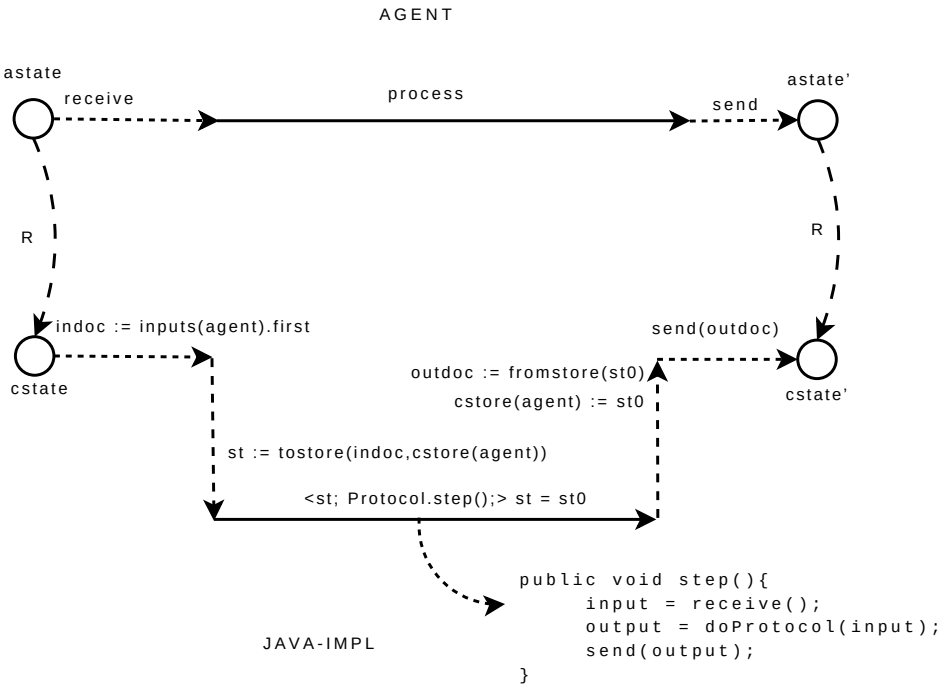


Abbildung 5.10: lokale Sicht auf einen Agenten

Zu Beginn des Schrittes des Agenten wird (ausgehend von einem Zustand, in dem die Simulationsrelation galt) die nächste Eingabe für den Agenten aus dessen lokaler Inputs-Funktion genommen. Diese Eingabe wird nun mittels einer *Konkretisierungsfunktion* im Rahmen der tostore Operation in einen Java Datentyp konvertiert und in den Speicher in einen temporären Zwischenspeicherplatz eingefügt. Der resultierende Speicher st_0 ist der Kontext für die Ausführung des Protokollschrittes in Java (`step()`). Hier handelt es sich nun um den funktionsfähigen Code der Anwendung. Deren Implementierung sieht nun zunächst das Empfangen einer Eingabe vor (`receive`). In Wirklichkeit wird hierbei die Kommunikationsinfrastruktur der Implementierung verwendet (bei Cindy z.B. eine MMS Nachricht oder ein Publish-Subscribe bzw. Command Mechanismus zur Kommunikation mit der graphischen Oberfläche). Zur Verifikation wird diese Methode jedoch axiomatisiert. Sie liefert damit die Eingabe zurück, die zuvor während tostore temporär dem Store hinzugefügt wurde. Anschließend wird diese Eingabe zur Abarbeitung der Protokollfunktionalität benutzt (hier durch eine Methode `doProtocol` illustriert). Diese Methode produziert eine Ausgabe `output`, welche schließlich mittels einer `send` Methode über die Kommunikationsinfrastruktur der Implementierung versendet wird. Auch hier verwenden wir eine Axiomatisierung für die Methode, welche dafür sorgt, dass die Ausgabe ebenfalls im Speicher temporär zwischengespeichert wird.

Ist dies erfolgt, wird die dadurch erzeugte Ausgabe mittels einer *Abstraktionsfunktion* im Rahmen von fromstore wieder in ein abstraktes Document outdoc konvertiert. Ebenso kann sich

durch den Protokollschritt der Store verändert haben (z.B. kann bei Cindy ein neues Ticket hinzugekommen sein). Daher wird die Zustandsfunktion cstore(agent) entsprechend aktualisiert. Die Ausgabe outdoc wird nun wiederum über die normale Kommunikationsstruktur der PROSECCO Welt über das SEND Makro an den richtigen Empfänger versendet. Schließlich muss im resultierenden Zustand wieder die Simulationsrelation gelten.

Sollten weitere Angaben (wie z.B. bei Cindy auf Grund der verbindungslosen Kommunikation der send-mode oder der receiver) für das SEND Makro benötigt werden, so müssen diese natürlich als Parameter der Java send Methode angegeben werden und dann mittels fromstore ebenso aus dem Speicher extrahiert werden.

Bemerkung: Die Darstellung hier kann den Eindruck erwecken, dass wir während der Verfeinerung nur die möglichen Ergebnisse einer Konkretisierungsfunktion angewandt auf die abstrakten Eingabe als Eingabe auf der konkreten Ebene betrachten. Dies ist nicht der Fall. Viel mehr geben wir später eine Implementierung für eine Kommunikationsinfrastruktur an (also eine Implementierung der `receive` und `send` Methoden), die es uns erlaubt, alle anderen in der konkreten Ebene möglicherweise zusätzlich vorkommenden Eingaben (man denke an z.B. nicht wohlgeformte Nachrichten) generisch auszusortieren. Tatsächlich ist dann eine weitere Spezifikation der konkreten Ebene, in der auch solche ungültigen Eingaben betrachtet werden, eine Verfeinerung der hier dargestellten konkreten Spezifikationsebene. Mehr zu dieser Überlegung wird in Kap. 8.1 und bei der Darstellung der Implementierung der Kommunikationsschicht in Kap. 10.4 erklärt.

Dieser Spezifikationsrahmen kann so für alle PROSECCO Spezifikationen verwendet werden. Die Funktionen tostore und fromstore sind dabei jeweils leicht an die jeweilige Fallstudie anzupassen. Einen generischen Rahmen gibt der nächste Abschnitt vor. So ist z.B. für mehrere Eingabeports, wie dies bei der Cindy Anwendung der Fall ist, eine getrennte Behandlung der jeweiligen Eingabequeues während tostore nötig. Insbesondere bekommt dann auch die receive Methode der Java Implementierung einen zusätzlichen Parameter, der den Eingabeport angibt, auf dem empfangen werden soll. Bei Cindy ist etwa ein Port für die Kommandobasierte-Kommunikation mit dem User Interface zuständig und ein anderer Port repräsentiert die Sende- und Empfangsinfrastruktur mittels MMS Nachrichten.

5.4.2 Integrationsprinzip in die ASM

Die technische Integration des beschriebenen Verfeinerungsrahmens für PROSECCO ASMs basiert auf der Tatsache, dass die verwendete Logik (und damit auch das KIV System) keinen Unterschied zwischen Java Programmformeln, Formeln der dynamischen Logik und ASM Programmformeln macht:

Die bisherige abstrakte Definition des verfeinerten Agenten wird aus der PROSECCO ASM entfernt. Der Zustand der ASM wird entsprechend Def. 5.3.2 angepasst. Alle anderen Teile der ASM (inklusive der Hauptregel STEP) bleiben unverändert. An die Stelle der bisherigen Definition des verfeinerten Agententyps AGENT_R tritt eine neue ASM Regel der Form:

```
AGENT_R(agent; inputs, cstore, ...)
   let outdoc = ⊥ in {
      TOSTORE(agent, inputs; cstore);
```

```
choose st with
    ∃ st0. st0 = cstore(agent) ∧
          ⟨st0 ; Protocol.theinstance.step(); ⟩ st0 = st in {
    FROMSTORE(agent, st; cstore, inputs, outdoc, outport); }
if outdoc ≠ ⊥ then SEND(outdoc, outport; inputs, ...)}
```

Die Definition der ASM Regel für den Agenten agent beginnt zunächst mit der Deklaration einer lokalen Variable outdoc : Document für die spätere Ausgabe. Danach wird mittels des TOSTORE Makros die Eingabe für den Agenten in den Store geschrieben sowie einige andere Initialisierungen durchgeführt (s. unten). Im Anschluss beinhaltet der Speicher cstore(agent) die Eingaben für den Agenten agent. Im Kontext dieses Speichers wird nun das eigentliche Java Programm mit der Protokollfunktionalität `step()` aufgerufen. Der Speicher st, der dann später in FROMSTORE als Ergebnis der konkreten Operation als neuer Speicher des Agenten weiterverwendet wird, wird hier technisch mittels einem **choose** Konstrukt ausgewählt. In dessen **with** Bedingung wird gerade die erfolgreiche Abwicklung des aktuellen Protokollschritts gefordert und der Speicher st als deren Nachfolgezustand (ausgehend vom aktuellen cstore(agent)) determiniert. Das anschließende FROMSTORE Makro wandelt u.a. die Ausgabe des Java Programms in ein abstraktes Document outdoc zurück, welches im Anschluss mittels des normalen PROSECCO SEND Makros versendet wird.

```
TOSTORE(agent, inputs; cstore)
  if inputs(agent)(port1) = [ ] then
     cstore(agent) := cstore(agent)[.ins1, null];
  if inputs(agent)(port1) ≠ [ ] then
     cstore(agent) :=
        cstore(agent)[doc2java(.ins1, inputs(agent)(port1).first, cstore(agent))];
  ...
  if inputs(agent)(portn) = [ ] then
     cstore(agent) := cstore(agent)[.insn, null];
  if inputs(agent)(portn) ≠ [ ] then
     cstore(agent) :=
        cstore(agent)[doc2java(.insn, inputs(agent)(portn).first, cstore(agent))];
  cstore(agent) := cstore(agent)[.outport, intval(0)];
  cstore(agent) := cstore(agent)[.outs, null];
  cstore(agent) := cstore(agent)[.rec1, boolval(false)];
  ...
  cstore(agent) := cstore(agent)[.recn, boolval(false)];
```

Das TOSTORE Makro muss zunächst den Speicher für die Abwicklung des nächsten Protokollschrittes vorbereiten. Intuitiv kann man sich vorstellen, dass hier die Sende- und Empfangsinfrastruktur der realen Welt simuliert wird. Die Implementierung kann hierbei, wie oben bereits erklärt, mehrere Ports für mögliche Eingaben besitzen. Daher existiert im Speicher für

jeden Port i eine temporäre Speicherstelle ins_i. Die Initialisierung dieser Speicherstellen erfolgt damit anhand des jeweiligen Zustands der Eingabequeue inputs(agent)(i). Ist diese leer (... = []), dann wird die entsprechende temporäre Speicherstelle mit dem Java Wert `null`[3] initialisiert, was keine Eingabe repräsentiert. Falls nicht, muss mittels einer Konkretisierungsfunktion doc2java die Eingabe inputs(agent)(i).first in einen Java Datentyp verwandelt werden (Details dazu s. Kap. 6.4). Ergebnis dieser Funktion doc2java(.ins_n, inputs(agent)(port$_n$).first, cstore(agent)) ist ein veränderter cstore(agent), in dessen künstlichem Feld cstore(agent)[.ins_n] ein Referenzwert steht, mit dem die Java Repräsentation des Dokumentes inputs(agent)(port$_n$).first abgebildet wird.

Ausser den Eingaben müssen noch andere temporäre Felder gesetzt werden, die ebenfalls für die Abwicklung der Kommunikation nötig sind. So muss der temporäre Speicherort für den Ausgabeport zurückgesetzt werden (cstore(agent)[.outport, intval(0)]). Dieser Port signalisiert, auf welchem Port das Ausgabedokument versandt werden soll. Analog muss die temporäre Speicherstelle für die Ausgabe selbst (`.outs`) auf `null` zurückgesetzt werden. Schließlich exisiert noch für jeden Eingabeport i ein boolsches Feld rec_i. Dieses Feld signalisiert, ob am jeweiligen Port eine Java receive Operation aufgerufen wurde oder nicht. Dies ist wichtig, um zu wissen, ob im Anschluss an den Java Methodenaufruf die abstrakte Eingabequeue des jeweiligen Ports zu verkürzen ist oder nicht. Vor dem Schritt werden alle diese Felder mit `false` initialisiert.

```
FROMSTORE(agent, st; cstore, inputs, outdoc, outport)
    cstore(agent) := st;
    outport := cstore(agent)[.outport].intval;
    outdoc := java2doc(.outs, cstore(agent))
    if (cstore(agent)[.rec1]) then inputs(agent)(1) = inputs(agent)(1).rest;
    ...
    if (cstore(agent)[.recn]) then inputs(agent)(n) = inputs(agent)(n).rest;
```

Das FROMSTORE Makro erhält nun den Speicher, den das Programm zur Abwicklung der Java Protokollfunktionalität als Nachfolgezustand hinterlassen hat (st). Zu Beginn des Makros wird die Zustandsfunktion cstore(agent) daher auf diesen neuen Speicher aktualisiert. Im Anschluss werden die für die Kommunikationsinfrastruktur wichtigen Parameter outport (für den Ausgabeport) und outdoc (für die eigentliche Ausgabe) aus dem Speicher ausgelesen. Im Fall des Ausgabedokuments ist hierbei eine zu der in TOSTORE verwendeten Funktion doc2java inverse Abstraktionsfunktion java2doc nötig. Im Anschluss daran werden schließlich anhand der .rec_i Felder noch die Eingabequeues des Agenten um ein Element verkürzt, falls nötig.

Der hier vorgestellte Spezifikationsrahmen ist für jeweils maximal eine Empfangs- und eine Sende-Operation pro Protokollschritt vorgesehen. Dies hat sich in den bisherigen Betrachtungen als ausreichend erwiesen. Dennoch kann der Rahmen einfach auf mehr als eine Empfangsoperation erweitert werden. Dann muss die TOSTORE Operation die gesamte Eingabequeue inputs(agent)(i) für jeden Port i in den Speicher legen. Ebenso müssen dann die rec_i Felder einen Zahlenwert haben, der angibt, wie oft empfangen wurde. Analog muss natürlich FROM-

[3] hier als Abkürzung für den tatsächlichen Wert refval(jvmref) verwendet

STORE angepasst werden. Für mehr als eine Sendeoperation muss schließlich auch das .outs Feld eine Liste bzw. ein Array von Ausgaben beinhalten und ebenso das .outport Feld.

5.4.3 Spezifikation der Initialisierung

Neben der Spezifikation für die einzelnen Protokollschritte muss auch die Initialisierung der PROSECCO ASM angepasst werden. Hier muss nun für jede Instanz des verfeinerten Agententyps der Konstruktor der Implementierung aufgerufen werden. Ein init Prädikat aus der PROSECCO ASM kann ebenso einfach auf die neue Spezifikation angepasst werden. Es werden wiederum alle Teile der Definition des Prädikats, die sich mit dem Zustand des verfeinerten Agenten befassen, entfernt. Statt dessen wird der Aufruf des Konstruktors für jeden verfeinerten Agenten hinzugefügt. Es ergibt sich eine Definition der Form:

```
   init_concrete(globalstate, ccontext, cagentstate, cstore)
↔    init(ccontext)
   ∧ init(cagentstate)
   ∧ ∀ agent. is_refined_agent(agent) →
           ∃ st.    okinitstore(st)
                ∧ ⟨st; Protocol.theinstance =
                          new Protocol(initparams); ⟩ st = cstore(agent)
```

Die Definitionen für die Initialisierung des Kontexts ccontext und der Zustände der nicht verfeinerten Agenten (cagentstate) ergeben sich direkt aus der Defintition der Initialisierungsbedingung init(aagentstate) der PROSECCO ASM, da ja aagentstate(i..n) = cagentstate. Der globale Zustand ist für die Initialisierung nicht wichtig, erst bei der Finalisierung werden die globalen Zustände (identisch zu denen der abstrakten Ebene) aus dem Speicher extrahiert. Daher werden hier keine Anforderungen an den globalstate gestellt. Die letzten drei Zeilen geben den Konstruktoraufruf für alle Instanzen des verfeinerten Agententyps vor (is_refined_agent). Der Aufruf des Konstruktors selbst muss bereits im Kontext eines wohlgeformten Java Speichers beginnen. Dies wird angegeben durch das Prädikat okinitstore für den initialen Speicher. So muss z.B. der Ausführungsmodus st[.mode] = normal sein, es darf also aktuell keine Exception vor Ausführung des Konstruktors vorliegen. Im Weiteren müssen noch einige andere Wohlgeformtheitskriterien für den Speicher gelten, z.B. müssen eventuelle Initialisierungsparameter initparams für den Konstruktor wohlgeformt und vom richtigen Typ sein. Initialisierungsparameter könnten z.B. kryptographische Schlüssel oder geheime Informationen sein (wie ein Passwort, auf das das Programm prüfen soll). Diese Bedingungen werden in Kap. 7 nochmals am Beispiel von Cindy genauer erläutert.

Im Kontext eines solchen Speichers soll nun der Konstruktor für die das Protokoll implementierende Klasse (hier `Protocol`) aufgerufen werden. Die daraus resultierende Referenz wird im statischen Feld `Protocol.theinstance` gespeichert, damit im Anschluss darauf die step() Methode für die Abwicklung der Funktionalität aufgerufen werden kann. Schließlich fordert die Nachbedingung des Konstruktoraufrufs (∀ agent. ... ⟨... ⟩ st = cstore(agent)), dass die Zustandsfunktion cstore der konkreten Ebene nun aus genau den aus diesen Aufrufen

resultierenden Stores gebildet ist.

5.4.4 Spezifikation der Finalisierung

Bei der Finalisierung werden nun alle Zustände der jeweiligen Ebenen in den Globalzustand extrahiert. Damit besteht globalstate aus den Zustandsfunktionen der abstrakten Ebene. Da wir an der Vererbung von Invarianzeigenschaften $\varphi(globalstate, globalstate')$ für alle ($globalstate$, $globalstate'$) $\in APROG$ interessiert sind, ist dabei auch die Teilmenge der Zustandsfunktionen ausreichend, die für φ massgeblich sind. Im Folgenden gehen wir der Einfachheit halber immer von der gesamten Signatur der abstrakten Ebene aus.

Für die abstrakte Ebene ergibt sich einfach die Identität (mit $globalstate := agentstate_{(1..n)} \times context$):

$$
\begin{array}{ll}
 & \mathsf{fin\_abstract(acontext,\ aagentstate,\ agentstate}_{(1..n)} \times \mathsf{context)} \\
\leftrightarrow & \quad \mathsf{context} = \mathsf{acontext} \\
 & \wedge\ \mathsf{agentstate}_{(1..n)} = \mathsf{aagentstate}
\end{array}
$$

Die Finalisierung der konkreten Ebene verwendet dagegen wiederum die Extraktionsfunktion (mit einer Zerlegung der agentenlokalen Zustandsfunktionen analog zu Kap. 5.3.3) :

$$
\begin{array}{ll}
 & \mathsf{fin\_concrete(ccontext,\ cagentstate,\ cstore,\ agentstate}_{(1..n)} \times \mathsf{context)} \\
\leftrightarrow & \quad \mathsf{context} = \mathsf{ccontext} \\
 & \wedge\ (\forall\ \mathsf{agent.\ agentstate}_{(1..i-1)}\mathsf{(agent)} = \mathsf{extract(cstore(agent)))} \\
 & \wedge\ \mathsf{agentstate}_{(i..n)} = \mathsf{cagentstate}
\end{array}
$$

5.4.5 Resultierende Beweisverpflichtungen

Schließlich muss nun die Data Refinement Theorie (s. Kap. 3.4.1) mittels des vorgestellten Spezifikationsrahmens instantiiert werden. In KIV wird dazu die Spezifikationsinstantiierung aus Kap. 3.4.3 verwendet. Dies erfordert insbesondere die Definition der relationalen Operationen des Data Refinements mittels der dynamischen Logik im Rahmen des Signaturisomorphismus. Die Definition der Instantiierung folgt dabei folgendem Schema zur Definition einer Relation OP zwischen Zuständen state und state' mittels einem (indeterministschen) Programm OP#. Die beiden in Relation stehenden Zustände werden durch Anfangs- und Endzustände des dazugehörigen Programms definiert:

$$
\mathsf{OP(state,\ state')} \leftrightarrow \langle \mathsf{OP\#(;\ state)} \rangle\ \mathsf{state} = \mathsf{state'}
$$

Damit diese Relation (wie beim hier verwendeten Data Refinement gefordert, s. Definition von Datentypen in Kapitel 3.4.1) total ist, muss noch die Terminierung von OP#(; state) gezeigt werden:

$$\vdash \langle\!|\mathsf{OP\#(;\ state)}|\!\rangle\ \mathsf{true}$$

Die Menge der möglichen Operationen im Data Refinement auf abstrakter Ebene beinhaltet damit ausschließlich die Einzelschritt-Regel STEP_A (s. Kap. 3.5.3) der PROSECCO ASM. Die Indexmenge I ist damit einelementig $\mathsf{I} = \{\mathsf{A}\}$. Die einzige Operation ist also AOP_A.

Die konkrete Indexmenge ist analog $\mathsf{I} = \{\mathsf{C}\}$ und die einzige Operation COP_C wird analog mit der Schrittregel STEP_C der konkreten ASM Spezifikation definiert.

Damit spezifizieren wir:

SPEZIFIKATION:

$$\begin{aligned}
&\mathsf{AOP}_A(\mathsf{inputs, connections, attackerknown, statefun}_1, \ldots, \mathsf{statefun}_n,\\
&\qquad \mathsf{inputs', connections', attackerknown', statefun'}_1, \ldots, \mathsf{statefun'}_n)\\
\leftrightarrow\ &\langle \mathsf{STEP}_A\mathsf{\#(;inputs, connections, attackerknown, statefun}_1, \ldots, \mathsf{statefun}_n) \rangle\\
&\qquad (\quad \mathsf{inputs} = \mathsf{inputs'} \wedge \mathsf{connections} = \mathsf{connections'}\\
&\qquad \wedge\ \mathsf{attackerknown} = \mathsf{attackerknown'}\\
&\qquad \wedge\ \mathsf{statefun}_1 = \mathsf{statefun'}_1 \wedge \ldots \wedge \mathsf{statefun}_n = \mathsf{statefun'}_n\ )
\end{aligned}$$

und analog für COP:

SPEZIFIKATION:

$$\begin{aligned}
&\mathsf{COP}_C(\mathsf{inputs, connections, attackerknown, statefun}_1, \ldots, \mathsf{statefun}_k, \mathsf{cstore},\\
&\qquad \mathsf{inputs', connections', attackerknown', statefun'}_1, \ldots, \mathsf{statefun'}_k, \mathsf{cstore'})\\
\leftrightarrow\ &\langle \mathsf{STEP}_C\mathsf{\#(;inputs, connections, attackerknown, statefun}_1, \ldots, \mathsf{statefun}_k) \rangle\\
&\qquad (\quad \mathsf{inputs} = \mathsf{inputs'} \wedge \mathsf{connections} = \mathsf{connections'}\\
&\qquad \wedge\ \mathsf{attackerknown} = \mathsf{attackerknown'}\\
&\qquad \wedge\ \mathsf{statefun}_1 = \mathsf{statefun'}_1 \wedge \ldots \wedge \mathsf{statefun}_k = \mathsf{statefun'}_k\\
&\qquad \wedge\ \mathsf{cstore} = \mathsf{cstore'})
\end{aligned}$$

Die Initialisierungsoperationen INIT werden als äquivalent zu den Prädikaten init aus der PROSECCO ASM bzw. dem oben vorgestellten Prädikat init_concrete definiert.

Als Beweisverpflichtungen ergeben sich damit aus der Instantiierung:

THEOREM:

Initialisierung:

init_concrete(ginputs, gconnections, gattackerknown,
gstatefun$_1$, ..., gstatefun$_n$,
cinputs, cconnections, cattackerknown,
cstatefun$_1$, ..., cstatefun$_k$, cstore)
$\vdash$
init_abstract(ginputs, gconnections, gattackerknown,
gstatefun$_1$, ..., gstatefun$_n$,
ainputs, aconnections, aattackerknown,
astatefun$_1$, ..., astatefun$_n$)
$\wedge$ R(ainputs, aconnections, aattackerknown, astatefun$_1$, ..., astatefun$_n$,
cinputs, cconnections, cattackerknown, cstatefun$_1$, ..., cstatefun$_k$, cstore)

THEOREM:

Korrektheit:

R(ainputs, aconnections, aattackerknown, astatefun$_1$, ..., astatefun$_n$,
cinputs, cconnections, cattackerknown, cstatefun$_1$, ..., cstatefun$_k$, cstore),
$\langle$STEP$_C$#(;cinputs, cconnections, cattackerknown, cstatefun$_1$, ..., cstatefun$_k$) $\rangle$
(cinputs = cinputs' $\wedge$ cconnections = cconnections'
$\wedge$ cattackerknown = cattackerknown'
$\wedge$ cstatefun$_1$ = cstatefun'$_1$ $\wedge$ ... $\wedge$ cstatefun$_k$ = cstatefun'$_k$
$\wedge$ cstore = cstore')
$\vdash$
$\langle$STEP$_A$#(;ainputs, aconnections, aattackerknown, astatefun$_1$, ..., astatefun$_n$) $\rangle$
R(ainputs, aconnections, aattackerknown,
astatefun$_1$, ..., astatefun$_n$,
cinputs', cconnections', cattackerknown',
cstatefun'$_1$, ..., cstatefun'$_k$, cstore')

THEOREM:

Finalisierung:
R(ainputs, aconnections, aattackerknown,
astatefun$_1$, ..., astatefun$_n$,
cinputs, cconnections, cattackerknown,
cstatefun$_1$, ..., cstatefun$_k$, cstore),
fin_concrete(cinputs, cconnections, cattackerknown, cstatefun$_1$, ..., cstatefun$_k$,
ginputs, gconnections, gattackerknown, gstatefun$_1$, ..., gstatefun$_n$)
$\vdash$
fin_abstract(ainputs, aconnections, aattackerknown,
astatefun$_1$, ..., astatefun$_n$,
ginputs, gconnections, gattackerknown,
gstatefun$_1$, ..., gstatefun$_n$)

Die wichtigste Beweisverpflichtung ist dabei die Bedingung **Korrektheit**. Der Beweis dieser Aussage erfolgt nun zunächst durch symbolische Ausführung des konkreten ASM Schrittes STEP$_C$. Da dessen Definition der Definition der PROSECCO ASM gleicht (s. Kap. 3.5), erhalten wir daraus eine Fallunterscheidung über den Agenten, der einen Protokollschritt ausführen soll. Da alle Makro-Definitionen bis auf den verfeinerten Agenten in der konkreten Ebene identisch sind, sind fast alle Fälle trivial und lassen sich durch entsprechende symbolische Ausführung des abstrakten Schrittes STEP$_A$ und Auswahl des entsprechenden Agenten schließen. Es verbleibt damit als Hauptbeweisverpflichtung für die Korrektheit des verfeinerten Agenten folgende Beweisverpflichtung (hier dargestellt nach Aufruf des Makros für den verfeinerten Agenten (s. Kap. 5.4.2) und nach entsprechender symbolischer Ausführung der abstrakten ASM):

THEOREM:

Correctness-Refined-Agent:
R(ainputs, aconnections, aattackerknown,
astatefun$_1$, ..., astatefun$_n$,
cinputs, cconnections, cattackerknown,
cstatefun$_1$, ..., cstatefun$_k$, cstore),
$\langle$TOSTORE(agent, cinputs; cstore) $\rangle$ cstore = cstore$_0$,
st$_0$ = cstore$_0$(agent),
$\langle$st$_0$; `Protocol.theinstance.step();` $\rangle$ st$_0$ = st,
$\langle$FROMSTORE(agent, st; cstore$_0$, cinputs, coutdoc, coutport) $\rangle$
(cstore$_0$ = cstore$_1$ $\wedge$ cinputs = cinputs$_1$

$$
\begin{array}{l}
\quad \wedge\ \mathsf{coutdoc} = \mathsf{coutdoc}_0 \wedge \mathsf{coutport} = \mathsf{coutport}_0) \\
\langle \textbf{if}\ \ \mathsf{coutdoc}_0 \neq \bot\ \textbf{then}\ \ \mathsf{SEND}(\mathsf{coutdoc}_0, \mathsf{coutport}_0; \mathsf{cinputs}_1, \mathsf{cattackerknown}, \ldots) \rangle \\
\quad (\mathsf{cinputs}_1 = \mathsf{cinputs}_2 \wedge \mathsf{cattackerknown} = \mathsf{cattackerknown}_2 \wedge \ldots) \\
\ldots \\
\vdash \\
\langle \mathsf{AGENT}_R(\mathsf{agent}, \mathsf{ainputs}, \mathsf{aconnections}, \mathsf{aattackerknown}, \\
\qquad\qquad \mathsf{astatefun}_1, \ldots, \mathsf{astatefun}_n) \rangle \\
\qquad \mathsf{R}(\mathsf{ainputs}, \mathsf{aconnections}, \mathsf{aattackerknown}, \\
\qquad\quad \mathsf{astatefun}_1, \ldots, \mathsf{astatefun}_n, \\
\qquad\quad \mathsf{cinputs}_2, \mathsf{cconnections}, \mathsf{cattackerknown}_2, \\
\qquad\quad \mathsf{cstatefun}_1, \ldots, \mathsf{cstatefun}_k, \mathsf{cstore}_1)
\end{array}
$$

Diese Beweisverpflichtung gibt dabei genau die Kommutierungsbedingung für das Diagramm aus Abb. 5.10 an.

Diese Beweisverpflichtungen und ihre Beweise werden in Kap. 7 für die Cindy Applikation und in Kapitel 13 für die Mondex Anwendung im Detail nochmals am Beispiel konkretisiert.

Kapitel 6

Referenz-Datentypen für sichere Kommunikationsprotokolle

Dieses Kapitel stellt die verwendeten Datentypen der Implementierung vor. Insbesondere wird eine generische Implementierung der Documents gegeben. Des Weiteren werden Funktionen zur Wandlung der abstrakten Typen in Implementierungstypen spezifiziert. Zusätzlich charakterisieren wir einen Gültigkeitsbegriff auf den entstehenden Java Repräsentationen. Die Behandlung der kryptographischen Operationen wird erst später in Kap. 9 eingeführt. Die in diesem Kapitel beschriebenen Arbeiten wurden in [68] und [66] publiziert.

6.1 Abstrakte Datentypen in Kommunikationsprotokollen

Wie bereits in Kap. 3.5 erläutert, verwenden Protokollspezifikationen nach dem PROSECCO Ansatz den Datentyp Document, um Nachrichten und auch interne Zustände zu spezifizieren. Die Verwendung dieses Datentyps ist auf Grund des generischen Angreifermodells in PROSECCO wichtig, da anhand dieses Datentyps die Fähigkeiten des Angreifers beschrieben werden.

Zur kurzen Wiederholung hier nochmals die algebraische Spezifikation des Datentyps Document, diesmal in vollständiger KIV Syntax mit den Typüberprüfungsprädikaten für jeden Subtyp. Dabei ist z.B. isIntDoc(doc) $\leftrightarrow \exists$ i. doc = intdoc(i).

SPEZIFIKATION:

```
Document =
              IntDoc(value : int) with isIntDoc
            | NonceDoc(nonce : Nonce) with isNonceDoc
            | SecretDoc(secret : Secret) with isSecretDoc
            | KeyDoc(key : Key) with isKeyDoc
            | HashDoc(hash : Document) with isHashDoc
```

```
              | EncDoc(key : Key, doc : Document) with isEncDoc
              | SigDoc(key : Key, doc : Document) with isSigDoc
              | Doclist(docs : Documentlist) with isDoclist

Documentlist =
    [ ] with isEmptyList
  | . + . (.first : Document, .rest : Documentlist) with isDocumentlist

Nonce = mkNonce(value : int) with isNonce
Secret = mkSecret(value : int) with isSecret
Key = mkKey(value : int) with isKey
```

Auf Grund der zentralen Rolle dieses Datentyps ist es wichtig, eine einheitliche und ebenso generische Implementierung für den Datentyp in Java anzugeben. Des Weiteren müssen Mapping Funktionen java2doc und doc2java zur Wandlung zwischen abstrakter und konkreter Ebene angegeben werden. Diese Funktionen dienen zur Spezifikation der extract Funktion im Rahmen der Simulationsrelation (s. Kap. 5.3.3) und der Makros FROMSTORE und TOSTORE (s. Kap. 5.4.2).

6.2 Diskussion

Die wichtigste Frage bei der Abbildung der Datentypen in die Implementierungsebene ist die Frage, ob die Abbildung typerhaltend erfolgen muss. Aus reiner Implementierungssicht würde man wohl zunächst abstreiten, die Angabe des Typs einer Nachricht in einer Implementierung unbedingt zu benötigen. Aus Sicht einer korrekten Verfeinerung sind die Typangaben allerdings wesentlich. Folgende Beispiele sollen dies illustrieren:

Wir betrachten im Folgenden eine einfache Passwort Überprüfung zur Authentifizierung eines Benutzers. Als Eingabe für die Überprüfungsoperation soll ein Document indoc dienen, das im Erfolgsfall als Wert ein SecretDoc mit dem richtigen Passwort als Secret enthält. Unter der Annahme, dass es eine Zustandsfunktion password : agent → secret gibt, die die Passwörter der Agenten abbildet, sowie eine weitere Funktion checkok : agent → boolean, die den Erfolg oder Misserfolg des Passwort-Checks angibt, kann eine erste Spezifikation für die Überprüfung für einen Agenten agent wie folgt aussehen:

```
CHECKPASSWORD(agent, password; inputs, checkok)
  if (inputs(agent)(1) ≠ [ ]) then {
   indoc = inputs(agent)(1).first;
   inputs(agent)(1) := inputs(agent)(1).rest;
   if (isSecretDoc(indoc)) then {
      if (indoc.secret = password(agent)) then checkok(agent) := true
      else checkok(agent) := false }
```

Nehmen wir im Weiteren an, dass diese Spezifikation nun implementiert werden soll. Wählen wir dazu zunächst eine Implementierung, in der Eingabenachrichten ohne Typ dargestellt werden sollen. Ohne Beschränkung der Allgemeinheit wählen wir zur Illustration eine Implementierung mittels einer einfachen Folge von bytes. Eine Konkretisierungsfunktion, die unseren abstrakten Datentyp Document in die Implementierungswelt abbildet, wäre damit eine Funktion, die den jeweiligen innersten Dokumentenwert (also z.B. den Integer value bei einem IntDoc) in eine Folge von bytes konvertiert. Eine solche Funktion doc2javabytes würde exemplarisch für SecretDoc und IntDoc wie folgt spezifiziert werden[1]:

doc2javabytes-intdoc:
doc2javabytes(Intdoc(i)) = int2bytes(i)

doc2javabytes-secretdoc:
doc2javabytes(SecretDoc(secret(i))) = int2bytes(i)

Ebenso können wir uns eine sehr einfache Java Implementierung in folgendem Stil vorstellen, wobei das Kommunikationsinterface jetzt natürlich ungetypte Folgen von bytes versenden und empfangen muss:

```
1
2  public class PasswordCheck {
3    private static PasswordCheck theinstance;
4
5    public boolean checkok;
6    public byte[] password;
7    public CommInterface comm;
8
9    public PasswordCheck(CommInterface c, byte[] password){
10     this.comm = c;
11     this.password = password;
12   }
13
14   public void checkPassword(){
15     byte[] input = comm.receive();
16     if(ByteArray.equals(input, password)) checkok = true;
17     else checkok = false;
18   }
19 }
```

Darauf aufbauend wäre eine mögliche Simulationsrelation[2] R wie folgt definiert[3]:

[1] Die Funktion int2bytes übernimmt hierbei die Abbildung eines int Wertes in eine byte Folge.

[2] Wir verzichten in der Darstellung hier der Einfachheit halber auf für die Diskussion nicht wesentlicher Zustandsfunktionen wie z.B. attackerknown.

[3] Die Hilfsfunktion getbytearray : reference × store → bytes liefert das byte Array, das im gegebenen Speicher an der gegebenen Referenz gespeichert ist.

```
   R(ainputs, checkok, password, cinputs, cstore)
↔    ainputs = cinputs
   ∧ ∀ agent.    cstore(agent)[.theinstance.checkok].boolval = checkok(agent)
               ∧ doc2javabytes(password(agent)) =
                        getbytearray(cstore(agent)[.theinstance.password].refval,
                                     cstore(agent))
```

Schließlich müsste während des TOSTORE Makros die Eingabe als `byte[]` in den Speicher des jeweiligen Agenten gelegt werden. Dessen Definition beinhaltet damit mindestens folgende Zeile[4]:

```
TOSTORE#
   ...
   if inputs(agent) ≠ [ ]  then
   cstore(agent) := addbytearray(.ins,
                          doc2javabytes(cinputs(agent)(1).first), cstore(agent))
   ...
```

Eine solche Implementierung müsste nun, um eine korrekte Verfeinerung der Spezifikation zu sein, die Korrektheitsaussagen nach Kap. 5.4.5 erfüllen. Dies bedeutet, dass zu jedem Ablauf der konkreten Ebene ein Ablauf der abstrakten Ebene gefunden werden muss, dessen Nachfolgezustand wiederum die Simulationsrelation R erfüllt. Betrachten wir einen Ausgangszustand, in dem gilt:

$$inputs(agent_k)(1) = secretdoc(password(agent_k)) + [\,]$$

Für diese Eingabe wird während TOSTORE ein byte Array in den Speicher eingefügt, das dem int Wert $password(agent_k).value$ entspricht. Für dieses byte Array gilt im Code nach dem Empfangen selbigens als input natürlich `ByteArray.equals(input, password)` auf Grund der Simulationsrelation. Damit wird `checkok = true`. Das gleiche gilt ebenso für die abstrakte Spezifikation und damit gilt auch nach deren Schritt wieder die Simulationsrelation. Für diese Eingabe stimmt also die Verfeinerung.

Betrachten wir nun eine Eingabe:

$$inputs(agent_k)(1) = intdoc(password(agent_k).value) + [\,]$$

[4] addbytearray(rk, bytes, st) fügt in den Speicher st an der Stelle rk die byte-Folge bytes ein.

Eingabe ist also ein IntDoc, dessen int-Wert gerade dem Wert des Passworts entspricht. Da während TOSTORE der Typ der Eingabe vergessen wird, gilt die selbe Argumentation wie im vorherigen Fall und es gilt wiederum checkok = true. Im Abstrakten gilt nun allerdings natürlich *nicht* isSecretDoc(indoc), womit hier checkok(agent) := false gilt. Die Verfeinerung ist also nicht korrekt.

Diese Diskussion zeigt, dass eine Implementierung einer solchen Spezifikation in der Lage sein muss, die Typen der Eingabedokumente prüfen zu können. Will man dies um jeden Preis vermeiden, kann man nun noch argumentieren, dass eine abstrakte Spezifikation dann keine Typüberprüfungsoperationen verwenden darf. Damit wäre die Spezifikation (die Veränderung ist durch *Kursivdruck* hervorgehoben):

```
CHECKPASSWORD(agent, password; inputs, checkok)
  if (inputs(agent)(1) ≠ [ ]) then {
   indoc = inputs(agent)(1).first;
   inputs(agent)(1) := inputs(agent)(1).rest;
   if (indoc = secretdoc(password(agent))) then checkok(agent) := true
   else checkok(agent) := false }
```

Hier tritt nun allerdings bei Eingabe eines Dokumentes, das kein SecretDoc ist, wiederum das gleiche Problem wie oben beschrieben auf: Die Implementierung überprüft lediglich den Wert, die Spezifikation überprüft durch den Test auf Gleichheit zu secretdoc(password(agent)) doch implizit den Typ. Die Verfeinerung ist also wiederum nicht korrekt.

Schließlich liegt der Gedanke nahe, auch in der Spezifikation lediglich den Wert zu überprüfen. Spezifizieren wir als letzte Möglichkeit also wie folgt:

```
CHECKPASSWORD(agent, password; inputs, checkok)
  if (inputs(agent)(1) ≠ [ ]) then {
   indoc = inputs(agent)(1).first;
   inputs(agent)(1) := inputs(agent)(1).rest;
   if (indoc.secret = password(agent)) then checkok(agent) := true
   else checkok(agent) := false }
```

Hier tritt jetzt ein anderes Problem auf: Bei Eingabe eines IntDocs wird hier eine Selektorfunktion .secret angewendet, die für den Typ IntDoc (durch die Datentypspezifikation für Dokumente) *unspezifiziert* ist. Damit kann der aus indoc.secret resultierende potentiell password(agent) sein oder eben auch nicht. Da dies nicht definiert ist, kann für diese Spezifikation

die Korrektheit der Verfeinerung nicht gezeigt werden[5].

Wir sehen also, dass eine Implementierung für die Dokumente, die Typinformationen erhält, für eine korrekte Verfeinerung wesentlich ist.

6.3 Eine Implementierung

Konzeptionell entspricht eine Data Specification eines Summentyps, wie sie beim Document Typ vorliegt, in der Java Implementierungsebene einer Klassenhierarchie, die mittels verschiedener Subklassen einer abstrakten Klasse die verschiedenen Typen des Summentyps nachbildet.

In diesem Abschnitt beschränken wir uns dabei in der Darstellung auf die Subtypen des Document Typs, die keine kryptographischen Operationen abbilden. Diese sind IntDoc, NonceDoc, SecretDoc, KeyDoc und Doclist. Die kryptographischen Typen HashDoc, SigDoc und EncDoc werden in Kap. 9 behandelt.

Wir verwenden zur Abbildung der Dokumente eine Implementierung nach dem Klassendiagramm in Abb. 6.1.

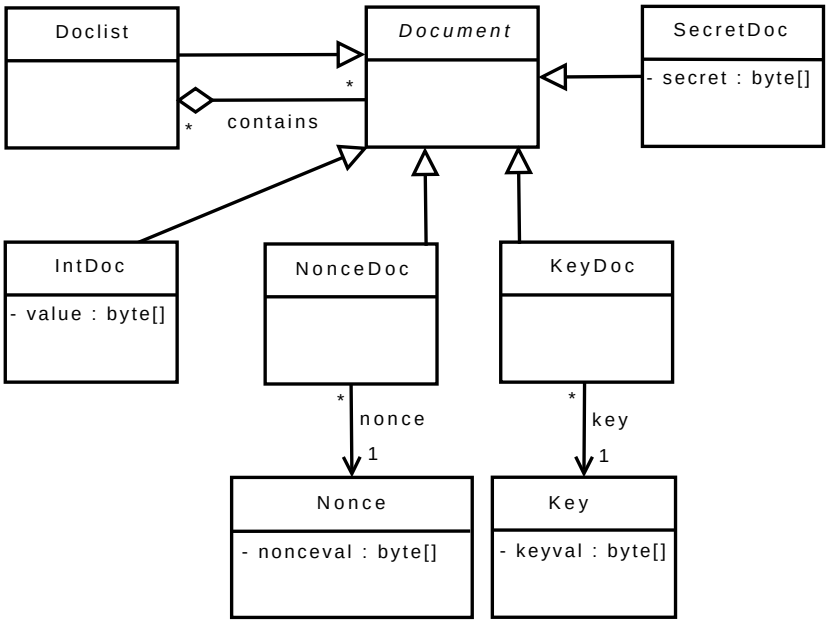


Abbildung 6.1: Klassenhierarchie für Dokumente ohne Kryptographie

Als Wurzel der gesamten Hierarchie dient eine abstrakte Klasse `Document`. Deren Subklassen bilden nun die verschiedenen Subtypen des abstrakten Document Typs ab.

Die reinen Integer-Werte der abstrakten Dokumente werden in der Implementierung durch byte Arrays umgesetzt. Dies erlaubt im Gegensatz zu einer Abbildung mit dem normalen

[5] Wäre die Selektorfunktion .secret für IntDoc definiert, würde auch das das Problem nicht lösen. Der einzig sinnvolle Rückgabewert bei Anwendung einer solchen "falschen" Selektorfunktion wäre ein konstanter Default-Wert x. Damit wäre die Implementierung wiederum falsch, solange password(agent) $\neq$ x.

Java `int` Typ auch die Implementierung sehr großer IntDocs. Der Java `int` Typ besitzt lediglich einen Wertebereich von $-2^{31} \leq i \leq 2^{31} - 1$. In PROSECCO Spezifikationen wird der IntDoc Typ allerdings auch häufig dazu verwendet, beliebige Daten abzubilden. Bei der Cindy Applikation werden mit ihm z.B. die Daten des bestellten Tickets (also Dinge wie Filmname, Sitzplatznummer, Kinonummer, ...) abgebildet, ohne diese näher zu spezifizieren. Betrachtet man die Aneinanderreihung dieser Daten einfach als große Zahl, so erhält man schnell Werte, die den Wertebereich des Java `int` Typs übersteigen. Aus diesem Grund verwenden wir eine Folge von bytes zur Abbildung. Im KIV Java Kalkül gibt es vereinfachend für Arrays keine Maximallänge. Damit wird rein formal gesehen die Abbildung beliebig langer IntDocs möglich. Bezogen auf echtes Java ist dies eine Abstraktion, Java Arrays haben eine Maximallänge. Diese ist der maximale `int` Wert $2^{31}-1$. Damit können in der Realität IntDocs mit einem abstrakten int Wert i im Bereich von $-256^{(2^{31})} \leq i \leq 256^{(2^{31}-1)} - 1$ abgebildet werden. Dieser Wert ist (insbesondere auf die hauptsächlich betrachteten Anwendungsdomänen "mobile Endgeräte" und "Chipkarten") groß genug um jemals als realer Nachrichteninhalt (schon rein technisch) auftreten zu können.

Ein abstraktes IntDoc(i) entspricht damit einer Pointerstruktur, die aus einem `IntDoc` Objekt mit einer Referenz auf ein `byte[]` mit Wert int2bytes(i) besteht. Analog werden auch SecretDocs abgebildet. Der eigene Secret Typ ist in der Implementierung nicht notwendig, daher wird das `byte[]` direkt in der Klasse `SecretDoc` referenziert.

Die NonceDoc und KeyDoc Typen werden auch durch entsprechende Subklassen von `Document` abgebildet. Hier werden die Werte allerdings noch in eigene Klassen `Nonce` und `Key` ausgelagert. Der Grund hierfür ist die häufige Speicherung und Benutzung dieser Werte ohne ihre "Dokumentenhülle" in Anwendungen. So soll eine Bibliothek für kryptographische Operationen zur Durchführung einer Verschlüsselungsoperation lediglich einen `Key` benötigen und nicht gleich ein `KeyDoc`. Ebenso soll die Generierungsfunktion für Zufallszahlen eine `Nonce` statt eines `NonceDocs` zurückgeben.

Die Dokumentenliste documentlist wird in der Implementierung als Array `Document[]` abgebildet. Ihre Hüllklasse `Doclist` bildet den entsprechenden abstrakten Doclist Typ ab und implementiert die gegenseitig rekursive Definition der abstrakten Ebene.

Diese Hierarchie implementiert damit eine typerhaltende Abbildung der abstrakten Dokumente.

6.4 Von abstrakten Datentypen nach Java

Für die eigentliche Abbildung der abstrakten Datentypen im Rahmen der Verfeinerung müssen nun Konkretisierungsfunktionen und Abstraktionsfunktionen definiert werden. In diesem Abschnitt definieren wir dazu die Funktionen doc2java und im nächsten Abschnitt dann java2doc für die beiden Transformationsrichtungen. Diese beiden Funktionen werden zur Definition der extract Funktion in der Simulationsrelation in der Verfeinerung und auch innerhalb der Definitionen von TOSTORE und FROMSTORE verwendet.

doc2java soll ein abstraktes Document in eine Pointerstruktur transformieren und diese dem Speicher hinzufügen. Dazu müssen neue Referenzen in den Speicher eingefügt werden. Eine neue Referenz ist dabei eine Referenz, die noch nicht als Referenz eines refkey rk = r – sk als Schlüssel im Speicher vorkommt. Dazu verwenden wir ein Prädikat newref : reference × store.

Es gilt

$$\mathsf{newref(r,st)} \leftrightarrow \neg\ \mathsf{r} \in \mathsf{st}$$

mit

$$\mathsf{r} \in \mathsf{st} \leftrightarrow \exists\ \mathsf{sk}.\ \mathsf{r}\ \text{-}\ \mathsf{sk} \in \mathsf{st}$$

und

$$\neg\ \mathsf{rk} \in \emptyset$$

$$\mathsf{rk} \in \mathsf{st}[\mathsf{rk}_0,\ \mathsf{val}] \leftrightarrow \mathsf{rk} = \mathsf{rk}_0 \vee \mathsf{rk} \in \mathsf{st}$$

Zur Konstruktion solcher neuer Referenzen verwenden wir eine Funktion newref : store → reference. Es gilt:

$$\mathsf{newref(newref(st),\ st)}$$

Zur Konstruktion einer einzelnen neuen Referenz mittels der Funktion newref : store → reference wird eine Ordnung auf den Referenzen definiert. Dann wird einfach die nächste nicht im Speicher vorkommende Referenz bezüglich dieser Ordnung von newref zurückgegeben. Für die Konstruktion mehrerer neuer (und untereinander disjunkter) Referenzen auf einmal gibt es die Funktion newref_list(i, st), die per einfacher Rekursion eine Liste von i neuen Referenzen bezüglich st erstellt.

Im Weiteren benötigen wir noch Funktionen, um dem Store Objekte hinzuzufügen. Dies wird mittels addobj : reference × javatype × fieldinits × store → store für neue Objekte und addarray : reference × javatype × javavalues × store → store für neue Arrays durchgeführt. Die Funktion addobj(r, ty, fss, st) liefert dabei einen neuen Speicher zurück, der an der Referenz r ein Objekt vom Typ[6] ty enthält und ansonsten unverändert gegenüber st ist. Die fieldinits fss sind dabei eine Liste von Tupeln aus fieldspec und javavalue. Diese Liste gibt damit die initialen Werte der Felder des Objektes an. Analog dazu fügt addarray(r, ty, vals, st) ein neues Array mit

[6] Ein Typ ty ist dabei ein Datentyp javatype = mkclasstype(.class : String) | mkarraytype(.type : javatype) | int_type | byte_type | ... , der die möglichen Typen in Java angibt. Abkürzend für mkclasstype(classname) werden wir im Folgenden einfach classname schreiben.

*Element*typ ty an der Referenz r in den Speicher st ein. Die Initialwerte und die Länge des Arrays werden dabei durch die Liste von javavalues vals angegeben.

Damit können wir nun doc2java spezifizieren. Die Funktion soll als Rückgabewert den neuen Speicher nach Einfügen des Objekts und auch die Referenz, die als Wurzel der neuen Zeigerstruktur eingefügt wurde, zurückgeben. Letzteres ist wichtig, um das Objekt nach dem Einfügen referenzieren zu können. Damit erhalten wir als Signatur doc2java : Document × store → reference × store.

Als Beispiel betrachten wir zunächst den IntDoc Typ:

SPEZIFIKATION:

doc2java-intdoc:
$r_1 + r_2 = \text{newref\_list}(2, st)$
$\rightarrow \text{doc2java}(\text{IntDoc}(i), st) =$
r_1
$\times \text{addobj}(r_1, \text{IntDoc}, .\text{value} \times \text{refval}(r_2),$
$\text{addarray}(r_2, \text{byte\_type}, \text{int2bytes}(i), st))$

Gegeben seien hier zwei neue Referenzen r_1 und r_2 bzgl. eines Speichers st. Dann erfolgt das Einfügen eines IntDoc(i) durch Einfügen eines byte Arrays (addarray(...)) an Referenz r_2 mit den byte Werten int2bytes(i). Dieses Array ist nun der Wert des `.value` Feldes eines `IntDoc` Objekts, das im Anschluss an Referenz r_1 eingefügt wird. r_1 bildet auch gleichzeitig die Wurzel der entstehenden Pointerstruktur.

Genauso erfolgt das Transformieren eines SecretDoc:

SPEZIFIKATION:

doc2java-secretdoc:
$r_1 + r_2 = \text{newref\_list}(2, st)$
$\rightarrow \text{doc2java}(\text{SecretDoc}(\text{secret}), st) =$
r_1
$\times \text{addobj}(r_1, \text{SecretDoc}, .\text{secret} \times \text{refval}(r_2),$
$\text{addarray}(r_2, \text{byte\_type}, \text{int2bytes}(\text{secret.int}), st))$

Analog dazu erfolgt das Wandeln eines KeyDoc und eines NonceDoc, allerdings muss hier noch jeweils ein Key und Nonce Objekt konstruiert werden. Man benötigt also drei neue Referenzen statt nur zwei:

SPEZIFIKATION:

doc2java-noncedoc:

$$
\begin{aligned}
& r_1 + r_2 + r_3 = \text{newref\_list}(3, st) \\
\rightarrow\ & \text{doc2java}(\text{NonceDoc}(nonce), st) = \\
& \quad r_1 \\
& \times \text{addobj}(r_1, \text{NonceDoc}, .\text{nonce} \times \text{refval}(r_2), \\
& \quad \text{addobj}(r_2, \text{Nonce}, .\text{nonceval} \times \text{refval}(r_3), \\
& \quad\quad \text{addarray}(r_3, \text{byte\_type}, \text{int2bytes}(nonce.int), st))
\end{aligned}
$$

SPEZIFIKATION:

doc2java-keydoc:

$$
\begin{aligned}
& r_1 + r_2 + r_3 = \text{newref\_list}(3, st) \\
\rightarrow\ & \text{doc2java}(\text{KeyDoc}(key), st) = \\
& \quad r_1 \\
& \times \text{addobj}(r_1, \text{KeyDoc}, .\text{key} \times \text{refval}(r_2), \\
& \quad \text{addobj}(r_2, \text{Key}, .\text{keyval} \times \text{refval}(r_3), \\
& \quad\quad \text{addarray}(r_3, \text{byte\_type}, \text{int2bytes}(key.int), st))
\end{aligned}
$$

Schließlich fehlt noch die Umwandlung der Doclist. Diese muss nun rekursiv über die Liste der enthaltenen Dokumente erfolgen. Wir spezifizieren daher eine neue Funktion docs2java : documentlist × store → references × store. Die Funktion fügt eine ganze Liste von Dokumenten in der Speicher ein und gibt alle Wurzelreferenzen der dabei entstandenen Pointerstrukturen zurück. Ihre Spezifikation ist damit:

SPEZIFIKATION:

docs2java-empty:

$$\text{docs2java}([\,], st) = [\,] \times st$$

docs2java-list:

$$
\begin{aligned}
& \text{doc2java}(doc, st) = r \times st_0 \\
\wedge\ & \text{docs2java}(docs, st_0) = refs \times st_1 \\
\rightarrow\ & \text{docs2java}(doc + docs, st) = r + refs \times st_1
\end{aligned}
$$

Damit können wir nun das Transformieren einer Doclist spezifizieren:

SPEZIFIKATION:

doc2java-doclist:

$$
\begin{aligned}
&\quad \mathsf{refs} \times \mathsf{st}_0 = \mathsf{docs2java}(\mathsf{docs}, \mathsf{st}) \\
&\wedge\ \mathsf{r}_1 + \mathsf{r}_2 = \mathsf{newref\_list}(2, \mathsf{st}_0) \\
&\rightarrow\ \mathsf{doc2java}(\mathsf{Doclist}(\mathsf{docs}), \mathsf{st}) = \\
&\qquad \mathsf{r}_1 \\
&\quad \times\ \mathsf{addobj}(\mathsf{r}_1, \mathsf{Doclist}, .\mathsf{docs} \times \mathsf{refval}(\mathsf{r}_2), \\
&\qquad\quad \mathsf{addarray}(\mathsf{r}_2, \mathsf{Document}, \mathsf{mkjavavalues}(\mathsf{refs})\ , \mathsf{st}_0))
\end{aligned}
$$

Das Einfügen einer Doclist erfolgt zunächst analog zu den obigen Spezifikationen durch Anlegen eines neuen Objekts mit `Doclist` Typ. Dessen Feld `.docs` beinhaltet eine Referenz r_1 auf ein neues Array mit Typ `Document[ ]`. Dieses Feld implementiert die Aggregation contains zwischen `Doclist` und `Document` in Abb. 6.1. Es beinhaltet die Referenzwerte (mkjavavalues(refs))[7], die aus dem Aufruf von docs2java für die Dokumentenliste entstanden sind.

Damit ist die Konkretisierungsfunktion doc2java für die nicht-kryptographischen Dokumente vollständig spezifiziert.

6.5 Anwendung von doc2java im Refinement

Die Funktion doc2java wird nun im Verfeinerungsframework innerhalb der Definition des TOSTORE Makros verwendet (s. Kap. 5.4.2). Sie dient dazu, eine Pointerstruktur in den Speicher einzufügen, die der abstrakten Eingabe auf den jeweiligen Eingabeports der verfeinerten Agenten entspricht. Die in Kap. Kap. 5.4.2 verwendete Funktion doc2java : refkey × document × store → store ist dabei nur eine Wrapper Funktion für das hier beschriebene doc2java : document × store → store × reference und ist demnach wie folgt axiomatisiert:

SPEZIFIKATION:

doc2java-wrapper:

$$
\begin{aligned}
&\quad \mathsf{doc2java}(\mathsf{doc}, \mathsf{st}) = \mathsf{r} \times \mathsf{st}_0 \\
&\rightarrow\ \mathsf{doc2java}(\mathsf{rk}, \mathsf{doc}, \mathsf{st}) = \mathsf{st}_0[\mathsf{rk}, \mathsf{refval}(\mathsf{r})]
\end{aligned}
$$

Nach Ausführung des TOSTORE Makros wird die Implementierung des verfeinerten Agenten aufgerufen (z.B. die `Protocol.step()` Methode bei Cindy). Hierin wird dann eine receive() Operation aufgerufen, die nun in der Realität mittels der Kommunikationsinfrastruktur-Implementierung (z.B. Zugriff auf MMS Nachrichten oder Kommandos der Oberfläche) auf das

[7] mkjavavalues verwandelt eine Liste von Referenzen refs in eine Liste von Referenzwerten refvals.

Netzwerk zugreift und eine reale Eingabe zurückliefert. In der Verifikation fordern wir nun von dieser Implementierung der `receive()` Methode, dass sie eine Instanz des `Document` Typs zurückliefert. Dabei handelt es sich dann genau um die Instanz, die zuvor während TOSTORE mittels java2doc in den Speicher eingefügt wurde.

Damit könnten wir als Eigenschaft für den Verfeinerungsbeweis für die `receive` Methode folgende Eigenschaft fordern:

SPEZIFIKATION:

receive:

$$\begin{aligned} &\text{validref("document-prog", r, CommInterface, st)} \\ &\quad \wedge\ \mathsf{r} \neq \text{jvmref} \\ &\quad \wedge\ \mathsf{st} = \mathsf{st}_0 \\ \rightarrow\ &\langle\, \mathsf{st}\ ;\ \mathsf{r}_1 = \mathsf{r}.\text{receive}()\ \rangle\ (\mathsf{st} = \mathsf{st}_0[.\text{rec}, \text{boolval(true)}] \wedge \mathsf{r}_1 = \mathsf{st}[.\text{ins}]) \end{aligned}$$

Die `receive` Methode soll also genau den Pointer auf die Struktur zurück liefern, die am Anfang des Schrittes des verfeinerten Agenten mittels TOSTORE und doc2java und den Speicher an die Stelle `.ins` eingefügt wurde. Sollten mehrere Ports für die Anwendung benötigt werden, wird hier dann natürlich für jeden Port ein entsprechendes Axiom wie oben benötigt. Die weitere Voraussetzung validref(...) beschreibt, dass es sich bei der Invoking-Expression r des Methodenaufrufs um eine ordentliche Referenz mit einem Subtyp von `CommInterface` handelt (s. dazu auch Kap. 3.6.3). Der Speicher wird durch `receive` nicht verändert, lediglich der boolsche Indikator st[.rec] wird auf `true` gesetzt. Dies wird (wie auch in Kap. 5.4.2 erläutert) von FROMSTORE später benötigt, um die abstrakte Eingabequeue entsprechend zu kürzen.

Wir gehen mit obiger Eigenschaft davon aus, dass eine Netzwerkschicht existiert, die in der Lage ist, Objekte des `Document` Typs zwischen den Agenten zu verschicken. Wir werden in Kap. 10.4 insbesondere eine Implementierung für eine solche Schicht angeben.

6.6 Von Java nach abstrakten Datentypen

Betrachten wir nun die inverse Transformation. Die Funktion java2doc soll die Transformation von Java Pointerstrukturen in abstrakte Dokumente durchführen. In dieser Richtung ist die Transformation allerdings etwas komplizierter, da Pointerstrukturen auftreten können, bei denen eine normale rekursive Definition inkonsistent wäre. Dies sind die zyklischen Pointerstrukturen. Bei einer zyklischen Pointerstruktur (wie z.B. in Abb. 6.2 als Objektdiagramm gezeigt) ist zudem nicht klar, was das Ergebnis von java2doc sein soll, da die termerzeugten abstrakten Documents natürlich nicht zyklisch sein können.

Wir benutzen daher zunächst ein Prädikat cyclic : string × reference × store, welches generisch angibt, ob die Pointerstruktur beginnend bei einer gegebenen Referenz im gegebenen Speicher einen Zyklus enthält. Ein Zyklus ist dabei intuitiv das Vorkommen einer Schleife auf einem Pfad, der bei der gegebenen Referenz beginnt und über das Entlangschreiten entlang der Felder der Referenz und der weiteren so erreichbaren Referenzen entsteht. Der erste String-wertige

Parameter gibt dabei den Namen der Spezifikation an, bezüglich der die Struktur als zyklisch angesehen werden soll. Dies ist nötig, da dieses Spezifikationslevel die Java Typdeklarationen charakterisiert, die zur Definition eines (möglicherweise zyklischen) Pfades herangezogen werden (s. auch Kap. 3.6.3). Dabei verwenden wir im Folgenden immer "docs2store" als Spezifikationslevel, da dies der Name der Spezifikation ist, in der alle Dokumenten-Klassen und Hilfsklassen deklariert sind. Näheres zur Spezifikationshierarchie findet sich in Anhang 13.1.

Wir schränken mit diesem Prädikat nun die Definitionen der Funktion java2doc auf azyklische Dokumente ein. Fälle, in denen die Eingabereferenz zyklisch ist, führen zur Rückgabe des Dokumentes $\perp$, welches als Default-Wert für Dokumente gesehen werden kann. Die Definition des Dokument Datentyps aus Kap. 6.1 muss also entsprechend um einen neuen Typ $\perp$ erweitert werden. Die entsprechende Definition für java2doc ist damit:

SPEZIFIKATION:

java2doc-cyclic:
cyclic("docs2store",r,st)
$\rightarrow$ java2doc(r, st) = $\perp$

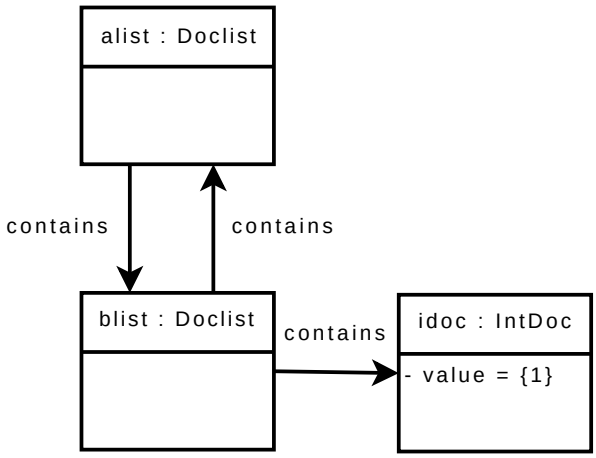


Abbildung 6.2: Ein Beispiel für eine zyklische Dokumentenstruktur

Ebenso kann eine Java Referenz den Wert `jvmref` haben. Auch hierfür haben wir in der Dokumentenebene keine Entsprechung und verwenden auch hier den $\perp$ Typ.

SPEZIFIKATION:

java2doc-null:
java2doc(jvmref, st) = $\perp$

Damit ist die einfachste Definition, in der ein echtes Dokument konstruiert wird, diejenige für den IntDoc Fall:

SPEZIFIKATION:

```
java2doc-intdoc:
      ¬ cyclic("docs2store",r,st)
   ∧ r ≠ jvmref
   ∧ st[r - .type] = IntDoc
→ java2doc(r, st) =
   IntDoc(bytes2int(getbytearray(st[r - .value].refval, st)))
```

Eine nicht zyklische Referenz `r`, die zudem einen Wert ungleich `jvmref` hat, und deren Typ `IntDoc` ist, wird in ein Dokument transformiert, indem man das byte Array in deren `.value` Feld in einen Integer transformiert (bytes2int(..)) und mit dem resultierenden Wert ein IntDoc bildet.

Wieder analog erfolgt die Behandlung des `SecretDoc` Typs:

SPEZIFIKATION:

```
java2doc-secretdoc:
      ¬ cyclic("docs2store",r,st)
   ∧ r ≠ jvmref
   ∧ st[r - .type] = SecretDoc
→ java2doc(r, st) =
   SecretDoc(bytes2int(getbytearray(st[r - .value].refval, st)))
```

Schließlich spezifizieren wir für `NonceDoc` und `KeyDoc` wegen der zusätzlichen `Nonce` und `Key` Typen jeweils zwei Axiome:

SPEZIFIKATION:

```
java2doc-noncedoc:
      ¬ cyclic("docs2store",r,st)
   ∧ r ≠ jvmref
   ∧ st[r - .type] = NonceDoc
→ java2doc(r, st) =
NonceDoc(java2doc(st[r - .nonce].refval, st))

java2doc-nonce:
      ¬ cyclic("docs2store",r,st)
```

```
    ∧ r ≠ jvmref
    ∧ st[r - .type] = Nonce
→ java2doc(r, st) =
    mknonce(bytes2int(getbytearray(st[r - .nonceval].refval, st)))

java2doc-keydoc:
      ¬ cyclic("docs2store",r,st)
    ∧ r ≠ jvmref
    ∧ st[r - .type] = KeyDoc
→ java2doc(r, st) =
    KeyDoc(java2doc(st[r - .key].refval, st))

java2doc-key:
      ¬ cyclic("docs2store",r,st)
    ∧ r ≠ jvmref
    ∧ st[r - .type] = Key
→ java2doc(r, st) =
    mkkey(bytes2int(getbytearray(st[r - .keyval].refval, st)))
```

Die jeweilige Definition für den Dokumententyp NonceDoc und KeyDoc stützt sich dabei auf die Definition für den dazugehörigen Inhaltstyp Nonce und Key ab.

Für den Listentyp `Doclist` müssen wir nun über das Array der Inhaltsdokumente im Speicher iterieren. Dazu stützen wir uns auf eine Bibliotheksfunktion getarray : reference × store → javavalues, die zu einer gegebenen Referenz alle im Array unter dieser Referenz enthaltenen Java Werte zurückliefert.

Damit kann zunächst eine einfache Iteration java2docs : javavalues × store → documentlist über solche Wertlisten definiert werden:

SPEZIFIKATION:

```
java2docs-empty:
java2docs([ ], st) = [ ]

java2docs-list:
java2docs(val + vals, st) = java2doc(val.refval, st) + java2docs(vals,st)
```

Damit können wir nun für die `Doclist` java2doc spezifizieren:

SPEZIFIKATION:

```
java2doc-doclist:
      ¬ cyclic("docs2store",r,st)
```

$$
\begin{aligned}
&\wedge\ r \neq \text{jvmref} \\
&\wedge\ st[r\text{ - .type}] = \text{Doclist} \\
\rightarrow\ &\text{java2doc}(r, st) = \\
&\text{Doclist}(\text{java2docs}(\text{getarray}(st[r\text{ - .docs}].\text{refval}, st)))
\end{aligned}
$$

Alle vorangegangenen Axiome beinhalten als Vorbedingung die Azyklizität der jeweiligen Pointerstruktur. Intuitiv sollte gelten, dass keine der durch Subtypen von `Document` aufgespannten Pointerstrukturen mit Ausnahme der `Doclist` zyklisch sein kann. Ein `IntDoc` beinhaltet schließlich lediglich ein `byte[]` und die darin enthaltenen primitiven Werte können keinen Zyklus mehr enthalten. Tatsächlich gilt auch:

THEOREM:

$$
\begin{aligned}
&\text{validref}(\text{"docs2store"}, r, st) \\
&\wedge\ r \neq \text{null} \\
&\wedge\ st[r\text{ - .type}] = \text{IntDoc} \\
\rightarrow\ &\neg\ \text{cyclic}(\text{"docs2store"}, r, st)
\end{aligned}
$$

Dies gilt allerdings nur unter der Voraussetzung, dass r eine *gültige* Java Referenz ist (gegeben durch validref("docs2store", r, st)). Gültig ist eine Referenz dabei intuitiv, wenn sie durch ein wohlgeformtes Java Programm erzeugt werden kann. Der Speicher selbst definiert keine Anforderungen an Typkorrektheit. So kann durchaus in einem Speicher z.B. ein `IntDoc` enthalten sein, das in seinem `.value` Feld ein Objekt vom Typ `Doclist` enthält. Das Prädikat validref("speclevel", r, st) schließt dabei solche Pointerstrukturen generisch aus (s. Kap. 3). Es fordert u.a. dass alle Referenzen, die von r auf gültigen Pfaden erreichbar sind, Typen haben, die in den aktuellen Typdeklarationen existieren. Die aktuellen Typdeklarationen werden dabei durch die Angabe des aktuellen Spezifikationslevels "speclevel" charakterisiert.

Wir haben aus Gründen einer einfachen Spezifikation in der Definition von java2doc die Eigenschaft validref nicht vorausgesetzt, aus Gründen einer systematischen Spezifikation die Eigenschaft ¬ cyclic dagegen schon.

6.7 Anwendung von java2doc im Refinement

Die Funktion java2doc dient im Refinement Framework nun für zwei Aufgaben:

Zum Einen übernimmt sie die Transformation einer Ausgabe des Java Programms auf der konkreten Ebene in ein abstraktes Dokument. Sie wird daher im ASM Makro FROMSTORE im allgemeinen Verfeinerungsframework in Kap. 5.4.2 verwendet. Analog zur Anwendung von doc2java während des Empfangens von Nachrichten und der Eigenschaft einer aufgerufenen `Document receive()` Methode, genau deren Ergebnis zurückzuliefern, wird java2doc

im Zusammenhang mit einer `send(Document)` Methode verwendet. Die `send` Methode übernimmt in der Realität die Zustellung einer Nachricht an einen Empfänger mittels der Kommunikationsinfrastruktur (z.B. Senden einer MMS in Cindy). Während der Verifikation wird diese Methode lediglich dazu verwendet, das gesendete Dokument an einer temporären Speicherstelle `.out` (s. auch Definition von FROMSTORE in Kap. 5.4.2) zwischenzuspeichern. Während FROMSTORE wird die Pointerstruktur dann im Anschluss in ein abstraktes Dokument verwandelt und die Verbindungs- und Kommunikationsinfrastruktur der PROSECCO Ebene mittels Update auf die inputs Funktion an den richtigen Empfangsport zugestellt.

Damit könnten wir im Idealfall für den Verfeinerungsbeweis für die `send` Methode folgende Eigenschaft benutzen:

SPEZIFIKATION:

send:
validref("docs2store", r, CommInterface, st)
$\wedge$ r $\neq$ jvmref
$\wedge$ st = st_0
$\rightarrow \langle$ st ; r.send(r_1) $\rangle$ (st = st_0[.out, refval(r_1)])

Die `send` Methode setzt also genau den übergebenen Pointer im Speicher an der temporären Speicherstelle `.out`. Bei mehreren Ausgabeports für die Implementierung bzw. der Angabe eines send-mode bzw. des receivers bei verbindungsloser Kommunikation wie bei Cindy muss obige Definition natürlich noch um diese Parameter im Aufruf der Methode send erweitert werden. Ebenso müssen die Parameter dann analog zu dem Setzen von `.out` temporär im Speicher bis zum Aufruf von FROMSTORE nach dem Java Programm gesetzt werden.

Neben dieser Benutzung für das Senden von Nachrichten definiert die java2doc Funktion auch die Extraktionsfunktion extract aus der Definition der Simulationsrelation und aus der Definition der Finalisierungsoperation aus dem allgemeinen Verfeinerungsframework. Sie stellt damit den Zusammenhang zwischen abstraktem und konkretem Zustand des verfeinerten Agenten her. Bei Cindy gilt z.B.

extract(cstore(agent))
$\equiv$ java2doc(cstore(agent)[.theinstance.tickets].refval, cstore(agent))

und damit insbesondere innerhalb der Definition der Simulationsrelation

java2doc(cstore(agent)[.theinstance.tickets].refval, cstore(agent)) = tickets(agent)

6.8 Charakterisierung gültiger Pointerstrukturen

Bei Betrachtung der möglichen Java `Document` Pointerstrukturen und der vorgestellten Axiome für doc2java und java2doc fällt nun auf, dass wir einige implizite Annahmen an den Aufbau der Strukturen im Speicher gemacht haben. In diesem Abschnitt wollen wir daher *Gültigkeit* von `Document` Pointerstrukturen definieren. Die hier verwendete Definition wird später in den eigentlichen Verfeinerungsbeweisen eine zentrale Rolle spielen, denn sie charakterisiert genau die Eingaben, mit denen unsere Protokollimplementierungen umgehen können müssen. Details hierzu folgen später in Kap. 8.

Wir verwenden dazu ein Prädikat validDoc : reference × store. Dieses Prädikat ist wiederum über die verschiedenen Dokumententypen spezifiziert. Wir spezifizieren zunächst allerdings allgemein für zyklische Pointerstrukturen und den `null` Pointer folgende Axiome:

SPEZIFIKATION:

validDoc-null:
validDoc(jvmref, st)

validDoc-cyclic:
r ≠ jvmref
∧ cyclic(r,st)
→ ¬ validDoc(r, st)

`jvmref` ist immer ein gültiges `Document` (es entspricht ⊥). Zyklische Strukturen sind dagegen niemals validDoc.

Instanzen der abstrakten Klasse `Document` können ebenso nicht mit doc2java konstruiert werden. Daher gilt:

SPEZIFIKATION:

validDoc-Document:
r ≠ jvmref
∧ ¬ cyclic(r,st)
∧ st[r - .type] = Document
→ ¬ validDoc(r, st)

Im Weiteren können nur ordentliche Subtypen von `Document` mit doc2java konstruiert werden. Es gilt also[8]:

[8]Das Prädikat type∃ (speclevel,ty) gibt an, ob ein Typ ty in den Typdeklarationen, die bis zur Spezifikationshierarchieebene "speclevel" deklariert wurden, existieren. Die Ebene "docs2store" ist dabei die Ebene, auf der die hier vorgestellten Funktionen und Prädikate definiert werden.

SPEZIFIKATION:

validDoc-no-Document:
$$
\begin{aligned}
&\quad r \neq \text{jvmref} \\
&\wedge \neg\, \text{cyclic}(r,st) \\
&\wedge (\neg\, st[r\text{ - .type}] \leq \text{Document} \vee \neg\, \text{type}\exists\ (\text{"docs2store"}, st[r\text{ - .type}]) \\
\rightarrow\ &\neg\, \text{validDoc}(r, st)
\end{aligned}
$$

Für die eigentlichen Dokumententypen müssen nun die oben genannten Eigenschaften erfüllt werden. So gilt z.B. für ein `IntDoc`:

SPEZIFIKATION:

validDoc-IntDoc:
$$
\begin{aligned}
&\quad r \neq \text{jvmref} \\
&\wedge \neg\, \text{cyclic}(r,st) \\
&\wedge\ st[r\text{ - .type}] = \text{IntDoc} \\
\rightarrow\ &(\quad \text{validDoc}(r, st) \\
&\leftrightarrow\quad st[r\text{ - .value}] \neq \text{refval}(\text{jvmref}) \\
&\qquad \wedge\ \text{validByteArray}(st[r\text{ - .value}].\text{refval}, st)
\end{aligned}
$$

Der `.value` Wert eines `IntDocs` darf nicht `jvmref` sein und das Array selbst muss in kürzestmöglicher Repräsentation vorliegen (d.h. keine unnötigen Vorzeichenbytes - dies wird durch validByteArray(...) beschrieben, s. auch Kap. 8.3.2).

Analog gilt für die anderen nicht rekursiven Typen:

SPEZIFIKATION:

validDoc-SecretDoc:
$$
\begin{aligned}
&\quad r \neq \text{jvmref} \\
&\wedge \neg\, \text{cyclic}(r,st) \\
&\wedge\ st[r\text{ - .type}] = \text{SecretDoc} \\
\rightarrow\ &(\quad \text{validDoc}(r, st) \\
&\leftrightarrow\quad st[r\text{ - .secret}] \neq \text{refval}(\text{jvmref}) \\
&\qquad \wedge\ \text{validByteArray}(st[r\text{ - .secret}].\text{refval}, st)
\end{aligned}
$$

validDoc-KeyDoc:
$$
\begin{aligned}
&\quad r \neq \text{jvmref} \\
&\wedge \neg\, \text{cyclic}(r,st) \\
&\wedge\ st[r\text{ - .type}] = \text{KeyDoc} \\
\rightarrow\ &(\quad \text{validDoc}(r, st)
\end{aligned}
$$

```
     ↔    st[r - .key] ≠ refval(jvmref)
          ∧ st[st[r - .key].refval - .keyval] ≠ refval(jvmref)
          ∧ validByteArray(st[st[r - .key].refval - .keyval].refval, st)

validDoc-NonceDoc:
       r ≠ jvmref
    ∧ ¬ cyclic(r,st)
    ∧ st[r - .type] = NonceDoc
→ (    validDoc(r, st)
     ↔    st[r - .nonce] ≠ refval(jvmref)
          ∧ st[st[r - .nonce].refval - .nonceval] ≠ refval(jvmref)
          ∧ validByteArray(st[st[r - .nonce].refval - .nonceval].refval, st)
```

Bei `KeyDoc` und `NonceDoc` muss noch, neben dem jeweiligen `byte[ ]` Wert, auch die diesen beinhaltende Instanz von `Nonce` oder `Key` ungleich `jvmref` sein.

Letztendlich müssen wir noch eine rekursive Definition für den `Doclist` Typ angeben. Dies geschieht analog zu den vorherigen rekursiven Definitionen durch Angabe einer Rekursionsfunktion für Listen von javavalues und anschließende Definition für den `Doclist` Typ mittels getarray:

SPEZIFIKATION:

```
validDocs-empty:
validDocs([ ], st)

validDocs-rec:
   validDocs(val + vals, st)
↔ is_referencevalue(val) ∧ validDoc(val.refval,st) ∧ validDocs(vals, st)

validDoc-Doclist:
       r ≠ jvmref
    ∧ ¬ cyclic(r,st)
    ∧ st[r - .type] = Doclist
→ (    validDoc(r, st)
     ↔    st[r - .docs] ≠ refval(jvmref)
          ∧ st[st[r - .docs].refval - .type] = Doclist
          ∧ validDocs(getarray(st[r - .docs].refval,st),st)
```

6.9 Wichtige Eigenschaften für Verfeinerungsbeweise

Die vorgestellten Definitionen für TOSTORE und FROMSTORE sowie die Eigenschaften für receive und send sind generischer Bestandteil des Verfeinerungskonzeptes für sämtliche Protokollimplementierungen. Es liegt daher nahe, bereits jetzt über diese Makros und Java-Methoden Eigenschaften auf höherem Abstraktionsniveau zu beweisen, damit auch diese generisch in allen Verfeinerungsbeweisen verwendet werden können. Insbesondere fällt dabei auf, dass eine Dualität zwischen TOSTORE und `receive` besteht: Letzteres liefert genau die konkrete Entsprechung der abstrakten Eingabe, die während TOSTORE in den Speicher eingefügt wurde. Die gleiche Dualität gilt auch zwischen FROMSTORE und `send`: Ersteres liefert die abstrakte Entsprechung der konkreten Ausgabe, die während `send` versendet wurde. Diese beiden Eigenschaften sind zentral für die Verfeinerungsbeweise und werden im Folgenden immer wieder verwendet werden. Wir wollen sie daher formal fassen:

6.9.1 TOSTORE und `receive`

Der Zusammenhang zwischen TOSTORE und `receive` ist formal:

THEOREM:

TOSTORE-receive:

$$
\begin{aligned}
& .\mathsf{mode} \in \mathsf{st} \wedge \mathsf{st}[.\mathsf{mode}] = \mathsf{noval} \\
& \wedge\ \mathsf{validrefnotnull}(\text{"CommInterface"}, \mathsf{r}, \mathsf{CommInterface}, \mathsf{st}) \\
& \wedge\ \mathsf{inputs}(\mathsf{agent})(\mathsf{i}) \neq [\,] \\
& \wedge\ \langle \mathsf{TOSTORE}(\mathsf{agent}, \mathsf{inputs}; \mathsf{cstore}) \rangle\ (\mathsf{cstore} = \mathsf{cstore}_1) \\
& \wedge\ \mathsf{st} = \mathsf{cstore}_1(\mathsf{agent}) \\
& \wedge\ \langle \mathsf{st};\ \mathsf{r}_1 \texttt{ = r.receive(i); } \rangle\ (\mathsf{st} = \mathsf{st}_0 \wedge \mathsf{r}_1 = \mathsf{r}_2) \\
\rightarrow\ & \mathsf{java2doc}(\mathsf{r}_2, \mathsf{st}_0) = \mathsf{inputs}(\mathsf{agent})(\mathsf{i}).\mathsf{first} \\
& \wedge\ \mathsf{st} \subseteq \mathsf{st}_0 \\
& \wedge\ \mathsf{validref}(\text{"docs2store"}, \mathsf{r}_2, \mathsf{Document}, \mathsf{st}_0) \\
& \wedge\ \mathsf{validDoc}(\mathsf{r}_2, \mathsf{st}_0)
\end{aligned}
$$

Wird TOSTORE aufgerufen, wenn Eingaben für den Agenten agent vorliegen (inputs(agent)(i) $\neq [\,]$), und entsteht daraus ein Speicher $\mathsf{st} = \mathsf{cstore}_1(\mathsf{agent})$ und wird zusätzlich `receive` für den Port i in diesem Kontext ausgeführt, so liefert diese Methode gerade die Pointerstruktur r_2 als Ergebnis, die gewandelt mittels java2doc der abstrakten Eingabe inputs(agent)(i).first entspricht. Zusätzlich ist die Ausgabe typkorrekt (validref) und erfüllt unsere Charakterisierung gültiger Dokumenten-Pointerstrukturen (validDoc). Der Beweis dieser Eigenschaft folgt direkt aus der Definition für TOSTORE und der Eigenschaft receive.

6.9.2 FROMSTORE und send

Ebenso gilt für FROMSTORE und send:

THEOREM:

FROMSTORE-send:

$$
\begin{aligned}
&\quad .\mathsf{mode} \in \mathsf{st} \wedge \mathsf{st}[.\mathsf{mode}] = \mathsf{noval} \\
&\wedge\ \mathsf{validrefnotnull}(\text{"CommInterface"}, \mathsf{r}, \mathsf{CommInterface}, \mathsf{st}) \\
&\wedge\ \mathsf{validref}(\text{"docs2store"}, \mathsf{r}_1, \mathsf{Document}, \mathsf{st}) \\
&\wedge\ \mathsf{validDoc}(\mathsf{r}_1, \mathsf{st}) \\
&\wedge\ \mathsf{st} = \mathsf{st}_0 \\
\rightarrow\ &\langle \mathsf{st};\ \texttt{r.send(}\mathsf{r}_1\texttt{, outport}_0\texttt{);}\ \rangle \\
&\quad \langle \mathsf{FROMSTORE}(\mathsf{agent}, \mathsf{st};\ \mathsf{cstore}, \mathsf{inputs}, \mathsf{outdoc}, \mathsf{outport})\ \rangle \\
&\quad (\quad \mathsf{outdoc} = \mathsf{java2doc}(\mathsf{r}_1, \mathsf{st}_0) \\
&\quad \wedge\ \mathsf{outport} = \mathsf{outport}_0)
\end{aligned}
$$

Wird send mit einer wohlgeformten (validDoc) Dokumenten-Pointerstruktur r_1 aufgerufen und danach im Kontext des resultierenden Speichers st das Makro FROMSTORE ausgeführt, so ist dessen Resultat outdoc gerade die abstrakte Entsprechung von r_1 (mittels java2doc). Der Port wurde ebenso richtig aus dem Speicher extrahiert.

Anmerkung: Wir benötigen für den Verfeinerungsbeweis keine weitere Eigenschaft analog zu FROMSTORE-send, die über nicht wohlgeformte Eingaben eine Aussage trifft. Dies begründet sich dadurch, dass eine korrekte Protokollimplementierung nur Ausgaben produzieren sollte, die auch validDoc erfüllen. Im Sinne einer robusten Implementierung werden wir dennoch später eine Implementierung für send angeben, die auch mit nicht wohlgeformten Eingaben umgehen kann.

Kapitel 7

Ergebnisse im Detail: Cindy

In theory, there is no difference between theory and practice;
in practice, there is.
Chuck Reid

Mit den Ausführungen der vorangegangenen Kapitel können wir nun die Verifikation der ersten Fallstudie, der Cindy-Anwendung für Mobiltelefone, vorstellen. Dieses Kapitel gibt einige Details zur Implementierung und den entsprechenden Beweisen dieser Anwendung. Die in diesem Kapitel beschriebenen Arbeiten wurden in [66] publiziert.

7.1 Spezifikation und Implementierung

Nun instantiieren wir die generischen Beweisverpflichtungen, die bereits in Kap. 5.4.5 erläutert wurden, mit den Definitionen für die Cindy Anwendung. In Kap. 5.1 wurde bereits ein Überblick über die abstrakte Spezifikation der Cindy Anwendung gegeben und in Kap. 5.3.1 wurde auch das Grundgerüst der Implementierung vorgestellt.

Die vollständigen Spezifikationen und Beweise finden sich im Projekt *cindy-refinement* unter [49]. Wir können hier lediglich einen Überblick über die Spezifikationen und Beweise sowie die Beweisstrategie geben. Wir wollen dazu insbesondere die Zusammenhänge zwischen Spezifikation und Implementierung beschreiben. Der interessante Teil ist dabei natürlich die Umsetzung der Mobiltelefone auf beiden Ebenen. Insbesondere haben wir in Kap. 5.3.1 noch nicht die wirkliche Implementierung der Protokollfunktionalität beschrieben. Insbesondere haben wir die Protokollroutinen `phoneStep` und `userStep` für die Verarbeitung von Eingaben an den beiden Eingabeports *MMS* und *Benutzerschnittstelle* noch nicht angegeben.

Für die *MMS* Eingabeschnittstelle ist lediglich eine Funktionalität möglich, das Speichern von Tickets (die Weitergabe von Tickets benutzt das gleiche Nachrichtenformat wie das Speichern neuer Tickets und benötigt daher keine eigene Implementierung):

```
1 public class Protocol {
2
```

```
3  private Doclist tickets = new Doclist();
4
5  public void step(){
6    ...
7    else if(comm.available(PHONEPORT)){
8     Document inmsg = comm.receive(PHONEPORT);
9     if(inmsg != null) phoneStep(inmsg);
10   }
11  }
12
13 private void phoneStep(Document inmsg) {
14   Document originator = inmsg.getPart(1);
15   inmsg = inmsg.getPart(2);
16   Doclist ticket = getTicket(inmsg, originator);
17   if(ticket != null && tickets.len() < MAXTICKETLEN){
18       tickets = tickets.attach(ticket);
19   }
20 }
21 }
```

Alle Nachrichten in der Cindy Anwendung haben, wie auch schon in Kap. 5.1 die Struktur Doclist(IntDoc(originator) + *Nutzdaten*). Wir selektieren daher zunächst den Absender (`inmsg.getPart(1)`) und die eigentlichen Nutzdaten (`(inmsg.getPart(2)`). Die anschließend benutzte Funktion `getTicket` prüft nun die Nutzdaten auf Wohlgeformtheit und extrahiert gleichzeitig das übermittelte Ticket. Wir erwarten dabei eine Nachricht der Form *loadTicket* oder *passOn* aus Tabelle 5.1.

Wenn wir uns zurück an die entsprechende abstrakte Spezifikation erinnern (s. Kap. 5.1, S. 46), so sehen wir, dass dort ein Prädikat is-load(doc) verwendet wurde, um die Wohlgeformtheit zu prüfen. Die abstrakte Spezifikation dazu sieht noch recht einfach aus und kann recht leicht mit der gewünschten Nachrichtenstruktur in Zusammenhang gebracht werden:

SPEZIFIKATION:

is-load-def:
is-load(doc)
↔ is-doclist(indoc)
∧ is-intdoc(indoc.list.first)
∧ indoc.inst = loadTicket
∧ is-doclist(indoc.list)
∧ # indoc.list.list = 2
∧ is-intdoc(get-part(indoc.list,1))
∧ is-noncedoc(get-part(indoc.list,2))

Dagegen wird die Implementierung komplizierter. Die Implementierung muss alle Teile der

Nachricht durchgehen und entsprechend prüfen, ob alles in Ordnung ist. Die entsprechenden abstrakten Gegenstücke zum jeweiligen Java Quellcode sind hierbei als Kommentare mit angegeben:

```
1  private Doclist getTicket(Document indoc, Document originator) {
2   // is-doclist(indoc) and is-intdoc(indoc.list.first)
3   if(indoc != null && indoc.is_comdoc()){
4    byte[] ins = indoc.getPart(1).getValue();
5    // indoc.inst ==  loadTicket
6    if(ins.length == 1 && ins[0] == LOADTICKET)
7    {
8     indoc = indoc.getPart(2);
9     // is-doclist(indoc.data) and # indoc.list.list = 2
10    if(indoc != null && indoc.len() == 2){
11     Document indoc1 = indoc.getPart(1);
12     Document indoc2 = indoc.getPart(2);
13     // is-intdoc(get-part(indoc.list,1)) and
14     // is-noncedoc(get-part(indoc.list,2))
15     if(indoc1 != null && indoc1.is_intdoc() &&
16        indoc2 != null && indoc2.is_noncedoc()){
17         return new Doclist(originator, indoc);
18     }
19    }
20   }
21  }
22  return null;
23 }
```

Ein `byte[ ]` muss in der Implementierung z.B. neben dem richtigen Wert (hier LOADTICKET) auch auf die richtige Länge geprüft werden (hier 1). Ebenso könnten Nachrichtenteile den Wert `null` haben, was abgefangen werden muss, um `NullPointerExceptions` abzufangen. Ist alles in Ordnung, so liefert die `getTicket` Methode schließlich eine Dokumentenliste zurück, die die Struktur eines Tickets hat (erster Teil Absender, zweiter Teil Datamatrix-Code als NonceDoc). Ist die Nachricht nicht wohlgeformt, wird `null` zurückgegeben.

Zurück in der Methode `phoneStep` wird nun geprüft, ob der Rückgabewert entsprechend ungleich `null` ist. Ist dies der Fall, wird das neue Ticket an die Liste der bisherigen Tickets `tickets` angehängt und die Methode ist beendet. Dies geschieht allerdings nur dann, wenn die Ticketliste noch nicht ihre maximale Länge erreicht hat (`MAXTICKETLENGTH`), analog zur Spezifikation in Kap. 5.1.

Entsprechender Code wurde auch für die diversen anderen Protokollschritte der Cindy Applikation implementiert. Er findet sich komplett in Anhang 13.1. Alle Protokollschritte folgen dabei dem hier bereits verwendeten Prinzip, zuerst die Eingabe auf Wohlgeformtheit zu prüfen und anschließend entsprechend den Zustand zu verändern bzw. eine Ausgabe zu versenden.

7.2 Beweisstrategie

Wir wollen nun die Korrektheit des angegebenen Quellcodes nachweisen. Dazu instantiieren wir die generische Verfeinerungstheorie des Data Refinements mit den Definitionen für Cindy. Dementsprechend erhalten wir nun eine Beweisverpflichtung entsprechend den Ausführungen in Kap. 5.4.5. Es ändert sich hierbei nichts wesentliches, lediglich der abstrakte Zustand wird nun durch die jeweiligen Zustandsfunktionen wie z.B. die Liste der gespeicherten Tickets ersetzt. Wir geben hier aus Einfachheitsgründen nicht den vollen Zustand an sondern beschränken uns auf diese Liste. Die Hauptbeweisverpflichtung für die Korrektheit eines Schritts der Implementierung ist damit (unter der Annahme, dass die Terminierung der konkreten Operationen und des Java Methodenaufrufs `step()` bereits anderweitig gezeigt wurde):

Correctness-CELLPHONE:

$$
\begin{array}{l}
\mathsf{R}(\mathsf{ainputs}, \mathsf{aconnections}, \mathsf{aattackerknown}, \mathsf{tickets}, \ldots, \\
\quad \mathsf{cinputs}, \mathsf{cconnections}, \mathsf{cattackerknown}, \ldots, \mathsf{cstore}), \\
\langle \mathsf{TOSTORE}(\mathsf{agent}, \mathsf{cinputs}; \mathsf{cstore}) \rangle\ \mathsf{cstore} = \mathsf{cstore}_0, \\
\mathsf{st}_0 = \mathsf{cstore}(\mathsf{agent}), \\
\langle \mathsf{st}_0 \ ; \texttt{Protocol.instance.step();} \rangle\ \mathsf{st}_0 = \mathsf{st}, \\
\langle \mathsf{FROMSTORE}(\mathsf{agent}, \mathsf{st}; \mathsf{cstore}_0, \mathsf{cinputs}, \mathsf{coutdoc}, \mathsf{coutport}) \rangle \\
\quad (\ \ \mathsf{cstore}_0 = \mathsf{cstore}_1 \wedge \mathsf{cinputs} = \mathsf{cinputs}_1 \\
\quad\ \ \wedge \mathsf{coutdoc} = \mathsf{coutdoc}_0 \wedge \mathsf{coutport} = \mathsf{coutport}_0) \\
\langle \textbf{if}\ \mathsf{coutdoc}_0 \neq \bot\ \textbf{then}\ \mathsf{SEND}(\mathsf{coutdoc}_0, \mathsf{coutport}_0; \mathsf{cinputs}_1, \mathsf{cattackerknown}, \ldots) \rangle \\
\quad (\mathsf{cinputs}_1 = \mathsf{cinputs}_2 \wedge \mathsf{cattackerknown} = \mathsf{cattackerknown}_2 \wedge \ldots) \\
\ldots \\
\vdash \\
\langle \mathsf{CELLPHONE}(\mathsf{agent}, \mathsf{ainputs}, \mathsf{aconnections}, \mathsf{aattackerknown}, \mathsf{tickets}, \ldots) \rangle \\
\qquad \mathsf{R}(\mathsf{ainputs}, \mathsf{aconnections}, \mathsf{aattackerknown}, \mathsf{tickets}, \ldots \\
\qquad\quad \mathsf{cinputs}_2, \mathsf{cconnections}, \mathsf{cattackerknown}_2, \ldots, \mathsf{cstore}_1)
\end{array}
$$

Beweisidee:

Die grundsätzliche Beweisidee bei den Verfeinerungsbeweisen für konkrete Implementierungen ist die symbolische Ausführung der beiden Programme (Java Implementierung und ASM Spezifikation). Zusätzlich müssen richtig gewählte Lemmata für beide Ebenen für eine hinreichende Modularisierung und Komplexitätsreduktion sorgen. Es hat sich dabei als geschickte Strategie herausgestellt, die Methoden der Java Implementierung durch Lemmata auf die ihnen zugrunde liegenden Operationen auf abstrakten Datentypen zurückzuführen. Ein erstes Beispiel für diese Strategie ist die am Ende von Kap. 6.9 eingeführte Eigenschaft *TOSTORE-receive*, die `receive` eliminiert, indem sie für dessen Rückgabewert die abstrakte Entsprechung liefert (nämlich genau das abstrakte Eingabedokument). Um überhaupt erst zur Anwendung von *TOSTORE-receive* kommen zu können, führen wir zunächst das Programm `step` in obiger Beweisverpflichtung symbolisch aus. Nachdem wir die `available` Methode mit einem entsprechenden Lemma behandelt haben (`available` gibt `true` zurück, falls eine Eingabe vorliegt und verändert den Speicher nicht), haben wir schließlich ein Ziel der Form:

```
R(ainputs, aconnections, aattackerknown, tickets, ... ,
   cinputs, cconnections, cattackerknown, ... , cstore),
cinputs(agent)(PHONEPORT) ≠ [ ],
⟨TOSTORE(agent, cinputs; cstore) ⟩ cstore = cstore0,
st0 = cstore(agent),
⟨st0 ; Document d = comm.receive(PHONEPORT);
     phoneStep(d);  ⟩ st0 = st,
...
⊢ ...
```

Hier können wir nun *TOSTORE-receive* verwenden und so `receive` eliminieren. Wir erhalten daraus, dass für die Java Variable `d` in obiger Sequenz gilt: java2doc(d, st) = inputs(agent)(PHONEPORT).first. Zusätzlich erhalten wir die Eigenschaft validDoc für die Eingabe.

Es verbleibt danach zu zeigen, dass sich `phoneStep` mit einer solchen Eingabe so verhält wie die entsprechende abstrakte Spezifikation. Z.B. bedeutet dies, dass sich die Liste der Tickets entsprechend der Spezifikation ändert:

```
java2doc(d, st) = inputs(agent)(PHONEPORT).first,
validDoc(d,st),
⟨CELLPHONE(agent, ainputs, aconnections, aattackerknown, tickets, ...) ⟩
      (tickets = tickets0 ∧...)
...
⊢ ⟨st0 ; phoneStep(d); ⟩
   java2doc(st[Protocol.theinstance.tickets].refval, st0) = tickets0(agent)
```

Auch hier führen wir unsere Beweisstrategie durch symbolische Ausführung weiter. Wir reduzieren schließlich das Java Programm auf prädikatenlogische Eigenschaften, die auch entsprechend von der abstrakten Spezifikation erfüllt werden (hier etwa das korrekte Verändern der Ticketliste). Somit kann das Java Programm vollständig eliminiert werden und schließlich muss nur noch die entsprechende Simulationsrelation gezeigt werden, welche gerade genau diese Eigenschaften (z.B. gleiche Ticketlisten) fordert, s. dazu insbesondere die Beziehung am Ende von Kap. 6.7. Damit kann der Beweis dann geschlossen werden.

Dabei haben wir in dieser Darstellung stark vereinfacht: Wir benötigen für die fehlerfreie Ausführung der Programme für eine Argumentation nach obigem Muster eine starke Invariante - sowohl auf abstrakter Ebene als auch auf konkreter Ebene. So ist es z.B. wesentlich, dass der Java Speicher an den verwendeten Speicherstellen nur typkorrekte Inhalte hat. Darüber hinaus muss z.B. die Liste der derzeit gespeicherten Tickets wohlgeformt im Sinne der abstrakten Struktur von Tickets sein (also Listen von Paaren aus Absender und Datamatrixcode). Um einen Eindruck von solchen Eigenschaften zu vermitteln geben wir im Folgenden die entspre-

chende Definition für eine wohlgeformte Ticketliste im Speicher an. Es ist eine relativ hohe Anzahl an Einzeleigenschaften nötig, um den Aufbau einer solchen Pointerstruktur korrekt zu charakterisieren:

SPEZIFIKATION:

is-ticketlist-def:
is_ticketlist(r, st)
$\leftrightarrow$ r $\neq$ jvmref
$\wedge$ st[r − .type] .type = Doclist
$\wedge$ st[r − .docs] .refval $\neq$ jvmref
$\wedge$ ($\forall$ i.
0 $\leq$ i $\wedge$ i < st[r.docs.length] .intval
$\rightarrow$ is_MMS_message(st[r.docs[i]] .refval, st)
$\wedge$ is_ticket(st[r.docs[i].docs[1]].refval,st))

is-MMS-message:
is_MMS_message(r, st)
$\leftrightarrow$ r $\neq$ jvmref
$\wedge$ st[r.type] = Doclist
$\wedge$ st[r.docs] .refval $\neq$ jvmref
$\wedge$ st[r.docs.length].intval = 2
$\wedge$ st[r.docs[0]].refval $\neq$ jvmref
$\wedge$ st[r.docs[0].type] = IntDoc

is-ticket:
is_ticket(r, st)
$\leftrightarrow$ r $\neq$ jvmref
$\wedge$ st[r.type] = Doclist
$\wedge$ st[r.docs] .refval $\neq$ jvmref
$\wedge$ st[r.docs.length].intval = 2
$\wedge$ st[r.docs[0]].refval $\neq$ jvmref
$\wedge$ st[r.docs[1]].refval $\neq$ jvmref
$\wedge$ st[r.docs[0].type] = IntDoc
$\wedge$ st[r.docs[1].type] = NonceDoc

Prädikate wie diese müssen aufgestellt werden und in die Simulationsrelation mit aufgenommen werden. Das korrekte Aufschreiben dieser Invarianten ist damit auch eine der Hauptschwierigkeiten beim Beweis der Verfeinerungskorrektheit.

Ein weiterer Punkt, der in obiger Darstellung vernachlässigt wurde, ist die Formulierung von Lemmata für die diversen Hilfsmethoden in der Implementierung. Die Methode `getTicket` aus dem Quellcode oben ist ein gutes Beispiel. Die Methode soll im wesentlichen erkennen, ob die (ins Abstrakte transformierte) Eingabe das abstrakte is-load Prädikat erfüllt. Die Beweisstrategie muss also sein, konkrete Entsprechungen für diese Eigenschaften zu finden und

damit Lemmata zu formulieren, die `getTicket` charakterisieren. Entsprechende Eigenschaften finden sich in den Beweisen in der Spezifikation *cindy-refinementfuns* im Projekt *cindy-refinement* unter [49]. Deren korrekte Formulierung gehört auch zu den Schwierigkeiten bei den hier gezeigten Beweisen.

Insbesondere die Eigenschaften, die aus validDoc folgern, bilden für die Implementierung und den Beweis bereits eine gute Hilfestellung. Betrachtet man z.B. obige Implementierung der Methode getTicket, so fällt z.B. auf, dass die Eingabe nicht vollständig auf Wohlgeformtheit geprüft werden muss, sondern bestimmte Eigenschaften bereits aus validDoc folgern. So muss z.B. bei einem `IntDoc` nicht geprüft werden, ob der `byte[ ]` Inhalt eventuell den `null` Wert hat. Wir schreiben einfach:

```
1  ...
2  byte[] ins = indoc.getPart(1).getValue();
3  if(ins.length == 1 && ins[0] == LOADTICKET)
4  ...
```

Die Tatsache, dass `ins != null` folgt bereits aus der Eigenschaft validDoc(indoc,st). Damit hilft die Charakterisierung wohlgeformter Dokumente mittels validDoc bereits hier, die Implementierungen einfacher und verständlicher zu gestalten.

7.3 Erfahrungen

Die reine Implementierung der Cindy Protokollfunktionalität in der Klasse `Protocol` umfasst ca. 200 Zeilen Quellcode. In dieser Arbeit wurde die Fallstudie mit allen Protokollschritten vollständig verifiziert.

Die algorithmische Komplexität der Cindy Anwendung ist dabei weder bezogen auf die Spezifikation noch auf die Implementierung recht hoch. Dennoch stellt die Fallstudie einen sinnvollen und realistischen Testfall für die Verfeinerungsmethodik dar, da die Hauptschwierigkeit bei solchen Beweisen, wie sie eben beschrieben wurden, nicht in der Schwierigkeit oder Unverständlichkeit der Programme liegt (es müssen z.B. so gut wie nie komplexe Schleifeninvarianten innerhalb von Security Protokollimplementierungen gefunden werden), sondern vielmehr in der richtigen Modularisierung und der korrekten Invariantenfindung sowie in der korrekten und hinreichenden Findung von Eigenschaften für verwendete Hilfsmethoden. Deutlich wird dies z.B. in der Definition der Invariante, die mit ca. 100 Spezifikationszeilen schon recht umfangreich und komplex wird. Teile davon wurden bereits oben gezeigt. Hier ist die Fehleranfälligkeit recht hoch. Es stellt sich erwartungsgemäß heraus, dass die Verifikation eines realen Java Programms auf Grund der Komplexität der verwendeten Pointerstrukturen und der vielen Zusatzschritte im Programm, die durch Verwendung selbiger entstehen, wesentlich komplexer ist, als die Verifikation von Eigenschaften auf der (wenn auch funktionsgleichen) abstrakten ASM Spezifikation. Die relativ hohe Komplexität von Invarianten, die beim Implementieren eines solchen Programms lediglich sehr implizit in der menschlichen Vorstellung gegeben sind, erschwert dies nochmals. Darüber hinaus entstehen natürlich in einer Java Implementierung wesentlich mehr Programmierfehler als in einer abstrakten Spezifikation. Daher wird auch eine relativ hohe Zahl von Iterationen nötig, bis eine korrekte Implementierung samt korrekter und passender Invariante gefunden ist.

Die Verifikation der Fallstudie bedurfte letztendlich der Formulierung von 337 Lemmata und

49 Axiomen (jeweils ohne benötigte Bibliotheken wie z.B. PROSECCO oder die Basisbibliothek). Insgesamt wurden dazu 10367 Beweisschritte benötigt, von denen 6007 automatisch durchgeführt wurden. Damit erreichen wir einen Automatisierungsgrad von 60 Prozent. Der Automatisierungsgrad erhöht sich dabei mit jeder weiteren Fallstudie, da viele Eigenschaften, wie z.B. die aus validDoc folgerbaren Eigenschaften oder Eigenschaften zur Typkorrektheit, generisch für alle weiteren Fallstudien wiederverwendbar werden. Das selbe gilt für die Bibliothek zur Formalisierung und Verifikation von Java Programmen im Allgemeinen. Insbesondere Aussagen zur Typkorrektheit oder Aussagen über den Aufbau von Pointerstrukturen (z.B. Zyklenfreiheit, Existenz von Pfaden, usw.) sind generisch auch für andere Fallstudien verwendbar und zum größten Teil auch erst schrittweise während der Verifikation der in dieser Arbeit vorgestellten Fallstudien entstanden.

Der Gesamtaufwand zur reinen Verifikation der Anwendungslogik von Cindy umfasste etwa 2 Personenmonate durch entsprechend geschultes Personal. Damit zeigt sich, dass der hier vorgestellte Ansatz auch durchaus in der Praxis anwendbar ist und zeitgleich die höchst möglichen Garantien für eine sichere und korrekte Implementierung erlaubt.

7.4 Besonderheiten der Cindy Anwendung

Die Haupteigenschaft, die aus der Korrektheit der Verfeinerungsbeweisverpflichtungen für die Cindy Applikation folgt, ist die Tatsache, dass die Implementierung zu jedem Zeitpunkt und bei jeder möglichen Eingabe die selbe Liste von Tickets speichert, wie die abstrakte Spezifikation bei entsprechenden Eingaben. Die Implementierung folgt also in diesem Sinne ihrer Spezifikation. Dazu ist keine weitere Argumentation oder Beweis mehr nötig, dies folgt direkt aus der Verfeinerungstheorie.

Darüber hinaus sind wir allerdings auch an der Gültigkeit von *Sicherheitseigenschaften* der Applikation in der Implementierung interessiert. Intuitiv gilt hier natürlich, dass das Programm (bei selbem Eingabe-/Ausgabe- und Zustandsübergangsverhalten) auch die gleichen Sicherheitseigenschaften erfüllt, wie dies in der Spezifikation der Fall war. Es stellt sich allerdings die Frage, ob dies auch formal zu zeigen ist. Dazu müssen wir kurz die gewünschten Eigenschaften sowie einige Spezialitäten des PROSECCO Ansatzes erläutern.

Cindy verwendet keine Kryptographie. Daher gehören klassische Sicherheitseigenschaften, wie z.B. Authentisierung, nicht zu den Eigenschaften, die wir hier betrachten wollen. Statt dessen stehen andere Eigenschaften im Vordergrund der Betrachtung. Auf abstrakter Ebene wurden z.B. Eigenschaften wie die Folgenden für Cindy nachgewiesen (s. dazu auch [63]):

- Alle Tickets, die vom Kino ausgegeben wurden, werden auch vom Kino akzeptiert
- Der Kunde bezahlt nicht für mehr als für das, was er wirklich bestellt hat
- Wenn der Kunde ein Ticket nicht weitergibt, wird dieses am Eingang zum Kino auch akzeptiert werden

Wichtig ist nun, wie diese Eigenschaften formuliert und bewiesen wurden. Dabei liegt diesen Beweisen die Arbeit von Haneberg [71] zu Grunde, die den PROSECCO Ansatz einführt. Dort werden zwei grundsätzliche Ansätze beschrieben, wie Spezifikationen zu Sicherheitsprotokollen aufgebaut sein können: mit Berücksichtigung aller vergangenen Ereignisse im Zustand der

ASM oder ohne eine solche. Formaler beschrieben bedeutet dies, dass PROSECCO die Speicherung des bisherigen *Traces* im Zustand der ASM optional erlaubt. Ein Trace ist dabei einfach eine Liste der Zustände (bzw. dem Kreuzprodukt aller Zustandsfunktionen) der ASM, die durch die bisherige Ausführung in der Vergangenheit erreicht wurden. Haneberg beschreibt allerdings auch, dass die ASM und auch die Beweise durch Einführung eines Traces komplizierter werden. Daher wurde in der Verifikation der abstrakten Cindy Spezifikation, zu der auch obige Eigenschaften gehören und die nicht vom Autor dieser Arbeit durchgeführt wurde, darauf verzichtet, einen expliziten Trace einzuführen. Statt dessen wird der Zustand der ASM um einige weitere Zustandsfunktionen erweitert, die lediglich protokollierende Funktion haben und mittels denen dann obige Eigenschaften ausgedrückt werden können. Z.B. wird eine Liste accepted : Documentlist eingeführt, die in der ASM immer beim erfolgreichen Vorzeigen eines Tickets beim Kino um genau dieses Ticket verlängert wird. Ebenso wird eine Liste booked : agent → Documentlist eingeführt, die beim Bestellen eines Tickets mit dem Mobiltelefon um das aktuell bestellte Ticket erweitert wird. Diese Zustandsfunktionen beeinflussen den Kontrollfluss innerhalb der Spezifikationen nicht, d.h. sie werden nur mit schreibendem Zugriff verwendet, niemals mit lesendem. Sie sind somit lediglich eine alternative Art und Weise, eine Zustands-History (andernfalls in Form eines Traces vorliegend) zu repräsentieren. Die oben genannten Sicherheitseigenschaften werden nun als Invarianten für die Hauptregel der ASM mittels der genannten Funktionen spezifiziert. Die booked Liste wird z.B. für die dritte Eigenschaft benötigt, die accepted Liste für die erste. Vereinfachend beschrieben werden also Aussagen der Form

$$\langle\!|\mathsf{CINDY}_{abstract}\#(;\mathsf{inputs}, \mathsf{connections}, \ldots, \mathsf{tickets}, \mathsf{booked}, \mathsf{accepted}, \ldots) |\!\rangle$$
$$\mathsf{doc} \in \mathsf{booked}(\mathsf{agent}) \rightarrow \ldots$$

anhand der abstrakten Spezifikation bewiesen.

Nun haben wir auch eine Implementierung für unsere Spezifikation angegeben. Es stellt sich nun die Frage, wie dort mit diesen (neuen) Teilen des abstrakten Zustands umgegangen werden soll. Betrachten wir als Beispiel die booked Liste. Wir können in der realen Implementierung nicht eine (potentiell beliebig große) Liste aller jemals bestellten Tickets eine Mobiltelefons speichern. Allein schon der Speicherplatz und Effizienzgründe verbieten das natürlich. Daher können wir in der Implementierung diese Zustandsfunktionen nicht darstellen. Folglich kommen diese Zustandsfunktionen in der derzeitigen konkreten ASM-Ebene (mit eingebettetem Java Code) nicht vor.

Dies bedeutet allerdings auch, dass wir die (speziellen) Sicherheitseigenschaften, wie sie oben formuliert wurden, nicht im Rahmen einer Verfeinerung für die konkrete Ebene formal nachweisen können. Wir können die Eigenschaften für die konkrete Ebene nicht einmal *formulieren*. *Intuitiv* gelten die Eigenschaften natürlich dennoch auch in der Implementierung, da wir lediglich Funktionen weggelassen haben, die nicht für die Funktionalität der Anwendung, sondern lediglich für die Ausdrückbarkeit der Eigenschaften eingeführt wurden. Es gilt sogar (trivial), dass die abstrakte Spezifikation, wenn man die History-Funktionen herauslässt, absolut verhaltensäquivalent zur bisherigen Spezifikation ist. Der Gewinn weiteren Verifikationsaufwands an dieser Stelle ist daher vernachlässigbar.

Tatsächlich ist noch eine andere Art und Weise denkbar, die Eigenschaften für Cindy in der abstrakten Ebene zu formulieren und zu beweisen: die Verwendung eines expliziten Traces. Es müsste dann bei der Formulierung obiger Sicherheitseigenschaften eine Spezifikation auf dem Trace angegeben werden, die aus den jeweiligen aufeinanderfolgenden Zustandsübergängen im Trace die selbe Information herausrechnet, wie sie auch Funktionen wie booked geben. Für diese Funktion wäre etwa das Auftreten einer neuen Bestellnachricht für ein Ticket T in der Input-Queue des Kinos gesendet von Absender A die Entsprechung für $\mathsf{T} \in \mathsf{booked(A)}$. Dieser Trace könnte dann unverändert auch in der konkreten Ebene verwendet werden. Die Simulationsrelation würde dann die bereits vorgestellte extract Funktion für den Speicher verwenden, um den Zustand der Mobiltelefone aus den konkreten Speichern zu extrahieren und zu fordern, dass diese dem jeweiligen abstrakten Traceinhalt entsprechen. Der restliche konkrete Traceinhalt (also die jeweiligen Zustände der anderen Agenten) wären dann einfach genauso wie im abstrakten spezifiziert. Die Finalisierungsoperation müsste dann den Trace lediglich noch in den Globalzustand extrahieren. Aus dem Verfeinerungsbegriff des Data Refinement würde dann Trace-Gleichheit in abstrakter und konkreter Ebene folgen, womit auch für die Implementierung die Sicherheitseigenschaften nachgewiesen sind.

Kapitel 8

Erweiterung des Verfeinerungskonzeptes

I have not failed. I've just found 10.000 ways that won't work.
Thomas A. Edison

Kapitel 6 hat Wandlungsfunktionen von abstrakten Datentypen in Java Klassen und umgekehrt vorgestellt. In diesem Kapitel betrachten wir nun die Fragestellung, ob eine direkte Verwendung dieser Wandlungsfunktionen adäquat für eine Verfeinerung zur Code-Ebene ist. Dabei werden wir motivieren, dass ein Angreifer in der Realität mehr (und auch andere) Möglichkeiten besitzt, Eingaben an ein Programm zu schicken, als dies in der abstrakten Spezifikationsebene der Fall ist. Wir werden daher in diesem Kapitel das Verfeinerungskonzept entsprechend anpassen. Die in diesem Kapitel beschriebenen Arbeiten wurden in [62] publiziert.

8.1 Diskussion

Bei der Verwendung der Funktionen java2doc und doc2java aus dem vorletzten Kapitel zur Wandlung von Datentypen während der Verfeinerung ist wesentlich, ob

- eine Wandlung der abstrakten Eingabe in eine Pointerstruktur während TOSTORE mittels doc2java sowie
- eine Wandlung der konkreten Ausgabe in eine Pointerstruktur während FROMSTORE mittels java2doc sowie
- eine Definition einer Simulationsrelation zwischen abstrakten und konkreten agentenlokalen Zuständen mittels java2doc

die gewünschte Korrektheitsaussage für unsere realen Implementierungen (insbesondere auch die Gültigkeit von Sicherheitseigenschaften) negativ beeinflusst.

Insbesondere stellt sich die Frage, ob eine Implementierung zum Verschicken und Empfangen von Instanzen des `Document` Typs einfach so wie angegeben in der Verfeinerung angenommen werden kann. Schließlich verwenden wir einfach eine Eigenschaft in Beweis der Verfeinerungskorrektheit, die besagt, dass unsere Implementierung der `receive` Methode genau die Eingabe (konvertiert mittels doc2java) zurückliefert, welche in der entsprechenden abstrakten Eingabequeue des Agenten vorhanden war. Ist dies für in der Realität auftretende Systeme und mögliche Angriffe wirklich ausreichend?

Tatsächlich scheint diese Herangehensweise auf den ersten Blick eine viel zu starke Einschränkung während der Verifikation zu sein und lässt den Korrektheitsbegriff, der aus einer korrekten Verfeinerung folgt, sehr schwach erscheinen. *Schließlich betrachten wir so auf den ersten Blick gar nicht diejenigen Eingaben für das konkrete Java Programm, die nicht Ergebnis von* doc2java *sein können.*

Zur Illustration und Wiederholung: Z.B. eine zyklische Pointerstruktur, wie in Abb. 6.2 gezeigt, kann nie Ergebnis der Operation doc2java sein, schließlich verwendet diese Funktion für jeden Term des Eingabedokumentes *neue* Referenzen. Tatsächlich gilt auch:

THEOREM:

$$\mathsf{doc2java}(\mathsf{doc}, \mathsf{st}) = \mathsf{r} \times \mathsf{st}_0 \rightarrow \neg\ \mathsf{cyclic}(\mathsf{r}, \mathsf{st}_0)$$

Woher sollen wir nun wissen, dass unsere reale Implementierung der Kommunikationsinfrastruktur nicht doch möglicherweise bei bestimmten realen Eingaben zyklische (oder aus anderen Gründen nicht wohlgeformte) Pointerstrukturen durch `receive()` zurückliefert, anstatt sich auf die Ergebnisse von doc2java zu beschränken? Womit ist dann also eine Verifikation mit einer TOSTORE Spezifikation, die doc2java verwendet und einer receive Axiomatisierung, die lediglich den dort eingefügten Wert zurückliefert, noch von irgendeiner Aussagekraft für eine reale Umgebung, in der ein Angreifer eventuell ganz andere Möglichkeiten besitzt? Anders formuliert ergibt sich folgendes Problem: Wir betrachten bisher nicht *alle* in der Realität möglichen Eingaben für die Protokollimplementierung. Wie kann eine solche Verifikation noch hinreichend für eine sinnige Korrektheitsaussage für die Implementierung sein?

Wir beantworten diese Frage folgendermaßen: Die ausschließliche Betrachtung der Eingabedokumente, die mittels doc2java konstruiert werden können, ist hinreichend für eine sinnvolle Korrektheitsaussage für eine Implementierung mittels des vorgestellten Verfeinerungsansatzes, wenn eine Äquivalenz der bisherigen konkreten Ebene zu einer weiteren Spezifikationsebene bewiesen werden kann, in der die Ein- und Ausgaben an das Programm nicht länger nur die Ergebnisse von doc2java sein können, sondern sich aus den in allen real möglichen Eingaben zusammensetzen können. Das bedeutet, das wir später die Äquivalenz zu einer zweiten Spezifikationsebene zeigen werden, in der die bisherige Konkretisierungs*funktion* während TOSTORE durch eine Konkretisierungs*relation* ersetzt wird, die auch andere Eingaben zulässt. Tatsächlich gehen wir in den konkreten Fallstudien sogar noch einen Schritt weiter und beweisen gleich in einem Schritt die korrekte Verfeinerung der abstrakten Ebene zu einer Implementierungs-

ebene, in der alle real möglichen Eingaben auftreten können, und nicht nur die möglichen Ergebnisse von doc2java.

Wir werden später ebenso eine Implementierung von `receive` und `send` angeben, die unsere bisherige Annahme an `receive` und `send` ablöst.

Zunächst wollen wir zur Illustration dieses Sachverhalts diskutieren, welche Eigenschaften eine reale Implementierung der `receive` und `send` Methoden eigentlich besitzen sollte, damit eine softwaretechnisch sinnvolle und einfache Programmierung von Kommunikationsprotokollen mit einem Datentyp wie der `Document` Klassenhierarchie möglich wird. Insbesondere müssen wir uns hierbei auch überlegen, was in der Realität getan werden muss, um überhaupt das Versenden und Empfangen von Pointerstrukturen zu ermöglichen.

Zum Versenden von Objekten des `Document` Typs in der Realität muss eine Kodierung in ein zum Versenden taugliches Format stattfinden. Dieses Format nennen wir im Folgenden das *Übertragungsformat*. In einer Implementierung kommen hier z.B. Folgen von bytes, XML-kodierte Strings oder Ähnliches in Frage. Im Bereich der verteilten Systeme entspricht dies dem Marshalling und Unmarshalling von Nachrichten. Wir verwenden in dieser Arbeit die Begriffe *kodieren* und *dekodieren*. Die Implementierung der Schicht, die das Kodieren und Dekodieren übernimmt, nennen wir im Folgenden *Transformationsschicht*. Real betrachten wir also Anwendungen, die dem Architekturschema aus Abb. 8.1 folgen.

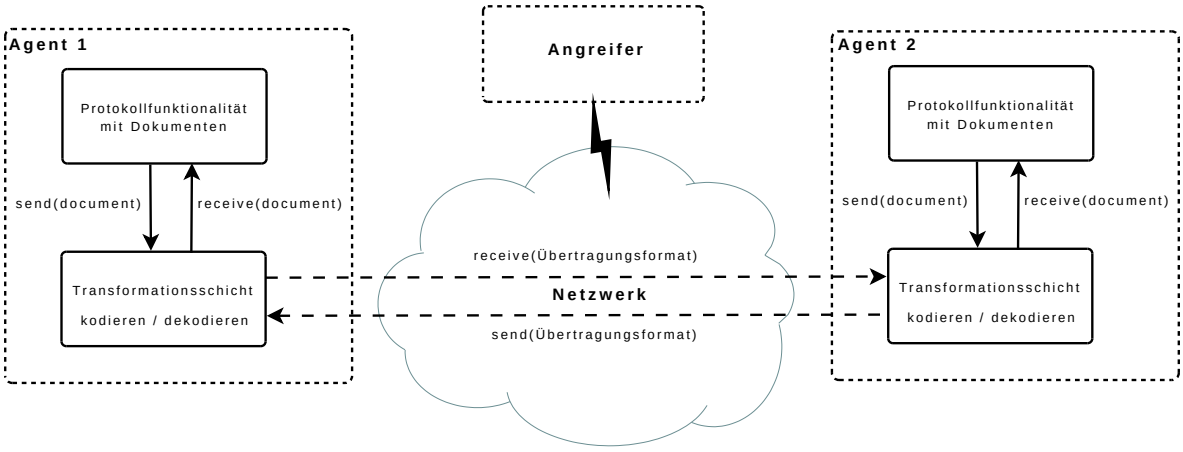


Abbildung 8.1: Architektur von Anwendungen mit Transformationsschicht

Abb. 8.2 stellt darauf aufbauend die neue äquivalente Ebene mit Transformationsschicht und die daraus entstehenden Zusammenhänge bei der Verfeinerung nochmals grafisch dar. Wie dort ersichtlich, sind auf der zweiten Ebene (die bisher beschriebene konkrete Ebene) Eingaben möglich, die nicht Ergebnis von doc2java sein können. Diese würde unsere bisherige Verfeinerungstheorie übersehen.

Eine Transformationsschicht kann neben der reinen Übertragung auch andere Aufgaben wie z.B. den Schutz vor Übertragungsfehlern durch Prüfsummen übernehmen. Insbesondere kann eine solche Implementierung auch inhaltliche Überprüfungen auf den im Übertragungsformat eingehenden und auch auf den ausgehenden Nachrichten durchführen.

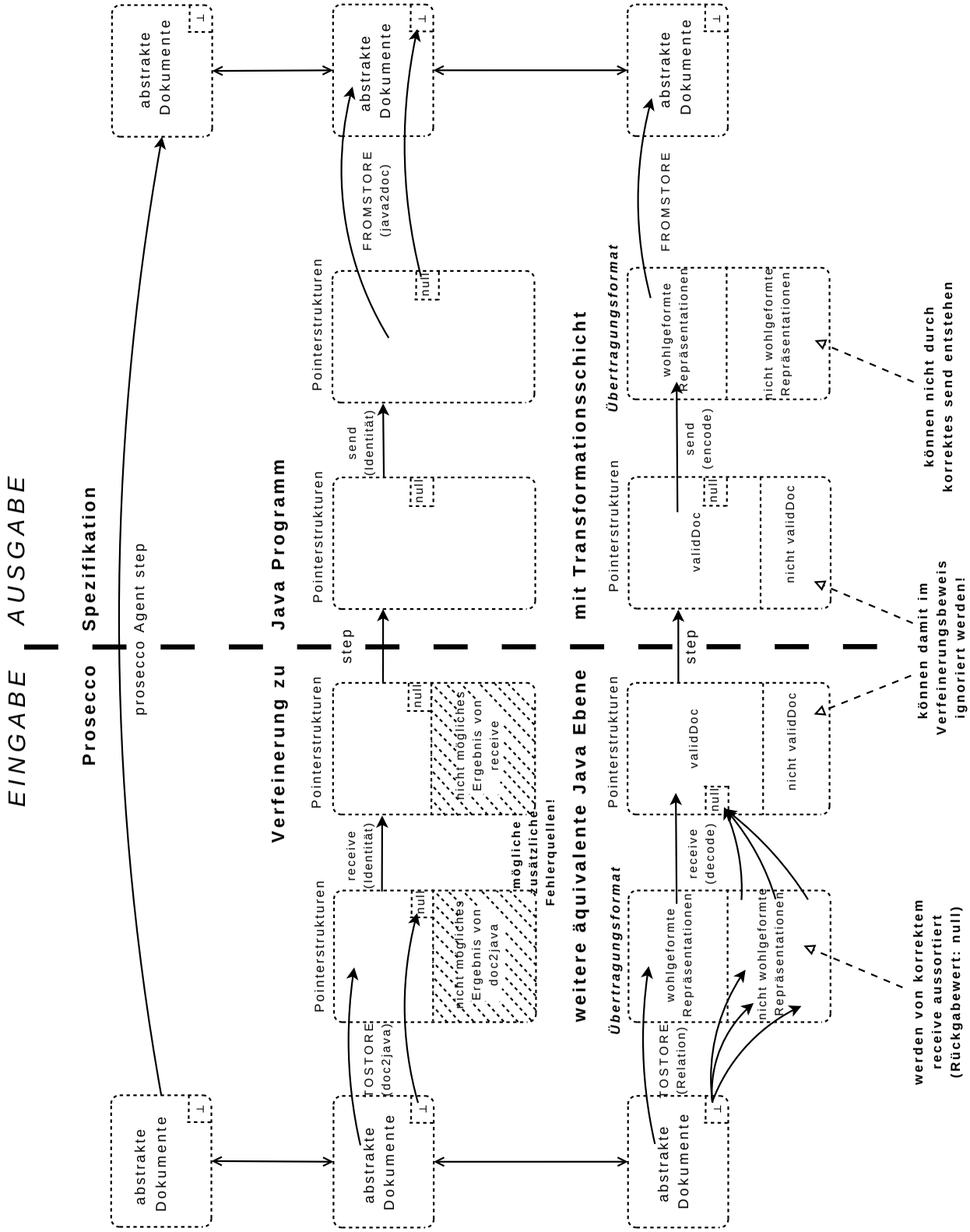


Abbildung 8.2: Übersicht über die bisherigen Ebenen

Diesen Umstand machen wir uns im Rahmen dieser Arbeit zu Nutze, um eine generische Implementierung für eine Transformationsschicht anzugeben, die als mögliche Rückgabewerte nur eine Teilmenge der konkreten Pointerstrukturen, die mittels dem Java `Document` Typ konstruierbar sind, liefern kann. Dies wird den Beweis der korrekten Verfeinerung einer Protokollimplementierung wesentlich vereinfachen, da wir z.B. *nicht* mit zyklischen Eingabedokumenten umgehen können müssen.

Alle anderen Eingaben, die potentiell in der Realität im Übertragungsformat auftreten können, dürfen allerdings nicht zu Sicherheitslücken führen und müssen sogar insbesondere für eine korrekte Implementierung und eine korrekte Verfeinerung auch zu genau dem Verhalten führen, das die Spezifikation zeigt, wenn sie eine ungültige Eingabe erhält. Unsere bisherige Spezifikation beinhaltet allerdings keine ungültige Eingaben in diesem Sinne. Daher haben wir hier eine echte Lücke im bisherigen Framework. Wir müssen diese daher auch in der abstrakten Ebene einführen.

8.2 Erweiterung der abstrakten Dokumente um ein Fehlerdokument

Diese Anforderung machen wir uns nun zu Nutze und benutzen das in Kap. 6 eingeführte Dokument $\perp$, um in der abstrakten Ebene alle ungültigen Eingaben der wirklichen Welt zu repräsentieren. Da dieses Dokument bisher nicht als sinnvolle Eingabe in einer Prosecco Spezifikation vorkommt, führt es in der Spezifikation zum Fehlerfall. Eine Änderung an der abstrakten Protokollspezifikation ist bis auf folgenden Sachverhalt nicht nötig: Wir axiomatisieren, dass der Angreifer $\perp$ immer aus seinem Angreiferwissen ableiten kann und damit eine $\perp$ Nachricht immer versenden kann. Dies stellt sicher, dass $\perp$ in der abstrakten Ebene immer potentiell als Eingabe vorhanden ist.

$\perp$ ist nun also Stellvertreter für alle ungültigen Eingaben in der Realität. Eine Implementierung einer Transformationsschicht muss nun also für *alle* potentiell möglichen und im Übertragungsformat ungültigen Eingaben einen Fehlerfall signalisieren. Dies implementieren wir in der Transformationsschicht durch Rückgabe der Pointerstruktur, die $\perp$ entspricht. Wie oben bereits eingeführt, ist dies einfach der `null` Pointer. Daher spezifizieren wir zusätzlich für doc2java:

Spezifikation:

doc2java-bottom:
$\mathsf{doc2java}(\perp, \mathsf{st}) = \mathsf{jvmref} \times \mathsf{st}$

Eine Implementierung eines Kommunikationsprotokolls, die eine konkrete Implementierung einer Transformationsschicht benutzt, soll nun mit ihren realen Eigenschaften anhand des jeweiligen Übertragungsformats verhaltensäquivalent zu der konkreten Ebene der bisherigen Verfeinerung sein. In dieser zweiten Ebene werden nun real existierende ungültige Eingaben im Übertragungsformat betrachtet und das abstrakte $\perp$ Dokument in der Eingabe als deren Stellvertreter behandelt. Während TOSTORE wird in dieser zweiten Ebene also dann ein Ver-

treter der abstrakten Eingabe im Übertragungsformat dem Speicher hinzugefügt (der dann entweder wohlgeformt oder nicht wohlgeformt sein kann) und während der `receive` Operation dann dekodiert. Die Konkretisierungs*relation* dieser zweiten konkreten Ebene mappt damit später $\perp$ auf alle möglichen ungültigen Eingaben im Übertragungsformat während sie für alle anderen Dokumente eine entsprechende Kodierung erzeugt. Analog wird dann eine Kodierung von der Implementierung von `send` durchgeführt und die resultierende Nachricht im Übertragungsformat während FROMSTORE entsprechend aus dem Speicher extrahiert. Diese Abläufe sind auch in Abb. 8.2 nochmals dargestellt.

Eine konkrete Implementierung einer solchen äquivalenten Ebene wird anhand einer exemplarischen Kodierung in Kap. 10 beschrieben.

Erst wenn eine solche zweite Verfeinerung der bisherigen konkreten Ebene möglich und korrekt ist, ergibt die Verifikation eine sinnvolle Korrektheitsaussage für die reale Implementierung.

8.3 Informelle Charakterisierung von Pointerstrukturen mit abstraktem Gegenstück

Wie bereits in Abb. 8.2 schematisch dargestellt und oben erläutert, wollen wir für eine konkrete Implementierung einer Transformationsschicht eine Charakterisierung derjenigen Pointerstrukturen angeben, die wir als mögliche Rückgabewerte der `receive` Methode erwarten können. Dabei müssen zumindest die möglichen Rückgabewerte der doc2java Funktion korrekt erfasst werden. Im Sinne einer liberaleren Implementierung wollen wir allerdings noch weitere andere Strukturen zulassen.

Wir wollen im Folgenden zunächst eine Charakterisierung für die Ergebnisse der Funktion doc2java versuchen. Wir werden dabei feststellen, dass es für eine reale Implementierung unrealistisch erscheint, als mögliche Ergebnisse genau die Ergebnismenge von doc2java zu haben. Eine reale Implementierung wird also nicht *genau* die Pointerstrukturen zurückliefern können, die von doc2java konstruiert werden können. Statt dessen werden wir im Folgenden unsere von `receive` garantierten Eigenschaften etwas aufweichen und auch noch andere Repräsentationen der abstrakten Dokumente in der Java Welt zulassen, als durch doc2java konstruiert werden können. Dabei versuchen wir, eine aus konzeptioneller Sicht möglichst sinnvolle Einschränkung auf den Pointerstrukturen zu spezifizieren. Programmiertechnisch unsinnige Repräsentationen sollen verboten werden, nur schwierig zu prüfende und zudem nicht wichtige Eigenschaften sollen jedoch auch verletzt werden können. Eine solche Eigenschaft ist etwa die Abwesenheit von Sharing in der Pointerstruktur. Wir erläutern dies im Folgenden genauer.

Abb. 8.3 stellt dazu zur Illustration die Zusammenhänge zwischen den beiden Ebenen der abstrakten Dokumente (links) und der Java Pointerstrukturen (rechts[1]) noch einmal genauer dar. Beide dargestellten Mengen sind dabei abzählbar unendlich, ebenso wie alle dargestellten Teilmengen.

[1] formal charakterisiert durch eine Menge von Tupeln aus einer (Einstiegs-) Referenz (die Wurzel der Pointerstruktur) und einem Speicher (der die Struktur beinhaltet)

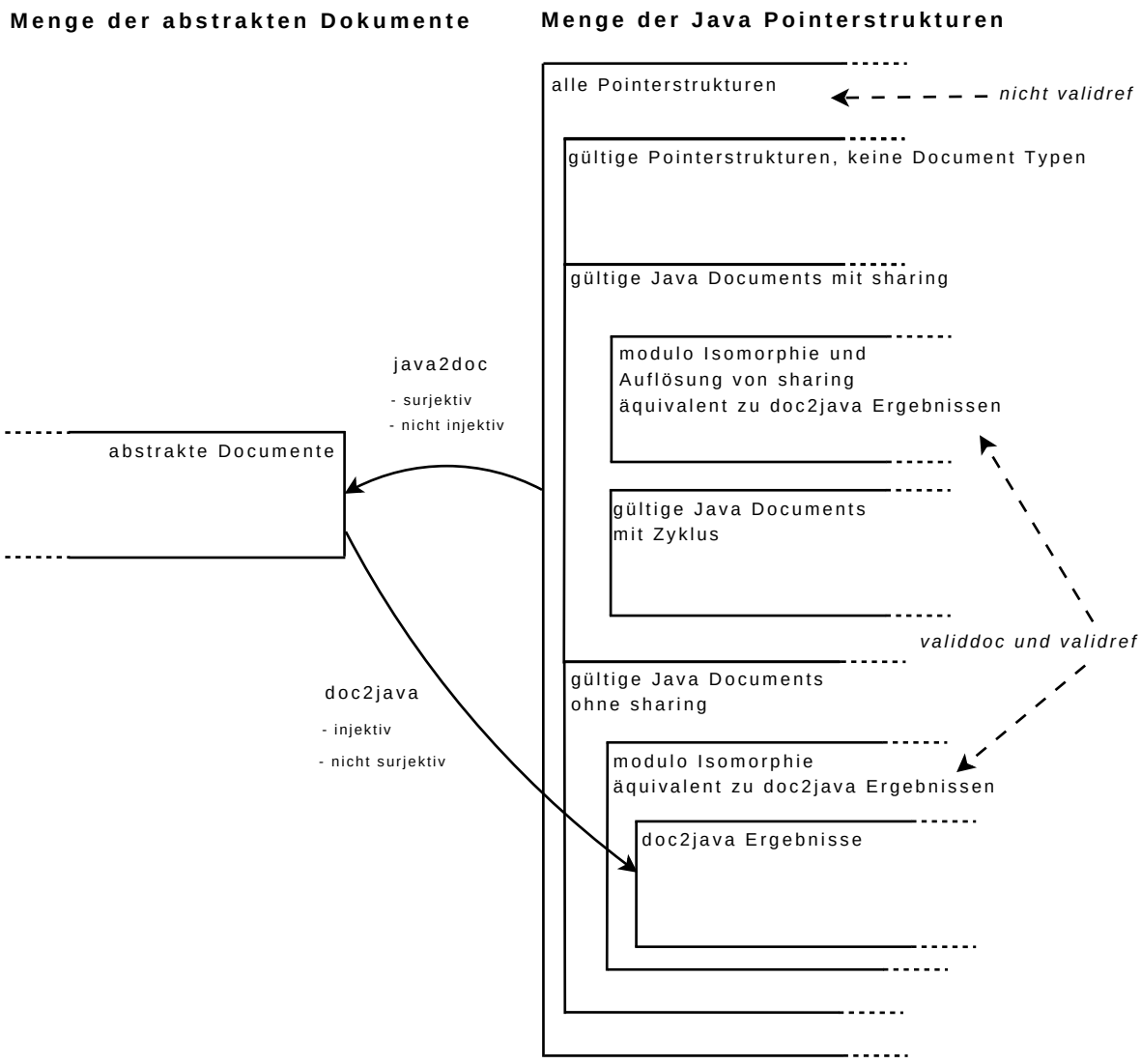


Abbildung 8.3: Die Mengen der abstrakten Dokumente und der Pointerstrukturen

Eine erste Charakterisierung für die Ergebnisse der doc2java Funktion haben wir bereits oben während der Definition von java2doc und auch im vorangegangenen Kapitel kennen gelernt: Zur Konstruktion der Pointerstrukturen wird für jedes Objekt eine neue Referenz herangezogen. Damit kann per Konstruktion das Ergebnis von doc2java nicht zyklisch sein.

Darüber hinaus gelten aber noch weitere Eigenschaften, die in den folgenden Abschnitten beschrieben werden.

8.3.1 Gültige Java Strukturen

Jede mit der Funktion doc2java erzeugte Pointerstruktur ist eine gültige Java Referenz im Sinne von validref. Insbesondere ist die Struktur typkorrekt und beginnt mit einer Referenz, die Subtyp von `Document` ist.

8.3.2 Eindeutige byte[] Repräsentationen von Werten

Beim Umwandeln eines Zahlenwertes eines abstrakten Dokuments (wie z.B. dem Wert i eines IntDoc(i)) wird innerhalb von doc2java immer die Funktion int2bytes verwendet, die den Zahlenwert in kürzest möglich als `byte[ ]` darstellt. Insbesondere bedeutet dies, da sowohl positive als auch negative Werte für i möglich sind, dass das erste bit des ersten bytes des entstehenden Arrays das Vorzeichen angibt (Zweier-Komplement-Darstellung). Insbesondere werden allerdings von int2bytes keine weiteren unnötigen (also den Zahlenwert nicht verändernden) 00- oder FF-bytes am Anfang des entstehenden Arrays generiert. Zusätzlich gibt es keinen Zahlenwert i, der unter int2bytes(i) auf eine leere byte Folge abgebildet wird.

Damit muss also für alle von der Wurzelreferenz erreichbaren byte-Array Referenzen r die folgende Eigenschaft validByteArray(r, st) gelten:

SPEZIFIKATION:

validByteArray-def:
validByteArray(r,st)
$\leftrightarrow$ int2bytes(bytes2int(getbytearray(r,st))) = getbytearray(r,st)
$\wedge$ getbytearray(r,st) $\neq$ []

int2bytes(bytes2int(getbytearray(...))) = getbytearray(...) garantiert dabei, dass die Zahlenrepräsentation getbytearray(...) minimal in obigem Sinne ist.

8.3.3 Keine Pointer auf null für Arrays, Nonces und Keys

Sämtliche Array-getypten Referenzen innerhalb einer Java Dokumentenstruktur nach Konstruktion mit doc2java haben einen Wert ungleich `null`. Dies gilt einfach per Konstruktion, da neue Referenzen (und nur diese werden in addarray innerhalb von doc2java verwendet) per Definition von newref_list(i,st) nicht null sein können. Das gleiche gilt für alle `Nonce` und `Key` Objekte innerhalb von addobj.

8.3.4 Doclists referenzieren den richtigen Array Typ

Innerhalb des `Doclist` Typs wird ein Array verwendet, um die Aggregation zu den beinhalteten Dokumenten zu implementieren. Dieses Array hat den statischen Typ `Document[]` innerhalb der `Doclist` Typdeklaration. Die Java Sprache lässt es nun allerdings zu, dass zur Laufzeit auch Subtypen eines `Document[]` hier referenziert werden. Insbesondere ist z.B. ein `IntDoc[]` möglich. Ein solches Array wird jedoch nicht von doc2java konstruiert. Wir müssen also die möglichen `Doclist` Objekte auf diejenigen einschränken, die ein `Document[]` beinhalten.

8.3.5 Kein Sharing innerhalb der Struktur

Per Konstruktion ist jede Referenz innerhalb der von doc2java konstruierten Pointerstruktur neu bezüglich dem Speicher zuvor. Ebenfalls sind alle verwendeten Referenzen per Konstruktion disjunkt. Damit ist die Struktur insbesondere nicht zyklisch, wie bereits oben bemerkt. Es gilt aber sogar noch eine stärkere Eigenschaft. Die Pointerstruktur besitzt kein Sharing. Sharing liegt dabei dann vor, wenn von einer Referenz in der Struktur eine andere Referenz in der Struktur auf zwei verschiedenen Wegen entlang der Verkettung der Referenzen erreicht werden kann. Ein Zyklus ist damit ein Spezialfall von Sharing. Die Sharingfreiheit liegt vor, da jede der verwendeten Referenzen nur an genau einer Stelle referenziert wird. Damit entsteht eine Baumstruktur. Damit müssen auch Pointerstrukturen mit Sharing ausgeschlossen werden. Wir spezifizieren dafür ein Bibliotheksprädikat für allgemeine Pointerstrukturen hasSharing : reference $\times$ store.

8.3.6 Gesamte Struktur disjunkt zu allen bisherigen Speicherinhalten

Durch die beschriebene Konstruktion ist jede Referenz innerhalb der Struktur neu. Das bedeutet, dass die vorher im Speicher enthaltenen gültigen Java Pointerstrukturen (wie z.B. die Objekte, die die Protokollimplementierung darstellen) diese neuen Referenzen nicht referenzieren können. Es gibt also keinen Pfad von schon im Ursprungsspeicher gültigen Referenzen hin zu einer Referenz die von der Wurzel der von doc2java konstruierten Struktur erreichbar ist. Dies kann für Programme, die mit destruktiven Updates auf den Pointern arbeiten, wesentlich sein.

8.4 Formale Charakterisierung von Pointerstrukturen mit abstraktem Gegenstück

Die eben beschriebenen Eigenschaften charakterisieren genau die Ergebnisse von doc2java. Am besten wäre es also, wenn eine reale Implementierung von `receive` auch nur genau solche Pointerstrukturen zurückliefert. Dies erscheint allerdings als zu starke Einschränkung. Insbesondere die Forderungen nach Sharing-Freiheit und Disjunktheit zu allen bisherigen Pointerstrukturen würden Caching und Wiederverwendung der Eingabedokumente oder speichereffiziente Implementierungen von `receive` verhindern. Dies erscheint nicht wünschenswert. Wir verwenden daher nicht alle oben genannten Eigenschaften, sondern versuchen, mit den einfachsten und grundlegendsten auszukommen. Natürlich müssen wir dann Implementierungen für die eigentliche Protokollfunktionalität angeben, die auch mit Eingaben mit Sharing oder mit gecachten Eingaben zurechtkommen. Dies ist allerdings unproblematisch.

Wie bereits vorher angemerkt, sollen zu schwierig zu überprüfende und zu restriktive Einschränkungen verletzt werden können. Formal definieren wir nun ein Charakterisierungsprädikat für die Ergebnisse von `receive`, mit dem wir die gewünschten Eigenschaften zu fassen versuchen. Dazu können wir nun die validDoc Charakterisierung verwenden, die wir bereits in Kap. 6.8 eingeführt haben. Wir verwenden:

SPEZIFIKATION:

isAbstractDoc-def:
isAbstractDoc(r,st)
↔ validDoc(r,st)
∧ validref(r,Document, st)

validref ist dabei die bereits oben eingeführte Charakterisierung typkorrekter Java Pointerstrukturen. Dies ist sicher als Eigenschaft sinnvoll, da kein gültiges Java Programm ungültige (z.B. nicht typkorrekte) Strukturen erzeugen kann. validDoc dagegen setzt gerade die obigen Forderungen nach *Zyklenfreiheit*, *eindeutiger byte[ ] Repräsentation*, *richtiger Array Typen* und *keinen Nullpointern* um.

8.5 Validierung: Eigenschaften von java2doc und doc2java

Mit dem Prädikat isAbstractDoc haben wir eine Charakterisierung für die Pointerstrukturen gefunden, die wir in einer Implementierung als Rückgabewerte von `receive` erlauben wollen. Man beachte z.B., dass die gegebene Axiomatisierung für isAbstractDoc Sharing in den Pointerstrukturen erlaubt. Innerhalb des Verfeinerungsbeweises könnten wir nun noch weitere Eigenschaften nutzen, die von doc2java garantiert werden (wie z.B. die Sharingfreiheit). Wie bereits oben diskutiert impliziert aber unsere Verifikation einer Verfeinerung nur dann einen sinnvollen Korrektheitsbegriff für eine Implementierung, wenn mittels einem konkreten Übertragungsformat und einer konkreten Implementierung für die Transformationsschicht eine verhaltensäquivalente alternative Spezifikation (mit gleicher Protokollimplementierung) der konkreten Ebene angegeben werden kann, die dann statt lediglich den Entsprechungen von abstrakten Dokumenten (und damit für das Programm auch nur den Ergebnissen von doc2java) auch andere in der Realität mögliche Eingaben zulässt. Diese zusätzlichen Eingaben müssen dann in der Implementierung der Transformationsschicht aussortiert werden und zum Rückgabewert `null`, der dem Fehlerdokument $\perp$ entspricht, führen. Das bedeutet wiederum, dass wir, wenn wir eine solche Verhaltensäquivalenz nachweisen wollen, im Beweis der Verfeinerung nur Eigenschaften der Transformationsschicht verwenden dürfen, die in der realen Implementierung auch gelten. Da Eigenschaften wie z.B. Sharingfreiheit nicht immer benötigt werden, liberalisieren wir die Eigenschaften hier anhand von isAbstractDoc.

Schließlich stellt sich die Frage, ob unsere Axiomatisierungen wirklich die richtigen und intendierten Eigenschaften treffen, die wir spezifizieren wollten. Zur diesbezüglichen Validierung unserer Axiomatisierung beweisen wir nun einige Eigenschaften für isAbstractDoc. Die Beweise der Eigenschaften wurden mit dem KIV System geführt und können in der Spezifikation docs2store nachgelesen werden [49]. Eine erste naheliegende Eigenschaft ist:

Jedes Ergebnis von doc2java erfüllt validref und validDoc:

THEOREM:

doc2java-validdoc:
doc2java(doc,st) = r × st_0 → validDoc(r, st_0)

doc2java-validref:
doc2java(doc,st) = r × st_0 → validref("docs2store", r, Document, st_0)

und damit auch:

THEOREM:

doc2java-abstractDoc:
doc2java(doc,st) = r × st_0 → isAbstractDoc(r, st_0)

Im Weiteren definieren wir nun einen Äquivalenzbegriff für Pointerstrukturen, der Sharing in der Pointerstruktur und Isomorphie in den Werten der *Referenzen* erlaubt, allerdings Gleichheit in den *primitiven Werten* und den *Typen* fordert. Damit können wir nun auch die umgekehrte Richtung dieser Implikation zeigen und damit beweisen, dass isAbstractDoc modulo diesem Äquivalenzbegriff genau die Ergebnisse von doc2java charakterisiert. Damit haben wir auch die Darstellung in Abb. 8.3 gerechtfertigt.

Der gewünschte Äquivalenzbegriff ist dabei rekursiv für beliebige Pointerstrukturen definiert. Wir spezifizieren eine Äquivalenzrelation pseq : (reference × store) × (reference × store) zwischen zwei Pointerstrukturen, jeweils charakterisiert durch ihre Einstiegsreferenz und den jeweiligen Speicher:

SPEZIFIKATION:

PSEQ:
pseq(r, st, r_0, st_0)
↔ (r = jvmref ∧ r_0 = jvmref)
∨ (r ≠ jvmref ∧ r_0 ≠ jvmref
∧ st[r - .type] = st_0[r_0 - .type]
∧ (is_classtype(st[r - .type]) → classeq(r, st, r_0, st_0)
∧ (is_arraytype(st[r - .type]) → arrayeq(r, st, r_0, st_0))

Zwei Strukturen beginnend bei den Referenzen r in st und bei r_0 in st_0 sind in obigem Sinne äquivalent, wenn beide gleich jvmref sind. Ebenso sind sie äquivalent, wenn beide ungleich jvmref sind und ihre Typen gleich sind. Darüber hinaus müssen dann bei einem Klassentyp die referenzierten Felder wieder äquivalent sein (classeq(r, st, r_0, st_0)), bzw. bei einem Arraytyp die referenzierten Arrayinhalte ebenso (arrayeq(r, st, r_0, st_0))

Dabei gilt:

SPEZIFIKATION:

classeq:

classeq(r, st, r_0, st_0)
$\leftrightarrow$ (∀ fs. fs ∈ instfields(st[r - .type])
∧ is_referencevalue(st[r - .fs])
→ pseq(st[r - .fs].refval, st, st_0[r_0 - .fs].refval, st_0))
∧ (∀ fs. fs ∈ instfields(st[r - .type])
∧ ¬ is_referencevalue(st[r - .fs])
→ st[r - .fs] = st_0[r_0 - .fs])

Eine Referenz r mit Klassentyp ist dabei in Speicher st äquivalent zu einer anderen Referenz r_0 in st_0, wenn für alle Instanzenfelder des jeweiligen Typs (∀ fs. fs ∈ instfields(st[r - .type]) ...) gilt: wenn ein Referenzwert im Feld enthalten ist (is_referencevalue(st[r - .fs])), müssen die entsprechenden Referenzen wieder äquivalent sein (Rekursion von pseq). Ansonsten müssen die Werte der Felder (die dann primitiv sind) einfach gleich sein.

Für Arrays gilt die gleiche Spezifikation für Äquivalenz, lediglich müssen hier auch die Längen der Arrays gleich sein und wir sprechen nicht über die Felder einer Klasse sondern über alle gültigen Arrayindizes ($0 \leq i \wedge i <$ st[r - .length].intval). Damit gilt für Arrays:

SPEZIFIKATION:

arrayeq:

arrayeq(r, st, r_0, st_0)
$\leftrightarrow$ st[r - .length] = st_0[r_0 - .length]
∧ (∀ i. $0 \leq i \wedge i <$ st[r - .length].intval
∧ is_referencevalue(st[r - i])
→ pseq(st[r - .i].refval, st, st_0[r_0 - .i].refval, st_0))
∧ (∀ i. $0 \leq i \wedge i <$ st[r - .length].intval
∧ ¬ is_referencevalue(st[r - .i])
→ st[r - .i] = st_0[r_0 - .i])

Mit diesem Äquivalenzbegriff lässt sich nun folgende Eigenschaft zeigen: Jede Pointerstruktur, für die isAbstractDoc gilt, lässt sich modulo pseq äquivalent durch Konstruktion mittels

doc2java aus einem abstrakten Dokument erzeugen. Formal:

THEOREM:

doc2java-abstractDoc-correct:
$$\begin{aligned}&\mathsf{isAbstractDoc}(r, st)\\ \rightarrow\ &\exists\ \mathsf{doc}, st_0, r_0, st_1.\\ &\quad \mathsf{doc2java}(\mathsf{doc}, st_0) = r_0 \times st_1 \wedge \mathsf{pseq}(r, st, r_0, st_0)\end{aligned}$$

Des Weiteren gilt auch auf der jeweiligen durch pseq definierten Äquivalenzklasse, dass doc2java die Umkehrfunktion von java2doc ist. Formal:

THEOREM:

doc2java-java2doc:
$$\begin{aligned}&\mathsf{isAbstractDoc}(r, st)\\ \wedge\ &\mathsf{doc2java}(\mathsf{java2doc}(r, st)) = r_1 \times st_1\\ \rightarrow\ &\mathsf{pseq}(r, st, r_1, st_1)\end{aligned}$$

Die andere Richtung gilt natürlich ohne eine solche Einschränkung:

THEOREM:

java2doc-doc2java:
$$\begin{aligned}&\mathsf{doc2java}(\mathsf{doc}, st) = r_1 \times st_1\\ \rightarrow\ &\mathsf{java2doc}(r_1, st_1) = \mathsf{doc}\end{aligned}$$

Damit ist eine hinreichende Validierung der Spezifikationen von validDoc und doc2java und java2doc bewiesen.

8.6 Nötige Veränderungen im Aufbau der konkreten ASM

In diesem Abschnitt beschreiben wir nun die Änderungen an der Spezifikationsstruktur für unsere Ebenen der Verfeinerung, damit die oben beschriebenen Zusammenhänge abgebildet werden. Die Änderungen betreffen dabei die Makros TOSTORE und FROMSTORE sowie die verwendeten Eigenschaften der Funktionen `receive` und `send`.

Das TOSTORE Makro darf nun nicht mehr lediglich doc2java verwenden, um eine Eingabe in den Speicher zu legen, sondern muss die Kodierung vornehmen und insbesondere auch alle nicht wohlgeformten Kodierungen berücksichtigen. FROMSTORE muss entsprechend dekodieren. Dazu müssen wir uns auf ein konkretes Übertragungsformat festlegen (jedoch noch nicht auf die konkreten Kodierungsvorschriften!). Wir wählen hier Folgen von Bytes als Übertragungsformat, die z.B. bei Chipkarten, Bluetooth Kommunikation oder Sockets Standard sind. Insbesondere benötigen wir nun ein Prädikat, dass die Gültigkeit von Kodierungen festlegt. Wir verwenden dazu validEncoding : bytes. Man beachte, dass die Spezifikation dieses Prädikates erst vom konkreten Kodierungsmechanismus abhängig ist und an dieser Stelle noch keine Rolle spielt (tatsächlich geben wir später in Kap. 10 eine exemplarische Kodierungsvorschrift und eine Spezifikation für z.B. validEncoding an).

Ebenso benötigen wir Funktionen zur Kodierung und Dekodierung von Dokumenten in Folgen von Bytes. Dazu verwenden wir encode : Document → bytes und decode : bytes → Document. Auch für diese Funktionen geben wir bis jetzt noch keine ausführlichen Spezifikationen an, wichtig sind lediglich die beiden Eigenschaften:

```
decode(encode(doc)) = doc

validEncoding(bytes) → encode(decode(bytes)) = bytes
```

Damit ändern wir nun unsere Definitionen von TOSTORE und FROMSTORE aus Kap. 5.4.2 wie folgt ab, wobei die Änderungen *kursiv* gedruckt sind:

```
TOSTORE(inputs; cstore)
   if inputs(agent)(port_1) = [ ] then
      cstore(agent) := cstore(agent)[.ins_1, null];
    else if inputs(agent)(port_1) ≠ [ ] ∧ inputs(agent)(port_1).first = ⊥ then
      choose bytes with ¬ validEncoding(bytes) in {
         cstore(agent) := addarray(.ins_1, bytes, cstore(agent)) } ;}
   else if inputs(agent)(port_1) ≠ [ ] then
      cstore(agent) :=
         addarray(.ins_1, encode(inputs(agent)(port_1).first), cstore(agent));
   ...
   if inputs(agent)(port_n) = [ ] then
      cstore(agent) := cstore(agent)[.ins_n, null];
    else if inputs(agent)(port_n) ≠ [ ] ∧ inputs(agent)(port_n).first = ⊥ then
```

```
        choose bytes with ¬ validEncoding(bytes) in {
            cstore(agent) := addarray(.ins_n, bytes, cstore(agent)) } ;
    else if inputs(agent)(port_n) ≠ [ ] then
        cstore(agent) :=
            addarray(.ins_n, encode(inputs(agent)(port_n).first), cstore(agent));
    cstore(agent) := cstore(agent)[.outport, intval(0)];
    cstore(agent) := cstore(agent)[.outs, null];
    cstore(agent) := cstore(agent)[.rec_1, boolvar(false)];
    ...
    cstore(agent) := cstore(agent)[.rec_n, boolvar(false)];
```

Bei einer Eingabe von $\perp$ (wie bereits angemerkt kann diese immer vom Attackermodell generiert werden), wählt TOSTORE nun eine *beliebige* ungültige Kodierung und fügt diese als `byte[ ]` an die jeweilige temporäre Eingabestelle in den Speicher ein. Damit berücksichtigen wir alle möglichen ungültigen Kodierungen als Eingabe. Bei anderen Eingabedokumenten verwenden wir einfach die (bis jetzt noch unspezifizierte) encode Funktion, um eine Kodierung herzustellen und fügen diese in den Speicher ein. Der Rest des Makros bleibt unverändert.

Auch FROMSTORE muss nun entsprechend verändert werden:

```
FROMSTORE(st; cstore, inputs, outdoc, outport)
    cstore(agent) := st;
    outport := cstore(agent)[.outport].intval;
    outdoc := decode(getbytearray(.outs, cstore(agent)))
    if (cstore(agent)[.rec_1]) then inputs(agent)(1) = inputs(agent)(1).rest;
    ...
    if (c store(agent)[.rec_n]) then inputs(agent)(n) = inputs(agent)(n).rest;
```

Statt der vorherigen Anwendung von java2doc wird nun decode auf einem `byte[ ]` an der `.outs` Speicherstelle verwendet, um die abstrakte Entsprechung outdoc der Ausgabe des Programms zu konstruieren.

Natürlich kann man TOSTORE und FROMSTORE im generischen Verfeinerungsframework nicht ändern, ohne auch die verwendeten Eigenschaften von `receive` und `send` entsprechend anzupassen. Vorher forderten wir einfach, dass `receive(i)` einfach den Inhalt des Feldes $\texttt{.ins}_i$ zurückgibt (also als einfache Abfragefunktion auf dem Feld darstellt). Hier muss nun eine entsprechende Dekodierung stattfinden *und* alle ungültigen Kodierungen erkannt werden. Die entsprechenden Eigenschaften für `receive` sind damit:

```
receive-ok:
    st[.mode] = normal, .mode ∈ st,
```

st[.ins$_i$] = refval(r$_1$)
bytes = getbytearray(r$_1$, st)
validEncoding(bytes),
validref("CommInterface", r$_2$, CommInterface, st), r$_2 \neq$ jvmref,
validref("docs2store", r$_1$, mkarraytype(byte_type), st), r$_1 \neq$ jvmref,
st = st$_0$
$\vdash \langle$st; `r3 = r2.receive(i);` $\rangle$
(st$_0$[.rec$_i$, boolval(true)] $\subseteq$ st
$\wedge$ java2doc(r$_3$, st) = decode(bytes)
$\wedge$ validDoc(r$_3$, st) $\wedge$ validref("docs2store", r$_3$, Document, st))

receive-ok besagt, dass alle gültigen Kodierungen (validEncoding(...)) im Eingabepuffer st[.ins$_i$] zu einem Rückgabewert r_3 führen, der bezüglich java2doc genau die Entsprechung der abstrakten Dekodierung decode auf dem Eingabearray entspricht.

receive-fail:
st[.mode] = normal, .mode $\in$ st,
st[.ins$_i$] = refval(r$_1$)
bytes = getbytearray(r$_1$, st)
$\neg$ validEncoding(bytes),
validref("CommInterface", r$_2$, CommInterface, st), r$_2 \neq$ jvmref,
validref("docs2store", r$_1$, mkarraytype(byte_type), st), r$_1 \neq$ jvmref,
st = st$_0$
$\vdash \langle$st; `r3 = r2.receive(i);` $\rangle$
(st$_0$[.rec$_i$, boolval(true)] $\subseteq$ st
$\wedge$ r$_3$ = jvmref)

receive-fail besagt, dass alle ungültigen Kodierungen erkannt werden und durch jvmref als Rückgabewert der `receive` Methode signalisiert werden. Damit filtert `receive` generisch alle ungültigen Eingaben aus.

Schließlich benötigen wir spiegelbildliche Eigenschaften für send. Bei einer wohlgeformten Eingabe im Sinne von validDoc soll eine entsprechende Kodierung erzeugt werden:

send-ok:
st[.mode] = normal, .mode $\in$ st,
validDoc(r$_1$, st),
validref("CommInterface", r$_2$, CommInterface, st), r$_2 \neq$ jvmref,
validref("docs2store", r$_1$, Document, st), r$_1 \neq$ jvmref
st = st$_0$

$$
\begin{array}{l}
\vdash \langle \mathsf{st};\ \mathtt{r_2.send(r_1)}\ \rangle \\
\quad (\ \ \exists\ \mathsf{r}.\quad \mathsf{addarray}(\mathsf{r},\ \mathsf{mkarraytype}(\mathsf{byte\_type}), \\
\qquad\qquad\qquad \mathsf{encode}(\mathsf{java2doc}(\mathsf{r_1},\ \mathsf{st_0})),\ \mathsf{st_0})[.\mathsf{outs},\ \mathsf{refval}(\mathsf{r})] \\
\qquad\quad \subseteq \mathsf{st} \\
\qquad \wedge\ \mathsf{newref}(\mathsf{r},\ \mathsf{st_0}))
\end{array}
$$

Diese Eigenschaft würde eigentlich schon reichen (schließlich können wir immer zeigen, dass korrekte Protokollimplementierungen immer gültige Dokumentenstrukturen als Eingabe produzieren). Aus Gründen der Robustheit fordern wir allerdings noch, dass ungültige Dokumente auch erkannt werden und in diesen Fällen nichts versendet wird:

send-fail:

$$
\begin{array}{l}
\quad \mathsf{st}[.\mathsf{mode}] = \mathsf{normal},\ .\mathsf{mode} \in \mathsf{st}, \\
\quad \neg\ \mathsf{validDoc}(\mathsf{r_1},\ \mathsf{st}), \\
\quad \mathsf{validref}(\text{"CommInterface"},\ \mathsf{r_2},\ \mathsf{CommInterface},\ \mathsf{st}),\ \mathsf{r_2} \neq \mathsf{jvmref}, \\
\quad \mathsf{validref}(\text{"docs2store"},\ \mathsf{r_1},\ \mathsf{Document},\ \mathsf{st}),\ \mathsf{r_1} \neq \mathsf{jvmref} \\
\quad \mathsf{st} = \mathsf{st_0} \\
\vdash \langle \mathsf{st};\ \mathtt{r_2.send(r_1)}\ \rangle \\
\qquad (\mathsf{st_0} \subseteq \mathsf{st})
\end{array}
$$

Wir geben in Kap. 10 eine Implementierung für `send` und `receive` und entsprechende Kodierungsregeln an, die die oben genannten Eigenschaften tatsächlich erfüllen. Damit ist also die Lücke zur Verifikation einer tatsächlichen Implementierung geschlossen.

8.7 Erhalt aller bisherigen Eigenschaften für die Verifikation

Der eigentlich interessante Aspekt ist nun, dass die neue Axiomatisierung von TOSTORE und FROMSTORE sowie die neuen Eigenschaften `receive` und `send` *immer noch die gleichen Lemmata* zur Verifikation der korrekten Verfeinerung einer Protokollimplementierung erfüllen. Insbesondere gelten die beiden Eigenschaften TOSTORE-receive und FROMSTORE-send, die bereits in Kap. 6.9 eingeführt wurden, weiterhin. Da lediglich diese Eigenschaften innerhalb der bisherigen Verfeinerungsbeweise verwendet wurden, ist die Verifikation der neuen zusätzlichen konkreten Ebene *vollständig analog* zur bisher beschriebenen Verifikation. Insbesondere können wir gleich direkt die Korrektheit der Verfeinerung zur untersten Ebene in Abb. 8.2 beweisen und können den Beweis genauso führen, als würden wir ohne Übertragungsformat verifizieren. Hier wird insbesondere deutlich, dass die generische Transformationsschicht tatsächlich transparent für die Protokollimplementierung ungültige Eingaben filtert und entsprechend der Beweis der Verfeinerungskorrektheit einer Protokollimplementierung dadurch auf einer höheren Abstraktionsebene möglich wird, als dies ohne eine solche Schicht möglich wäre.

Es muss allerdings nun natürlich der Beweis der Eigenschaften entsprechend angepasst werden.

Dies beschreiben wir kurz in den nächsten beiden Abschnitten:

8.7.1 TOSTORE und `receive`

Zur Wiederholung nochmals die für den Verfeinerungsbeweis gewünschte Eigenschaft:

THEOREM:

TOSTORE-receive:

$$
\begin{array}{ll}
 & \quad .\mathsf{mode} \in \mathsf{st} \wedge \mathsf{st}[.\mathsf{mode}] = \mathsf{noval} \\
 & \wedge\ \mathsf{validrefnotnull}(\text{"CommInterface"}, \mathsf{r}, \mathsf{CommInterface}, \mathsf{st}) \\
 & \wedge\ \mathsf{inputs}(\mathsf{agent})(\mathsf{i}) \neq [\,] \\
 & \wedge\ \langle \mathsf{TOSTORE}(\mathsf{agent}, \mathsf{inputs}; \mathsf{cstore}) \rangle\ (\mathsf{cstore} = \mathsf{cstore}_1) \\
 & \wedge\ \mathsf{st} = \mathsf{cstore}_1(\mathsf{agent}) \\
 & \langle \mathsf{st};\ \mathsf{r}_1 = \mathsf{r.receive}(\mathsf{i}) \rangle\ (\mathsf{st} = \mathsf{st}_0 \wedge \mathsf{r}_1 = \mathsf{r}_2) \\
\rightarrow & \quad \mathsf{java2doc}(\mathsf{r}_2, \mathsf{st}_0) = \mathsf{inputs}(\mathsf{agent})(\mathsf{i}).\mathsf{first} \\
 & \wedge\ \mathsf{st} \subseteq \mathsf{st}_0 \\
 & \wedge\ \mathsf{validref}(\text{"docs2store"}, \mathsf{r}_2, \mathsf{Document}, \mathsf{st}_0) \\
 & \wedge\ \mathsf{validDoc}(\mathsf{r}_2, \mathsf{st}_0)
\end{array}
$$

Beweisidee:

Auf der konkreten Ebene unter Berücksichtigung eines Übertragungsformates müssen nun für den Beweis dieser Eigenschaft auch ungültige Eingaben berücksichtigt werden. TOSTORE entscheidet schließlich anhand der abstrakten Eingabe (Vorliegen von $\perp$), ob eine ungültige Kodierung als Eingabe für das Java Programm gewählt werden soll. Da `receive` nun allerdings wie oben dargestellt diese ungültigen Eingaben erkennt (Lemma receive-fail oben!) und in diesen Fällen jvmref zurückliefert, gilt TOSTORE-receive weiterhin. Schließlich ist jvmref gerade die Entsprechung von $\perp$ = inputs(agent)(i).first bezüglich java2doc. Im Fall wohlgeformter Eingabekodierungen folgt die obige Eigenschaft einfach aus decode(encode(doc)) = doc, da TOSTORE encode verwendet und `receive` sich nach Lemma receive-ok wie decode verhält.

8.7.2 FROMSTORE und `send`

Auch hier wiederholen wir kurz die gewünschte Eigenschaft:

THEOREM:

FROMSTORE-send:

$$
\begin{aligned}
&\text{.mode} \in \text{st} \wedge \text{st[.mode]} = \text{noval} \\
&\wedge\ \text{validrefnotnull("CommInterface", r, CommInterface, st)} \\
&\wedge\ \text{validref("docs2store", } r_1\text{, Document, st)} \\
&\wedge\ \text{validDoc}(r_1, \text{st}) \\
&\text{st} = \text{st}_0 \\
\rightarrow\ &\langle \text{st; r.send}(r_1, \text{outport}_0) \rangle \\
&\quad \langle \text{FROMSTORE(agent, st; cstore, inputs, outdoc, outport)} \rangle \\
&\quad\ (\ \ \text{outdoc} = \text{java2doc}(r_1, \text{st}_0) \\
&\quad\ \ \wedge\ \text{outport} = \text{outport}_0)
\end{aligned}
$$

Beweisidee:

Spiegelbildlich zur obigen Argumentation gilt auch diese Eigenschaft weiterhin, da `send` nach Eigenschaft send-ok von oben sich wie encode verhält und FROMSTORE decode auf dessen Ausgabe verwendet. Damit benötigen wir wiederum decode(encode(doc)) = doc und können den Beweis wieder schließen. Wie bereits in Kap. 6.9 erläutert, benötigen wir für den Verfeinerungsbeweis keine Eigenschaft, die das Verhalten von `send` bei nicht wohlgeformten Eingaben beschreibt, da korrekte Programme nur wohlgeformte Ausgaben produzieren.

8.8 Resultierende Kommunikationsstruktur in der Implementierung

Mit encode und decode haben wir nun Formalisierungen für eine Transformationsschicht in die Verfeinerungsmethodik integriert. Natürlich muss eine reale Implementierung eines Protokolls nun auch auf einer entsprechenden Implementierung für `encode` und `decode` innerhalb von `receive` und `send` basieren. Eine solche Implementierung wird in Kapitel 10 erläutert. Am Ende von Kap. 10 wird dann auch gezeigt, dass die beiden eben gezeigten Eigenschaften FROMSTORE-send und TOSTORE-receive mit dieser Implementierung tatsächlich gelten. Zur Erläuterung der Funktionsweise einer solchen Kommunikationsinfrastruktur wird in Abb. 8.4 gezeigt, wie die Implementierung der Protokollfunktionalität eines Agenten (`Protocol`) mit der Umgebung interagiert.

Ein Schritt (`step()`) eines Agenten beginnt demnach typischerweise damit, eine Eingabe zu empfangen (`receive()`). Dabei kann ein Port `port_i` angegeben werden. Dies führt nun im Kommunikationsinterface dazu, dass eine entsprechende low-level Empfangsfunktion in einer Netzwerkschicht (die wir hier nicht mehr betrachten) aufgerufen wird. Dies kann bei Cindy z.B. das Empfangen einer MMS sein. Die daraus resultierende Eingabe (hier: eine Bytefolge `inbytes`) wird nun dekodiert (`decode`) und das Ergebnisdokument `indoc` zurückgegeben.

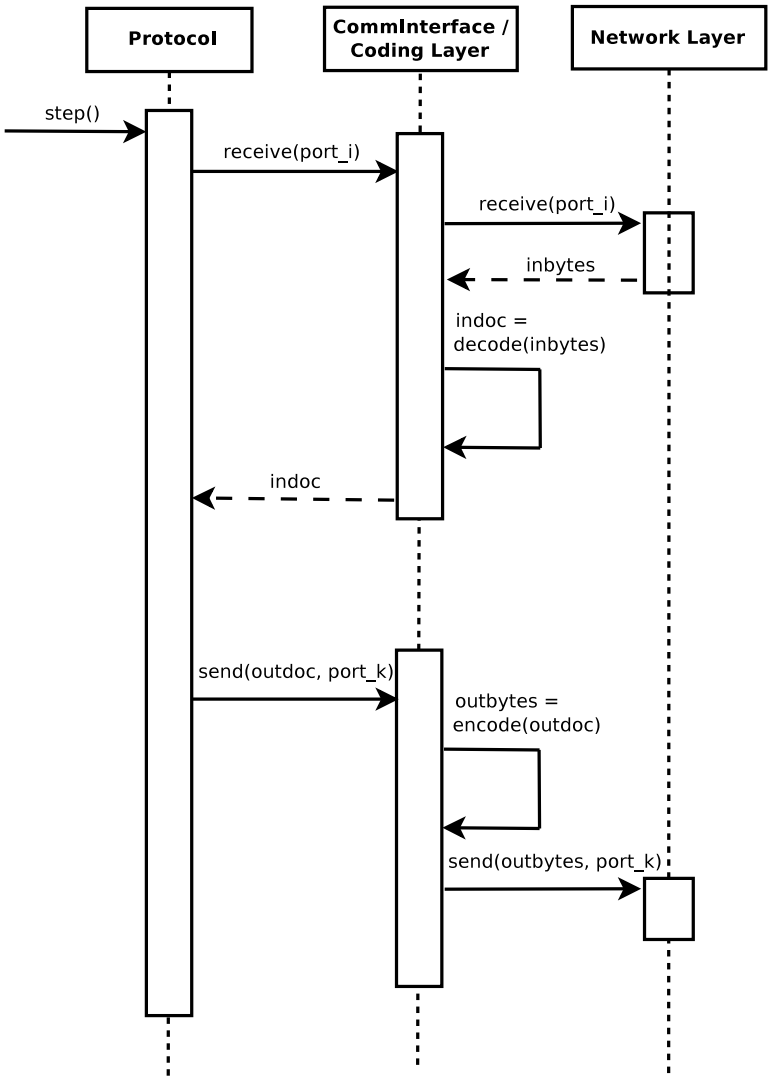


Abbildung 8.4: Kommunikation mit der Transformationsschicht

Nach Abarbeitung des dazugehörigen Protokollschritts wird schließlich eventuell die `send` Methode aufgerufen, um eine Antwort `outdoc` auf Port `port_k` zu verschicken. Dies führt nun spiegelbildlich zu Kodierung mittels `encode` zu einer Bytefolge `outbytes`, die dann über die Netzwerkschicht verschickt werden kann.

Kapitel 9

Erweiterung der Datentypen: Kryptographie

It's kind of fun to do the impossible.
Walt Disney

In diesem Kapitel werden die Java Klassen zur Repräsentation der abstrakten Dokumente um Typen für die kryptographischen Operationen erweitert. Dazu werden insbesondere die erforderlichen Eigenschaften der Kryptographie für ein Refinement diskutiert. Ebenso werden entsprechende Definitionen der bisherigen Wandlungsfunktionen eingeführt. Die in diesem Kapitel beschriebenen Arbeiten wurden in [62] und [66] publiziert.

9.1 Abstrakte kryptographische Operationen

Die PROSECCO Ebene unterstützt mit den Typen HashDoc, EncDoc und SigDoc die kryptographischen Operationen. Dabei basiert die Angreifermodellierung in PROSECCO auf der Annahme der *perfekten Kryptographie*. Dies bedeutet, dass ein Angreifer kryptographische Operationen nicht *brechen* kann. Insbesondere kann er verschlüsselte Daten nur unter Kenntnis des richtigen Schlüssels entschlüsseln, Hashfunktionen nicht invertieren und Signaturen nicht fälschen. Insbesondere bedeutet dies z.B. auch dass alle genannten Funktionen injektiv spezifiziert werden. Die Annahme perfekter Kryptographie ist Standard im Bereich der Kryptoprotokoll-Verifikation. Auch alle anderen wichtigen Ansätze zur Verifikation von Protokollspezifikationen wie z.B. [139], [41], [34] und [118] verwenden diese Annahme. Sie ist essentiell, um überhaupt Sicherheitseigenschaften auf abstrakter Ebene verifizieren zu können. Mit einem Angreifermodell, das kryptographische Nachrichten (ob bewusst oder unbewusst) fälschen könnte (wenn z.B. Kollisionen beim Verschlüsseln oder beim Hashen auftreten könnten), würden sicherlich keine Sicherheitseigenschaften zeigbar sein. Die Sicherheit eines Protokolls, das z.B. ein Passwort verschlüsselt überträgt und dieses zur Zugangssteuerung für eine beliebige Ressource verwendet, kann natürlich nur gewährleistet werden, wenn ein Angreifer ohne Wissen über das Passwort eine entsprechende verschlüsselte Nachricht, die beim Entschlüsseln das Passwort ergibt, *nicht* erzeugen kann.

Zur Verdeutlichung geben wir dazu im Folgenden einen Teil der Axiomatisierung aus der

PROSECCO Ebene für die decrypt : key × document → document Funktion an, die das Entschlüsseln eines Dokumentes mit einem Schlüssel spezifiziert. Dazu ist noch wichtig, dass die PROSECCO Ebene zwischen zwei verschiedenen Schlüsseltypen unterscheidet: symmetrische und asymmetrische Schlüssel zur Abbildung der beiden bekannten Kryptographie-Arten. Letztere zerfallen in private und öffentliche Schlüssel. Dabei wird zusätzlich davon ausgegangen, dass jeder Schlüssel anhand seines Wertes eindeutig einer der drei Mengen zugeordnet werden kann (in der Realität ist dies durch die entsprechenden Schlüsselkodierungen als Folge von bytes wie z.B. PKCS1 möglich). Dazu spezifizieren wir die drei Prädikate is_sessionkey : key für symmetrische Schlüssel, is_pubkey : key für öffentliche asymmetrische Schlüssel und is_privkey : key für private asymmetrische Schlüssel. Für jeden Schlüssel gilt damit genau eines der drei Prädikate. Damit spezifizieren wir nun als Beispiel das Entschlüsseln eines symmetrisch verschlüsselten EncDoc:

SPEZIFIKATION:

decrypt-sym-ok:
is_sessionkey(key) → decrypt(key, encdoc(key, doc)) = doc

decrypt-sym-fail:
is_sessionkey(key) ∧ key ≠ key_0 → is-intdoc(decrypt(key, encdoc(key_0, doc)))

Das Ergebnis des Entschlüsseln eines symmetrisch verschlüsselten EncDocs mit einem falschen Schlüssel (key ≠ key_0) ist also ein beliebiges IntDoc - die decrypt Funktion ist hier also unterspezifiziert. Dies bildet in der PROSECCO Ebene den Sachverhalt ab, dass nicht vorhersagbar ist, welcher Wert bei einer solchen Entschlüsselung z.B. bei Einsatz von 3DES Kryptographie entstehen würde. Ein in diesem Sinne "zufälliges" IntDoc kommt diesem Umstand am nächsten.

Eine kryptographische Bibliothek, die die decrypt Funktion in diesem Sinne umsetzen würde, ist mit Algorithmen wie DES oder RSA nicht implementierbar. Dies liegt daran, dass obige Definition die Injektivität der Verschlüsselung durch den Konstruktor EncDoc : key × document → Document voraussetzt, also darauf beruht, dass alle EncDoc(k,d) und EncDoc(k',d') bei Verschiedenheit von k zu k' oder von d zu d' auch verschieden sind. In der Realität ist lediglich die Abbildung $encrypt_k$: plaintext → ciphertext bei fixem Schlüssel k injektiv. Z.B. bei DES gibt es zu jedem Schlüssel k auch einen Schlüssel k', so dass $encrypt_k$(p) = $encrypt_{k'}$(p). Dies folgt aus der Tatsache, dass bei DES die Mächtigkeit des Definitionsbereichs und des Wertebereichs gleich ist, Verschlüsselung bildet bei fixem Schlüssel 8 byte Klartext injektiv auf 8 byte Ciphertext ab. Damit muss also für jedes Paar aus Plaintext und Ciphertext bezüglich der Verschlüsselung mit einem bestimmten Schlüssel mindestens ein zweiter Schlüssel existieren, der einen (anderen) Plaintext auf den gleichen Ciphertext abbildet.

Damit ist hier zunächst ohne Weiteres keine korrekte Verfeinerung möglich. Für eine reale Implementierung ist somit lediglich eine Aussage über die Komplexität der eingesetzten Algorithmen möglich. Damit lässt sich schließlich nur eine probabilistische Aussage bzgl. der Sicherheit der Algorithmen in der Realität angeben. Wir sind im Rahmen dieser Arbeit allerdings nicht in der wahrscheinlichkeitstheoretisch sicheren Implementierung der kryptographischen Operationen selbst interessiert, sondern an der Korrektheit des *Designs* des Protokolls und

der Korrektheit der Umsetzung in der Implementierung selbigens. Dabei nehmen wir kryptographische Algorithmen als sicher an, d.h. wir abstrahieren von der theoretischen Möglichkeit z.B. von Kollisionen bei der Verschlüsselung oder beim Hashing auch in der Implementierungsebene. Daher übertragen wir für unseren Ansatz die Annahme der perfekten Kryptographie auf die Implementierungsebene. Dabei wird also z.B. die konkrete Implementierung obiger abstrakter decrypt Funktion (und auch die anderen kryptographischen Operationen) in Java auf der Ebene von byte Folgen nicht mehr im Verfeinerungsbeweis betrachtet, sondern lediglich eine Axiomatisierung für ihr Verhalten angegeben. Natürlich ist dies eine Idealisierung der Realität. Wir müssen daher eine Argumentation finden, warum eine in diesem Sinne korrekte Implementierung nicht doch in der Realität Sicherheitslücken im Sinne eines auf Wahrscheinlichkeiten beruhendem Sicherheitsbegriff beinhalten kann. Im Folgenden werden daher die nötigen Annahmen und die resultierende Axiomatisierung erläutert.

Zunächst sollen aber noch die Java Klassen vorgestellt werden, die die abstrakten Dokumente mit Kryptographie implementieren.

9.2 Umsetzung der Kryptographie

Die Implementierung der kryptographischen Typen hängt natürlich stark von der Funktionsweise der kryptographischen Operationen in Java ab. Die Programmiersprache sieht deren Integration mittels einer Implementierung der Java Cryptographic Architecture (JCA) vor. Hierbei soll ein *Provider* implementiert werden, der ein generisches Interface zur Verschlüsselung (symmetrisch und asymmetrisch) zur Verfügung stellt. Dabei arbeiten die Kryptooperationen der JCA immer auf Arrays von bytes, da das generische Interface keine andere Repräsentation vorsieht. Dies gilt für Hashing, Verschlüsselung, Entschlüsselung und Signaturen. Damit muss also eine Implementierung der kryptographischen `Document` Typen ebenso mit `byte[ ]` Werten arbeiten. Im Folgenden betrachten wir lediglich Hashing und Verschlüsselung. Die Argumentation und auch die Axiomatisierung von Signaturoperationen ist identisch zu denen zur Verschlüsselung.

Die Klassen aus Abb. 9.1 implementieren damit die drei kryptographischen Dokumente HashDoc, EncDoc und SigDoc.

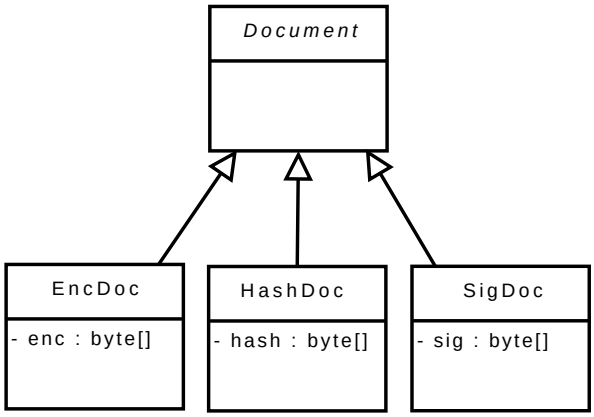


Abbildung 9.1: Klassenhierarchie für Dokumente mit Kryptographie

Alle drei Klassen beinhalten ein `byte[]`, das das Ergebnis der jeweiligen kryptographischen Operation speichert. Insbesondere enthalten konkrete Implementierungen z.B. der `EncDoc` Klasse natürlich *nicht* das zu verschlüsselnde Dokument und den Schlüssel, wie dies beim abstrakten EncDoc(key, doc) der Fall ist.

Wir müssen also vor Anwendung von kryptographischen Operationen die `Document` Objekte, die als Parameter fungieren, in eine byte Array Repräsentation umwandeln. Die entsprechende Implementierung z.B. für die `EncDoc` Klasse ist damit:

```
1  public class EncDoc extends Document {
2    private byte[] encrypted;
3    public EncDoc(Key k,Document d){
4      encrypted = Crypto.getCrypto().encrypt(k,d);
5    }
6  }
7
8  public class Crypto {
9    private static CryptoInterface crypto;
10   public static CryptoInterface getCrypto() {
11     return crypto;
12   }
13 }
14
15 public interface CryptoInterface {
16         public byte[] encrypt(Key key, Document doc);
17         public Document decrypt(Key key, Document doc);
18         public byte[] sign(Key key, Document doc);
19         public boolean verify(Key key, Document doc);
20         public byte[] hash(Document doc);
21 }
```

Beim Anlegen einer Instanz der Klassen `EncDoc` wird die Methode `encrypt` verwendet, um den `byte[ ]` Wert des `EncDocs` entsprechend zu initialisieren. Wir benutzen dazu die Klasse `Crypto`, um eine Instanz des Interfaces `CryptoInterface` über das statische Feld `.crypto` ständig zugreifbar zu haben. Tatsächlich benötigen wir damit eine ordentliche Instanz des Interfaces `CryptoInterface` in allen unseren Implementierungen in diesem Feld. Dies wird später auch Vorbedingung und Invariante aller Refinement Beweise sein. Eine konkrete Implementierung von `CryptoInterface` werden wir später in Kap. 10 im Rahmen der Kodierungsvorschrift für Dokumente in das Übertragungsformat vorstellen. Zum jetzigen Zeitpunkt geben wir lediglich gewünschte Eigenschaften der Methoden im `CryptoInterface` an. Ziel der Eigenschaften muss sein, das Verhalten der abstrakt spezifizierten Kryptographie auf die Implementierungsebene abzubilden. Warum dies gerechtfertigt ist, werden wir später in Kap. 9.3 diskutieren.

Zunächst betrachten wir die Entschlüsselung von Java Dokumenten mittels `decrypt`. Wir wählen für die Angabe der Eigenschaften der Funktion in `decrypt` folgende Vorgehensweise: Zuerst wandeln wir den konkreten Java Parameter für die Methode mittels java2doc in ein abstraktes Dokument (und entsprechend auch einen Key). Danach wenden wir die abstrakt spezifizierte Funktion für die Java Methode an, also hier das abstrakte decrypt. Daraus erhalten

wir ein abstraktes Ergebnisdokument resultdoc. Schließlich spezifizieren wir, dass auf dem resultierenden Speicher nach dem Methodenaufruf in Java die Abstraktionsfunktion java2doc ebenso das Dokument resultdoc liefert, solange kein Fehler aufgetreten ist. Ein Fehler ist dabei aufgetreten, wenn die entsprechenden Eingaben das Prädikat validDoc aus dem letzten Kapitel nicht erfüllen oder die Entschlüsselung nicht erfolgreich verlaufen ist. Letzteres erkennt man am Rückgabewert der abstrakten decrypt Funktion, wie oben spezifiziert (dieser ist vom Typ `IntDoc`). Sollte ein Fehler aufgetreten sein, sollen die Java Methoden `null` zurückliefern, was ja auch schon im vorherigen Kapitel den Fehlerfall signalisierte. Damit erhalten wir also für `decrypt` folgende zwei Axiome:

decrypt-ok:
st[.mode] = normal, .mode ∈ st,
validref("docs2store", r_1, EncDoc, st), $r_1 \neq$ jvmref,
validDoc(r_1, st),
validref("docs2store", r_2, CryptoInterface, st), $r_2 \neq$ jvmref,
validref("docs2store", r_3, Key, st), $r_3 \neq$ jvmref,
st[r_3 - .keyval] ≠ jvmref,
¬ is-intdoc(decrypt(java2doc(r_3, st_0), java2doc(r_1, st_0))),
st = st_0
⊢ ⟨st; `r = r2.decrypt(r3, r1);` ⟩
(java2doc(r_2, st) = decrypt(java2doc(r_3, st_0), java2doc(r_1, st_0))
∧ validDoc(r_2,st) ∧ $st_0 \subseteq$ st)

Das Axiom besagt: Falls die Eingabeparameter und die Invoking Expression gültige Java Pointerstrukturen ungleich null sind (validref(...) und ... ≠ jvmref Vorbedingungen) und auch der Ausführungsmodus im Speicher normal ist (also falls z.B. gerade keine Exception geworfen wird), dann liefert die Implementierung von `decrypt` die Pointerstruktur, die dem Ergebnis der Entschlüsselung auf den abstrakten Gegenstücken der Parametern entspricht (java2doc(r_2, st) = decrypt(java2doc(r_3, st_0), java2doc(r_1, st_0))), falls diese Entschlüsselung fehlerfrei funktioniert hat (¬ is-intdoc(decrypt(java2doc(r_3, st_0), java2doc(r_1, st_0)))).

Ebenso soll gelten:

decrypt-fail:
st[.mode] = normal, .mode ∈ st, init(st),
validref("docs2store", r_1, Document, st), $r_1 \neq$ jvmref,
(st[r_1 - .type] = EncDoc → st[r_1 - .enc] ≠ refval(jvmref))
validref("docs2store", r_2, CryptoInterface, st), $r_2 \neq$ jvmref,
validref("docs2store", r_3, Key, st), $r_3 \neq$ jvmref,
st[r_3 - .keyval] ≠ jvmref,
(is-intdoc(decrypt(java2doc(r_3, st_0), java2doc(r_1, st_0))),
∨ st[r_1 - .type] ≠ EncDoc ∨ ¬ validDoc(r_1, st))
st = st_0
⊢ ⟨st; `r = r2.encrypt(r3, r1);` ⟩

$(st_0 \subseteq st \wedge r = jvmref)$

Im Fehlerfall - also bei Vorliegen einer nicht erfolgreichen abstrakten Entschlüsselung (isintdoc(decrypt(java2doc(r_3, st_0), java2doc(r_1, st_0)))) oder bei ungültigen Eingabeparametern (st[r_1 - .type] $\neq$ EncDoc $\vee \neg$ validDoc(r_1, st)), liefert die Implementierung von `decrypt` den Wert jvmref und verändert den Speicher nicht.

Obige zwei Eigenschaften verwenden dabei java2doc für den `EncDoc` Typ. Dies haben wir bisher nicht spezifiziert. Hier müssen nun Java `EncDocs` mit beinhalteten byte Arrays auf abstrakte EncDocs abgebildet werden. Gleiches gilt für doc2java. Diese Funktion muss entsprechend abstrakte EncDocs auf Java `EncDocs` mit byte Arrays darin abbilden.

Dazu spezifizieren wir nun entsprechende Funktionen cryptbytes2doc : bytes $\rightarrow$ document und doc2cryptbytes : key $\times$ document $\rightarrow$ bytes. Die letztere Funktion spezifizieren wir dabei als injektive Abbildung. Dies ist nötig, da wir in der Lage sein müssen, aus einer gegebenen byte Folge den zur Verschlüsselung verwendeten Schlüssel wieder herauszurechnen, um anschließend die abstrakte decrypt Funktion zur Spezifikation des Ergebnisses verwenden zu können (wie oben angegeben). Wir spezifizieren dies, in dem wir angeben, dass cryptbytes2doc die Funktion doc2cryptbytes revertiert:

SPEZIFIKATION:

doc2cryptbytes-revert:
cryptbytes2doc(doc2cryptbytes(key, doc)) = encdoc(key, doc)

Daraus folgt sofort die Injektivität:

THEOREM:

doc2cryptbytes-inj:
doc2cryptbytes(key, doc) = doc2cryptbytes(key_0, doc_0) $\rightarrow$ key = key_0 $\wedge$ doc = doc_0

Für die Spezifikation von java2doc muss cryptbytes2doc verwendet werden. Für eine einfache und handhabbare Spezifikation fordern wir hier zusätzlich die Surjektivität von doc2cryptbytes. Damit müssen wir keine byte Folgen während cryptbytes2doc betrachten, die nicht Ergebnis von doc2cryptbytes sein können. Wir fordern also:

SPEZIFIKATION:

doc2cryptbytes-surj:
$\exists$ doc. doc2cryptbytes(doc) = bytes

Schließlich können wir damit java2doc spezifizieren, indem wir uns auf cryptbytes2doc abstützen:

SPEZIFIKATION:

java2doc-Encdoc:
$\neg$ cyclic("docs2store",r,st)
$\wedge$ r $\neq$ jvmref
$\wedge$ st[r - .type] = EncDoc
$\rightarrow$ java2doc(r, st) =
cryptbytes2doc(getbytearray(st[r - .value].refval, st))

Die abstrakte Entsprechung eines Java `EncDoc` Objektes ist demnach das Ergebnis von cryptbytes2doc auf dem gespeicherten (verschlüsselten) `byte[ ]` . Ebenso spezifizieren wir doc2java, indem wir das `byte[ ]` mittels doc2cryptbytes setzen:

SPEZIFIKATION:

doc2java-encdoc:
$r_1 + r_2$ = newref_list(2, st) $\rightarrow$
doc2java(EncDoc(key, doc), st) =
r_1
$\times$ addobj(r_1, EncDoc, .secret $\times$ refval(r_2),
addarray(r_2, byte_type, mkjavavalues(doc2cryptbytes(key,doc)), st))

Schließlich fehlen noch die Eigenschaften für die Verschlüsselung von Dokumenten mittels der Java Methode `encrypt`. Diese stützt sich nun natürlich auf doc2cryptbytes:

encrypt-ok:
st[.mode] = normal, .mode $\in$ st,
validref("docs2store", r_1, Document, st), $r_1 \neq$ jvmref,
validDoc(r_1, st),
validref("docs2store", r_2, CryptoInterface, st), $r_2 \neq$ jvmref,
validref("docs2store", r_3, Key, st), $r_3 \neq$ jvmref

$$
\begin{array}{l}
\quad \mathsf{st}[r_3\text{ - .keyval}] \neq \mathsf{jvmref},\\
\quad \mathsf{st} = \mathsf{st}_0\\
\vdash \langle \mathsf{st};\ \mathtt{r = r_2.encrypt(r_3,\ r_1);}\ \rangle\\
\qquad (\ \ \mathsf{addarray}(\mathsf{r}, \mathsf{byte\_type}, \mathsf{doc2cryptbytes}(\\
\qquad\qquad \mathsf{java2doc}(r_3,\mathsf{st}_0), \mathsf{java2doc}(r_1, \mathsf{st}_0)), \mathsf{st}_0) \subseteq \mathsf{st}\\
\qquad\ \ \wedge\ \mathsf{newref}(\mathsf{r},\mathsf{st}))
\end{array}
$$

Es wird (bei entsprechender Wohlgeformtheit der Eingaben) dem Speicher genau das Array von bytes hinzugefügt, das bzgl. doc2cryptbytes dem abstrakten Gegenstück zu den Eingabeparametern in Java entspricht.

Ebenso soll für nicht-wohlgeformte Eingabedokumente aus Symmetriegründen gelten, dass die Verschlüsselungsoperation diese aussortiert und einen Fehler meldet:

encrypt-fail:

$$
\begin{array}{l}
\quad \mathsf{st}[\text{.mode}] = \mathsf{normal},\ \text{.mode} \in \mathsf{st},\ \mathsf{init}(\mathsf{st}),\\
\quad \mathsf{validref}(\text{"docs2store"}, r_1, \mathsf{Document}, \mathsf{st}),\ r_1 \neq \mathsf{jvmref},\\
\quad \neg\ \mathsf{validDoc}(r_1, \mathsf{st}),\\
\quad \mathsf{validref}(\text{"docs2store"}, r_2, \mathsf{CryptoInterface}, \mathsf{st}),\ r_2 \neq \mathsf{jvmref},\\
\quad \mathsf{validref}(\text{"docs2store"}, r_3, \mathsf{Key}, \mathsf{st}),\ r_3 \neq \mathsf{jvmref}\\
\quad \mathsf{st}[r_3\text{ - .keyval}] \neq \mathsf{jvmref},\\
\quad \mathsf{st} = \mathsf{st}_0\\
\vdash \langle \mathsf{st};\ \mathtt{r = r_2.encrypt(r_3,\ r_1);}\ \rangle\\
\qquad (\mathsf{st}_0 \subseteq \mathsf{st} \wedge \mathsf{r} = \mathsf{jvmref})
\end{array}
$$

Damit ist die Behandlung der Ver- und Entschlüsselung abgeschlossen.

Schließlich müssen wir noch analog die Hashing-Funktion betrachten. Auch hier gilt wieder, dass die perfekte Kryptographie die Injektivität des Hashens fordert. Für eine korrekte Verfeinerungsoperation übertragen wir diese Annahme auf die Implementierung. Damit gehen wir völlig analog zur Verschlüsselung vor und spezifizieren eine Bijektion doc2hashbytes : document → bytes mit Umkehrfunktion hashbytes2doc : bytes → document

SPEZIFIKATION:

hash-revert:

$$\mathsf{hashbytes2doc}(\mathsf{doc2hashbytes}(\mathsf{doc})) = \mathsf{doc}$$

hash-surj:

$$\exists\ \mathsf{doc}.\ \mathsf{doc2hashbytes}(\mathsf{doc}) = \mathsf{bytes}$$

Auch hier folgt die Injektivität:

THEOREM:

hash-inj:
doc2hashbytes(doc) = doc2hashbytes(doc_0) $\rightarrow$ doc = doc_0

Analog zu oben spezifizieren wir daher java2doc und doc2java:

SPEZIFIKATION:

java2doc-hashdoc:
$\neg$ cyclic("docs2store",r,st)
$\wedge$ r $\neq$ jvmref
$\wedge$ st[r - .type] = HashDoc
$\rightarrow$ java2doc(r, st) =
HashDoc(hashbytes2doc(getbytearray(st[r - .value].refval, st)))

doc2java-hashdoc:
r_1 + r_2 = newref_list(2, st)
$\rightarrow$ doc2java(HashDoc(doc), st) =
r_1
$\times$ addobj(r_1, HashDoc, .secret $\times$ refval(r_2),
addarray(r_2, byte_type, mkjavavalues(doc2hashbytes(doc)), st))

Schließlich fordern wir für die Hashing Methode in Java:

hash-ok:
st[.mode] = normal, .mode $\in$ st,
validref("docs2store", r_1, Document, st), $r_1 \neq$ jvmref,
validDoc(r_1, st),
validref("docs2store", r_2, CryptoInterface, st), $r_2 \neq$ jvmref,
st = st_0
$\vdash \langle$st; `r = r2.hash(r1);` $\rangle$
(addarray(r, byte_type, doc2hashbytes(java2doc(r_1, st_0)), st_0) $\subseteq$ st
$\wedge$ newref(r,st))

hash-fail:
st[.mode] = normal, .mode ∈ st,
validref("docs2store", r_1, Document, st), $r_1 \neq$ jvmref,
¬ validDoc(r_1, st),
validref("docs2store", r_2, CryptoInterface, st), $r_2 \neq$ jvmref,
st = st_0
⊢ ⟨st; `r = r2.hash(r1);` ⟩
($st_0 \subseteq$ st ∧ r = jvmref)

9.3 Diskussion: Perfekte Kryptographie im Refinement

Für eine korrekte Verfeinerung ist die Übernahme der Annahme der perfekten Kryptographie auf die Ebene der Implementierung wesentlich. Schließlich soll jeder Schritt der Implementierung einem Schritt der abstrakten Spezifikation entsprechen.

In der Realität erfüllt natürlich keine der genannten Operationen die Forderungen der perfekten Kryptographie. Insbesondere sind Hashfunktionen und auch Verschlüsselungsoperationen für beliebige Schlüssel / Werte Paarte *nicht* injektiv. Intuitiv können wir also durch die Verfeinerung nur *Korrektheit modulo dem Auftreten von zufälligen Kollisionen bei der Kryptographie* beweisen. Wie rechtfertigen wir also die Verwendung der Annahmen der perfekten Kryptographie auch auf der Implementierungsebene?

Das Schlüsselargument zur Beantwortung dieser Frage ist, dass für eine Instanz einer realen Implementierung einer Anwendung, die Kryptographie verwendet, die Wahrscheinlichkeit des Auftretens von *konkreten* Eingaben, die die Annahmen der perfekten Kryptographie verletzen (also z.B. zwei Klartexte, die nach Hashen eine Kollision erzeugen) so verschwindend gering ist, dass sie ignoriert werden kann.

Für eine der Realität entsprechende Axiomatisierung dürfte z.B. die Funktion doc2cryptbytes nicht injektiv sein. Real treten allerdings nicht alle Eingaben für diese Funktion in einer Anwendung auf, sondern nur eine endliche Teilmenge. Wir können auf Grund der probabilistischen Eigenschaften der Kryptoalgorithmen damit annehmen, dass die Gründe für die Nicht-Injektivität der Funktion nicht in dieser beschränkten Menge von Eingaben zu suchen sind. Damit ist also die Einschränkung der realen kryptographischen Funktionen (wie z.B. Verschlüsselung) auf eine Teilmenge von Eingaben, bei denen alle möglichen Ergebnisse disjunkt sind, ein Modell der Axiome, die wir für doc2cryptbytes angegeben haben, wenn wir auch dort Definitions- und Wertebereich entsprechend einschränken.

Ebenso sind die realen Kryptofunktionen nicht surjektiv. Es kann also byte Arrays geben, die nicht Ergebnis einer Verschlüsselungs- oder Hashfunktion sind. Lediglich dürfen wir (auf Grund der Injektivität) die Länge der Ausgabe nicht beschränken. Damit muss unsere Implementierung der Protokollfunktionalität also auch mit potentiell "zu langen" oder "zu kurzen" `byte[]` Werten zurechtkommen. Würde man z.B. SHA-1 als Hashalgorithmus einsetzen, würden alle Hashwerte eine Länge von 20 byte haben (wiederum mit der Annahme, das bei den real vorkommenden Eingaben keine Kollisionen vorkommen). Ein längerer oder kürzerer Wert könnte real niemals Ergebnis von SHA-1 sein. Verifizieren wir die Protokollfunktionalität allerdings ohne eine solche Einschränkung (schließlich haben wir die Länge von doc2hashbytes(doc) nicht

spezifiziert), kann real auch nicht "schlimmes" passieren, wenn wir SHA-1 verwenden (unter der Annahme, dass die tatsächlich verwendeten Werte keine Kollisionen auf diesem 20 byte großen Wertebereich produzieren).

Ein weiteres Kriterium, das berücksichtigt werden muss, wenn Surjektivität gefordert wird, ist die Tatsache, dass man damit dem Angreifer die Möglichkeit nimmt, Eingaben an das Programm zu senden, die nicht Ergebnis einer Kryptofunktion sein können. Dies ist allerdings auch unproblematisch, da dies nur bedeutet, dass er keine *zufälligen* Eingaben versenden kann. Wichtig ist nur, dass er die Möglichkeit besitzt, eine reale Eingabe zu senden, die zu allen zufälligen Eingaben bezogen auf das Verhalten unserer Implementierung äquivalent ist. Dies ist z.B. mit HashDoc($\perp$) bzw. EncDoc(key, $\perp$) bei beliebigem key immer gegeben. Beide Dokumente kann der Angreifer immer erzeugen und die Spezifikationen der Operationen "vergleichen" (HashDocs) und "entschlüsseln" (EncDocs) bezüglich beider Dokumente sind äquivalent zu dem Verhalten, was in der Realität von zufälligen Bytes gefordert wird - nämlich der Unmöglichkeit, durch das Senden von Zufallszahlen plötzlich richtige und sinnvolle Ergebnisse nach Anwendung einer Entschlüsselung oder einer Hashing Funktion zu bekommen.

Zusammen rechtfertigen diese Erläuterungen unsere Axiomatisierung.

9.4 Spezifikation von Wohlgeformtheit für kryptographische Typen

Mit obigen Wandlungsfunktionen für die kryptographischen Operationen benötigen wir keine zusätzlichen Annahmen für die Spezifikation des validDoc Prädikates für die kryptographischen Typen. Lediglich ein `null` Wert für die entsprechenden `byte[]` Felder muss ausgeschlossen werden (dieser kann nicht Ergebnis einer der Funktionen doc2hashbytes oder doc2cryptbytes sein). Auch die Forderung nach eindeutig kürzestmöglicher Zahlenrepräsentation in den Werten der byte Arrays muss bei den kryptographischen Typen nicht entsprochen werden (auch solche Arrays können schließlich das Ergebnis von Verschlüsselung oder Hashing sein).

Daher spezifizieren wir einfach für `EncDoc`:

SPEZIFIKATION:

validDoc-EncDoc:

$$
\begin{aligned}
&\quad r \neq \text{jvmref} \\
&\wedge \neg\ \text{cyclic}(r,st) \\
&\wedge\ st[r\text{ - .type}] = \text{EncDoc} \\
\rightarrow (&\quad \text{validDoc}(r, st) \\
&\leftrightarrow st[r\text{ - .enc}] \neq \text{refval}(\text{jvmref})
\end{aligned}
$$

und analog für `HashDoc` und `SigDoc`:

SPEZIFIKATION:

validDoc-HashDoc:

$$
\begin{aligned}
& r \neq \text{jvmref} \\
\wedge\ & \neg\ \text{cyclic(r,st)} \\
\wedge\ & \text{st[r - .type]} = \text{HashDoc} \\
\rightarrow\ & (\quad \text{validDoc(r, st)} \\
\leftrightarrow\ & \text{st[r - .hash]} \neq \text{refval(jvmref)}
\end{aligned}
$$

SPEZIFIKATION:

validDoc-SigDoc:

$$
\begin{aligned}
& r \neq \text{jvmref} \\
\wedge\ & \neg\ \text{cyclic(r,st)} \\
\wedge\ & \text{st[r - .type]} = \text{SigDoc} \\
\rightarrow\ & (\quad \text{validDoc(r, st)} \\
\leftrightarrow\ & \text{st[r - .sig]} \neq \text{refval(jvmref)}
\end{aligned}
$$

Mit dieser Spezifikation und den obigen Eigenschaften von z.B. doc2hashbytes bleiben auch die Eigenschaften von validDoc und die Charakterisierung durch isAbstractDoc gültig.

Kapitel 10

Eine verifizierte Laufzeitumgebung

In diesem Kapitel wird eine exemplarische Java Implementierung für eine Transformationsschicht gegeben. Die Transformationsschicht erfüllt dabei die genau die gewünschten Eigenschaften von `send` und `receive`, die in Kapitel 6 und 8 bereits eingeführt wurden um die Verfeinerung korrekt zeigen zu können. Damit ist sichergestellt, dass eine Verfeinerung der bisherigen konkreten Implementierungsebene (in der lediglich Ergebnisse von doc2java als Eingaben betrachtet werden) zu einer Ebene, die die Transformationsschicht benutzt (und damit beliebige Eingaben im jeweiligen Übertragungsformat berücksichtigt), gezeigt werden kann. Fehlerhafte Eingaben, die nur in der Realität, aber nicht in der abstrakten Spezifikation möglich sind, werden generisch aussortiert und können daher nicht zu Fehlverhalten führen. Die in diesem Kapitel beschriebenen Arbeiten wurden in [62] publiziert.

10.1 Das Kodierungsschema

Die Transformationsschicht, die in diesem Kapitel vorgestellt wird, hat die Aufgabe, die Übertragung von Nachrichten, die aus Instanzen der Dokumentenklassen des vorherigen Kapitels aufgebaut sind, in ein Format zu kodieren, das tatsächlich zur Übertragung über ein Netzwerk verwendet werden kann. In Frage kämen hier z.B. Kodierungen in Strings (wie XML) oder Folgen von bytes. Wir wählen hier das byte Array Format als Übertragungsformat, nicht zuletzt weil auch die kryptographischen Funktionen auf Folgen von bytes arbeiten. Damit können wir somit auch Verschlüsselung durch Anwendung der Kodierungsfunktion und anschließendem Verschlüsseln mit einer Kryptographieimplementierung umsetzen. Dies wird, wie wir später sehen werden, auch unsere Annahme an die Kryptofunktionen `encrypt` und `decrypt` aus dem letzten Kapitel nochmals untermauern. Im Folgenden beschränken wir uns in der Darstellung daher auf byte Arrays als Übertragungsformat. Die Konzepte bleiben dennoch auch für andere Formate analog anwendbar.

Das Kodierungsschema, das uns hier als Beispiel dient, ist angelehnt an die Basic Encoding Rules der ASN-1 Kodierung [51]. Wir bilden unsere Dokumente dabei nach folgendem Schema ab:

- Jeder Dokumententyp wird durch ein eindeutiges Typ-Byte repräsentiert.
- Jedes Dokument wird in eine Folge von bytes abgebildet, wobei gilt:
  - das erste byte kennzeichnet den Typ des Dokumentes
  - das zweite bis fünfte byte geben die Länge des Wertes des Dokumentes an
  - die restlichen bytes sind der Wert des Dokumentes

Dabei ist der Wert von nicht-rekursiven Dokumenten der jeweils innerste enthaltene int Wert als Folge von bytes. Beim Doclist Typ ist der Wert die Sequenz der Kodierungen der Listeninhalte und die Länge ist entsprechend die Länge dieser Sequenz.

Wir definieren folgende Typ-Byte Konstanten[1]:

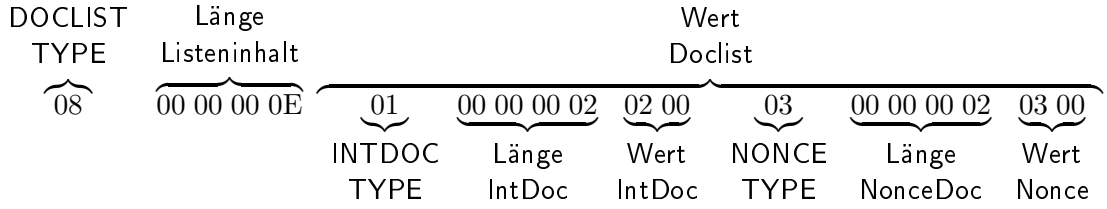
SPEZIFIKATION:

```
BOTTOMDOCTYPE = i2b(0)
INTDOCTYPE = i2b(1)
KEYDOCTYPE = i2b(2)
NONCEDOCTYPE = i2b(3)
SECRETDOCTYPE = i2b(4)
HASHDOCTYPE = i2b(5)
ENCDOCTYPE = i2b(6)
SIGDOCTYPE = i2b(7)
DOCLISTTYPE = i2b(8)
```

Wir illustrieren die Kodierungsregeln am Beispiel. Das Dokument

```
Doclist(IntDoc(512) + NonceDoc(mkNonce(768)) + [ ])
```

wird kodiert in die byte Folge:



Eine formale Definition der Kodierung folgt unten in Kap. 10.3.

[1] i2b(i) wandelt einen int Wert i in einen entsprechenden byte Wert um.

10.2 Motivation und Diskussion

Zur Motivation dieses Kapitels folgt nun zunächst eine Diskussion der Fragestellung, warum eine Transformationsschicht (die dazu generisch für die vorgestellten `Document` Typen implementiert ist) sinnvoll für die Implementierung und den Korrektheitsnachweis der Verfeinerung ist. Warum rechnet man nicht einfach gleich auf byte Arrays in der Implementierung und definiert doc2java entsprechend als Abbildung von den abstrakten Dokumenten in die byte Arrays?

Eine erste Antwort auf diese Frage wurde bereits in Kap. 8.1 gegeben: Die Typinformationen für die Eingaben der Implementierungsebene sind wesentlich für die Korrektheit einer Verfeinerung. Doch dafür sind nicht zwingend Klassen notwendig. Die oben vorgestellte Kodierungsvorschrift beinhaltet schließlich die Typinformationen. Tatsächlich könnten wir also unsere Protokollimplementierung direkt auf diesem Format umsetzen.

Betrachten wir zur Illustration wieder das Beispiel des vorangegangenen Kapitels 6.2: die Passwort-Überprüfung. Zur Wiederholung nochmals die abstrakte Spezifikation:

```
CHECKPASSWORD(agent, password; inputs, checkok)
  if (inputs(agent)(1) ≠ [ ]) then {
   indoc = inputs(agent)(1).first;
   inputs(agent)(1) := inputs(agent)(1).rest;
   if (isSecretDoc(indoc)) then {
      if (indoc.secret == password(agent)) then checkok(agent) := true
      else checkok(agent) := true }
```

Wir implementieren nun die Passwort-Überprüfung mit obigem Übertragungsformat. Die Methode `checkPIN` soll dabei einfach überprüfen, ob die Eingabe (die zuvor über das Netzwerk empfangen wurde), einem SecretDoc mit dem richtigen Passwort entspricht. Falls ja, soll die Methode `true` zurückliefern, ansonsten `false`. Vereinfachend nehmen wir hier eine kleine Modifikation der oben vorgestellten Kodierung an, die lediglich ein byte zur Angabe der Länge des Inhalts des Dokuments besitzt. Die Implementierung sieht damit wie folgt aus:

```
1  public class PasswordCheck {
2    private static PasswordCheck theinstance;
3
4    public boolean checkok;
5    public byte[] password;
6    public CommInterface comm;
7
8    public PasswordCheck(CommInterface c, byte[] password){
9      this.comm = c;
10     this.password = password;
11   }
12
13   public static void checkPassword(){
```

```
14      byte[] input = comm.receive();
15      if(checkPin(input)) checkok = true;
16      else checkok = false;
17    }
18
19    public boolean checkPin(byte[] input){
20      if(input != null &&       // not null
21         input.length > 2 &&    // no array index out of bounds
22         input[0] == SECRETDOCTYPE &&  // right type
23         input[1] == 4)         // right length
24      {
25         for(int i = 2; i < input.length; i++){
26            if(input[i] != password[i - 2]) return false;
27         }
28         return true;
29      }
30      return false;
31    }
32  }
```

Die Implementierung prüft zunächst, ob die Eingabe einen Wert ungleich `null` besitzt, um potentielle Übertragungsfehler abzufangen. Anschließend wird Typangabe und Längenangabe geprüft. Damit hierbei keine `ArrayIndexOutOfBoundsException` geworfen wird, muss die Länge der Eingabe mindestens 2 byte sein. Wir fordern gleich, dass sie echt größer als zwei byte ist, da schließlich auch noch ein Dokumenteninhalt vorhanden sein muss. Ist dies der Fall, wird der Typ geprüft (`SECRETDOCTYPE`). Ebenso erwarten wir genau 4 byte Inhalt. Auch dies wird geprüft. Ist alles somit in Ordnung, testen wir den Inhalt des Eingabearrays (also den Rest der Eingabe) byte für byte. Ist die erste Abweichung zum erwarteten Passwort gefunden, geben wir `false` zurück, ansonsten stimmt das Passwort und wir geben `true` zurück.

Die Simulationsrelation R und auch das TOSTORE Makro für den Verfeinerungsbeweis ist analog zu Kap. 6.2 spezifiziert. Lediglich die Funktion doc2javabytes aus dem selben Kapitel muss eine Kodierung der Eingabe nach obigem Schema vornehmen. Aus Einfachheitsgründen verzichten wir an dieser Stelle auf eine entsprechende Axiomatisierung.

Haben wir damit eine korrekte und sichere Implementierung angegeben? Auf den ersten Blick: ja. Falls die Eingaben nur korrekte Kodierungen sind, verhält sich das Programm analog zur Spezifikation. Nun haben wir in Kapitel 8 allerdings erläutert, dass in der Realität auch andere (nicht-wohlgeformte) Eingabe vorkommen können und wir daher auch diese betrachten müssen (s. dazu Kap. 8.1 und insbesondere Abb. 8.2). Ein entsprechendes TOSTORE Makro müsste also (wie in Abb. 8.2 gezeigt) die Eingabe $\perp$ indeterministisch auf eine der ungültigen byte Array Eingaben abbilden. Ist unsere Implementierung auch gegenüber solchen Eingaben, die also nicht der Struktur der Kodierung entsprechen, sicher?

Auch hier gilt auf den ersten Blick: ja. Schließlich überprüfen wir die Wohlgeformtheitskriterien des Dokumentes wie richtige Längen und richtiger Typ zu Beginn der Methode `checkInput`. Allerdings haben wir durch einen Implementierungsfehler einen Fall übersehen. Ein Angreifer könnte folgendes byte Array an obiges Programm senden:

```
0x04 0x04 0x01
```

Er schickt also ein Array, das das richtige Typ-Byte hat und auch die (scheinbar) richtige Längenangabe, lediglich der Inhalt des Dokumentes ist mit nur einem byte zu kurz. Ein solches Array kann nicht durch Kodierung eines abstrakten Documents entstanden sein, kann aber natürlich von einem realen Angreifer gesendet werden. Bei Eingabe dieses Dokumentes prüft unsere Implementierung allerdings *nur das erste byte des Passwortes*. Damit würde für jedes Passwort, das mit einer 0x01 beginnt, die Implementierung `true` zurückliefern und damit einen Angriff (bei Gleichverteilung der Passwörter) bei einem von 16 Fällen erlauben. Zwar würden wir in diesen Fällen auch die Korrektheit der Verfeinerung nicht zeigen können, dennoch zeigt das Beispiel, dass eine Implementierung, die auf einem Übertragungsformat wie den byte Arrays programmiert ist, sehr einfach subtile Programmierfehler enthalten kann.

Wesentlich schöner wäre es, wenn wir all diese Fehlerfälle durch Nicht-Wohlgeformtheit der Eingaben im Übertragungsformat *generisch* aussortieren könnten und per Definition alle Eingaben nur in *wohlgeformte* Rückgabewerte wandeln könnten. Damit wäre unsere Implementierung sehr viel einfacher möglich und damit weniger fehleranfällig. Eine entsprechende Charakterisierung für die dann nur möglichen wohlgeformten Rückgabewerte haben wir mit dem validDoc Prädikat des letzten Kapitels bereits gefunden. Tatsächlich werden wir in diesem Kapitel eine Implementierung für `receive` und `send` angeben, die die Gültigkeit der Eingaben prüft und nur Pointerstrukturen zurückliefert, die validDoc erfüllen. Die Implementierung obiger `checkPIN` Funktion kann damit einfach folgendermassen aussehen und dennoch die Korrektheit für alle Eingaben erhalten:

```
1  ... //Rest der Klasse wie oben
2
3  public void checkPassword(){
4    Document input = comm.receive();
5    if(checkPin(input)) checkok = true;
6    else checkok = false;
7  }
8
9  public boolean checkPin(Document input){
10   if(input instanceof SecretDoc &&
11      ByteArray.equals(((SecretDoc)input).getValue(), password))
12         return true;
13   }
14   return false;
15 }
```

Hier muss sich nun die Implementierung der Protokollfunktionalität *nicht* mehr um Überprüfung von Strukturkorrektheit der Eingabe kümmern, ein einfacher Typcheck reicht aus.

10.3 Eine Spezifikation für eine Transformationsschicht

Im Folgenden wird eine Spezifikation für die Transformationsschicht nach dem oben vorgestellten Kodierungsregeln angegeben. Dazu spezifizieren wir zwei Funktionen encode : Document → bytes und decode : bytes → Document. Darüber hinaus muss noch ein Gültigkeitsbegriff auf den byte Arrays definiert werden, der angibt, ob es sich bei einem vorliegenden Byte Array um die Kodierung eines abstrakten Dokumentes handelt, denn analog zu den Zusammenhängen zwischen Pointerstrukturen und abstrakten Dokumenten ist auch die Funktion encode zwar injektiv, aber nicht surjektiv.

10.3.1 Kodieren

Die Spezifikation der Kodierung erfolgt nach dem gleichen Schema wie das Wandeln von Dokumenten in eine Java Pointerstruktur: Wir geben für jeden Dokumententyp ein Axiom an, das die jeweilige Funktionalität beschreibt.

Am einfachsten ist dies für das leere Dokument ⊥:

SPEZIFIKATION:

encode-bottom:
encode(⊥) = BOTTOMDOCTYPE + fillsgn(4, int2bytes(0))

Das Kodieren eines ⊥ Dokumentes bedeutet folglich das Setzen des Typ-Bytes auf BOTTOMDOCTYPE und Angabe einer Länge von null weiteren bytes. Aus Symmetriegründen verwenden wir, wie für die anderen folgenden Dokumente, hierfür die Funktion fillsign(i,bys), die eine Folge von bytes bys vorzeichenerhaltend auf i bytes erweitert. Damit ist das Ergebnis mindestens i bytes lang ist und es gilt bytes2int(fillsign(i,bys)) = bytes2int(bys). Da im Weiteren das Dokument ⊥ keinen Inhalt besitzt, reicht dies hier zur Kodierung aus.

Die anderen nicht rekursiven Document Typen werden alle nach dem gleichen Schema behandelt. Wir illustrieren dies am IntDoc Typ:

SPEZIFIKATION:

encode-intdoc:
encode(IntDoc(i)) = INTDOCTYPE
+ fillsgn(4, int2bytes(n2i(# int2bytes(i))))
+ int2bytes(i)

Ein IntDoc(i) wird in eine byte Folge kodiert, die zunächst mit dem richtigen Typ-Byte INTDOCTYPE beginnt. Im Anschluss muss die Länge des Wertes i des Dokumentes folgen, wenn man diese in eine byte Folge abbildet (# int2bytes(i)). Die Längenfunktion # aus der KIV

Standardbibliothek liefert dabei eine natürliche Zahl zurück, die zuerst mittels n2i in einen int Wert umgewandelt wird. Schließlich wird dieser int Wert in eine byte Folge transformiert (int2bytes(...)) Das entstandene Array wird mittels fillsgn analog zu oben auf 4 byte aufgefüllt. Letztendlich folgt der eigentliche Wert des IntDocs ebenfalls als byte Folge.

Völlig analog erfolgen die Kodierungsvorschriften für die anderen Dokumententypen:

SPEZIFIKATION:

```
encode-keydoc:
encode(KeyDoc(key)) =     KEYDOCTYPE
                        + fillsgn(4, int2bytes(n2i(# int2bytes(key.int))))
                        + int2bytes(key.int)

encode-secretdoc:
encode(SecretDoc(nonce)) =    SECRETDOCTYPE
                            + fillsgn(4, int2bytes(n2i(# int2bytes(secret.int))))
                            + int2bytes(secret.int)

encode-noncedoc:
encode(NonceDoc(nonce)) =     NONCEDOCTYPE
                            + fillsgn(4, int2bytes(n2i(# int2bytes(nonce.int))))
                            + int2bytes(nonce.int)
```

Für die kryptographischen Typen verwenden wir die gleichen Konkretisierungsfunktionen wie bereits bei der Umwandlung in Pointerstrukturen:

SPEZIFIKATION:

```
encode-hashdoc:
encode(HashDoc(doc)) =    HASHDOCTYPE
                        + fillsgn(4, int2bytes(n2i(# doc2hashbytes(doc))))
                        + hash(doc)

encode-encdoc:
encode(EncDoc(key, doc)) =    ENCDOCTYPE
                            + fillsgn(4, int2bytes(n2i(# doc2cryptbytes(key, doc))))
                            + encrypt(key, doc)

encode-sigdoc:
encode(SigDoc(key, doc)) =    SIGDOCTYPE
                            + fillsgn(4, int2bytes(n2i(# doc2sigbytes(key, doc))))
                            + sign(key, doc)
```

Schließlich fehlt noch die Kodierung der Doclist. Hierzu definieren wir zunächst wieder eine einfache Rekursionsfunktion encode : documentlist → bytes auf Listen docs vom Typ Document. Das Ergebnis dieser Funktion definiert den Inhalt der kodierten Doclist, das Längenfeld ist einfach die Länge des Ergebnisses für die Listeninhalte:

SPEZIFIKATION:

encode-empty:
encode([ ]) = [ ]

encode-rec:
encode(doc + docs) = encode(doc) + encode(docs)

encode-doclist:
encode(Doclist(docs)) = DOCLISTTYPE
+ fillsgn(4, int2bytes(n2i(# encode(docs))))
+ encode(docs)

Damit ist die Spezifikation der Kodierungsfunktion vollständig.

10.3.2 Charakterisierung gültiger Kodierungen

Wir wollen später unsere Kodierung in einer Transformationsschicht implementieren. Insbesondere hat diese Transformationsschicht die Aufgabe, mit *allen* real möglichen Eingaben umgehen zu können. Dies ist nötig, da ein Angreifer in der Realität auch nicht wohlgeformte Kodierungen erzeugen kann. Zur formalen Behandlung müssen wir daher spezifizieren, welche Listen von bytes gültige Kodierungen in diesem Sinne darstellen. Gültige Kodierungen sollen dabei genau alle diejenigen Byte Listen sein, die Ergebnis der encode Funktion sein können, mit einer Ausnahme: Alle diejenigen Byte Listen, deren Gesamtlänge zu groß ist, sollen nicht als gültige Kodierungen gelten. Dies begründet sich wie folgt: Wir haben vier byte als Länge des Längenfeldes gewählt. Aus Gründen der Einfachheit haben wir damit eine Kodierung gewählt, die nicht sinnvoll mit beliebig langen Dokumenten umgehen kann. Wir können somit maximal $2^{31} - 1$ bytes als kodierten Inhalt eines Dokumentes sinnvoll dekodieren. Dies ist bezogen auf eine reale Anwendung keine Einschränkung, schließlich sind wir an Anwendungen mit kryptographischen Protokollen für mobile Endgeräte interessiert, bei denen so lange Nachrichten natürlich niemals real vorkommen werden. Im Weiteren liegt die Maximallänge für byte Arrays in Java sowieso bei $2^{31} - 1$ bytes. Formal sind jedoch natürlich auch Dokumente möglich, die nach obiger Kodierungsvorschrift auf längere byte Arrays abbilden. Die fillsgn Funktion verhält sich in diesem Fall wie die Identitätsfunktion. Beim Dekodieren ist es allerdings wesentlich, von einer Länge des Längenfelds von genau 4 bytes ausgehen zu können.

Zur Spezifikation dieser Anforderungen verwenden wir ein Prädikat validEncoding : bytes. Dies ist wie folgt spezifiziert:

Die leere Liste ist keine gültige Kodierung:

SPEZIFIKATION:

validEncoding-empty:
¬ validEncoding([ ])

Ebenso muss das erste Byte ein gültiges Typ Bytes sein. Damit können wir spezifizieren:

SPEZIFIKATION:

validEncoding-nodoctype:
bytes ≠ [ ]
∧ bytes[0] ≠ BOTTOMDOCTYPE
∧ bytes[0] ≠ INTDOCTYPE
∧ bytes[0] ≠ NONCEDOCTYPE
∧ bytes[0] ≠ SECRETDOCTYPE
∧ bytes[0] ≠ KEYDOCTYPE
∧ bytes[0] ≠ ENCDOCTYPE
∧ bytes[0] ≠ HASHDOCTYPE
∧ bytes[0] ≠ SIGDOCTYPE
∧ bytes[0] ≠ DOCLISTTYPE
→ ¬ validEncoding(bytes)

Listen mit Typ-Byte BOTTOMDOCTYPE ist nur dann gültig, wenn das Längenfeld den null-Wert hat und die Gesamtliste 5 bytes lang ist. Das Längenfeld selektieren wir mittels der Funktion sublist(i, j, bytes) aus der Byte Liste. Diese Funktion liefert die Teilliste beginnend bei Index i (0-basiert) mit Länge j zurück. Damit ist:

SPEZIFIKATION:

validEncoding-bottom:
bytes ≠ [ ]
∧ bytes[0] = BOTTOMDOCTYPE
→ (validEncoding(bytes)
↔ # bytes = 5
∧ bytes2int(sublist(1, 4, bytes)) = 0

Für die Dokumente, die Zahlenwerte enthalten (IntDoc, NonceDoc, KeyDoc und SecretDoc) liefert die obige Kodierung nur minimale Zahlendarstellungen - analog zur Umwandlung dieser Dokument in Pointerstrukturen. Damit ist also int2bytes(bytes2int(restn(5, bytes))) = restn(5,bytes)

ein Wohlgeformtheitskriterium für eine beliebige Kodierung bytes dieser Dokumententypen[2]. Zusätzlich muss die Länge der Liste mindestens 6 bytes betragen (1 byte Typ, 4 byte Länge und minimal 1 byte Wert). Das Längenfeld (bytes2int(sublist(1, 4, bytes)) muss die richtige Länge angeben, nämlich genau die Länge der Liste ohne den Header (# bytes - 5). Zusätzlich darf die Gesamtliste nicht zu lang sein. Insgesamt erhalten wir damit:

SPEZIFIKATION:

validEncoding-numberdocs:

$$
\begin{array}{l}
\quad \text{bytes} \neq [\,] \\
\wedge\ (\text{bytes}[0] = \text{INTDOCTYPE} \vee \text{bytes}[0] = \text{NONCEDOCTYPE} \vee \\
\quad\ \text{bytes}[0] = \text{SECRETDOCTYPE} \vee \text{bytes}[0] = \text{KEYDOCTYPE}) \\
\rightarrow (\quad \text{validEncoding(bytes)} \\
\quad \leftrightarrow \quad \#\,\text{bytes} < \text{i2n}(2^{31}) \\
\qquad \wedge\ \text{n2i}(\#\,\text{bytes}) - 5 = \text{bytes2int}(\text{sublist}(1, 4, \text{bytes})) \\
\qquad \wedge\ 6 \leq \#\,\text{bytes} \\
\qquad \wedge\ \text{int2bytes}(\text{bytes2int}(\text{restn}(5, \text{bytes}))) = \text{restn}(5, \text{bytes}))
\end{array}
$$

Für die kryptographischen Dokumente HashDoc, EncDoc und SigDoc muss keine Einschränkung bzgl. des Wertes gegeben sein, wir betrachten jedes byte Array potentiell als Ergebnis einer kryptographischen Operation (s. Kap. 9). Alle anderen Eigenschaften sind analog zu den Zahlen-Dokumenten:

SPEZIFIKATION:

validEncoding-cryptdocs:

$$
\begin{array}{l}
\quad \text{bytes} \neq [\,] \\
\wedge\ (\text{bytes}[0] = \text{ENCDOCTYPE} \vee \text{bytes}[0] = \text{HASHDOCTYPE} \vee \\
\quad\ \text{bytes}[0] = \text{SIGDOCTYPE}) \\
\rightarrow (\quad \text{validEncoding(bytes)} \\
\quad \leftrightarrow \quad \#\,\text{bytes} < \text{i2n}(2^{31}) \\
\qquad \wedge\ \text{n2i}(\#\,\text{bytes}) - 5 = \text{bytes2int}(\text{sublist}(1, 4, \text{bytes})) \\
\qquad \wedge\ 5 \leq \#\,\text{bytes})
\end{array}
$$

Damit fehlt nur noch die Spezifikation für Doclists. Hier müssen die ersten 5 byte die gleichen Kriterien erfüllen wie für andere Dokumente auch. Der Inhalt des Dokumentes muss eine dagegen eine Liste von gültigen Kodierungen darstellen. Wir lagern letzteres in ein eigenes Prädikat validListEncoding : bytes aus und spezifizieren zunächst für die Doclist:

[2] restn(i,bytes) liefert die Liste bytes um die ersten i Einträge verkürzt.

SPEZIFIKATION:

validEncoding-doclist:
bytes $\neq$ [ ]
$\wedge$ bytes[0] = DOCLISTTYPE
$\rightarrow$ (validEncoding(bytes)
$\leftrightarrow$ # bytes < i2n(2^{31})
$\wedge$ n2i(# bytes) - 5 = bytes2int(sublist(1, 4, bytes))
$\wedge$ 5 $\leq$ # bytes
$\wedge$ validListEncoding(bytes))

Eine Liste von gültigen Kodierungen kann eine leere Liste sein (nämlich bei Doclist([ ]) als Eingabe. Daher gilt:

SPEZIFIKATION:

validListEncoding-empty:
validListEncoding([ ])

Für nicht leere Listen gilt: Eine gültige Listenkodierung muss mindestens ein Element haben (also 5 $\leq$ # bytes) und darf gleichzeitig nicht zu lang sein (# bytes < i2n(2^{31})). Ebenso muss das Längenfeld des ersten enthaltenen Dokumentes (sublist(1, 4, bytes)) einen Wert enthalten, der nicht größer ist als die Gesamtlänge des Dokumentes. Des Weiteren muss das erste Dokument eine gültige Kodierung im Sinne von validEncoding sein und schließlich die Restliste ebenso eine gültige Listenkodierung sein. Damit erhalten wir:

SPEZIFIKATION:

validListEncoding-list:
bytes $\neq$ [ ]
$\rightarrow$ (validListEncoding(bytes)
$\leftrightarrow$ 5 $\leq$ # bytes
$\wedge$ # bytes < i2n(2^{31})
$\wedge$ i2n(bytes2int(sublist(1,4,bytes)) + 5) $\leq$ # bytes
$\wedge$ validEncoding(sublist(0, i2n(bytes2int(sublist(1,4,bytes))) + 5, bytes))
$\wedge$ validListEncoding(restn(i2n(bytes2int(sublist(1,4,bytes))) + 5, bytes))

10.3.3 Dekodieren

Neben der Kodierung müssen wir auch die Dekodierung spezifizieren. Die Dekodierung entscheidet anhand des Typ-Bytes, welches abstrakte Dokument zu konstruieren ist. Dabei müssen wir nur Eingaben richtig dekodieren, die gültige Kodierungen von Dokumenten in obigem Sinne sind. Für alle anderen Eingaben soll unsere spätere Implementierung einen Fehler signalisieren. In der Spezifikation der Dekodierung auf abstrakter Ebene lassen wir in diese Fälle unspezifiziert. Wieder gehen wir zur Spezifikation anhand der verschiedenen Typen vor.

Zunächst als einfachster Fall die Dekodierung des leeren Dokumentes $\bot$:

SPEZIFIKATION:

decode-bottom:
validEncoding(bytes)
$\wedge$ bytes[0] = EMPTYDOCTYPE
$\rightarrow$ decode(bytes) = $\bot$

Da wir, wie oben bereits bemerkt, validEncoding zur Dekodierung voraussetzen, ist eine solche ansonsten sehr offensive Spezifikation der Dekodierung zulässig. Wir müssen lediglich das Typ-Byte prüfen und können uns auf Grund von validEncoding auf die Wohlgeformtheit der restlichen Kodierung verlassen.

Entsprechend sind auch die Spezifikationen für die Dekodierung der restlichen nicht-rekursiven Dokumente recht einfach und stützen sich einfach auf die Umwandlung der byte Folgen hinter den Längen-Bytes in einen Integer Wert. Wir erhalten somit:

SPEZIFIKATION:

decode-intdoc:
validEncoding(bytes)
$\wedge$ bytes[0] = INTDOCTYPE
$\rightarrow$ decode(bytes) = intdoc(bytes2int(restn(5, bytes)))

decode-keydoc:
validEncoding(bytes)
$\wedge$ bytes[0] = KEYDOCTYPE
$\rightarrow$ decode(bytes) = keydoc(mkkey(bytes2int(restn(5, bytes))))

decode-secretdoc:
validEncoding(bytes)
$\wedge$ bytes[0] = SECRETDOCTYPE
$\rightarrow$ decode(bytes) = secretdoc(mksecret(bytes2int(restn(5, bytes))))

decode-noncedoc:
validEncoding(bytes)

```
   ∧ bytes[0] = NONCEDOCTYPE
→ decode(bytes) = noncedoc(mknonce(bytes2int(restn(5, bytes))))
```

Für die kryptographischen Dokumente greifen wir analog zur Kodierung nun auf die entsprechenden Umkehrfunktionen für die `byte[ ]` Abbildung zurück. Somit sind die Axiome für diese Typen:

SPEZIFIKATION:

```
decode-encdoc:
      validEncoding(bytes)
   ∧ bytes[0] = ENCDOCTYPE
→ decode(bytes) = cryptbytes2doc(restn(5, bytes))

decode-hashdoc:
      validEncoding(bytes)
   ∧ bytes[0] = HASHDOCTYPE
→ decode(bytes) = hashdoc(hashbytes2doc(restn(5, bytes)))

decode-sigdoc:
      validEncoding(bytes)
   ∧ bytes[0] = SIGDOCTYPE
→ decode(bytes) = sigbytes2doc(restn(5, bytes))
```

Schließlich benötigen wir noch für den Doclist Typ entsprechende rekursive Axiome. Auch hier setzen wir validListEncoding voraus. Die Dekodierung einer Listenkodierung erfolgt, indem zuerst das Präfix der Listenkodierung, das so lang ist wie es das erste Längenfeld in der Liste angibt, dekodiert wird und anschließend der Rest der Liste dekodiert wird. Die leere Bytefolge entspricht dabei der leeren Liste. Das Längenfeld einer nicht-leeren Liste ist bei wohlgeformter Kodierung gerade die sublist(1, 4, bytes). Damit erhalten wir also:

SPEZIFIKATION:

```
decode-doclist:
      validEncoding(bytes)
   ∧ bytes[0] = DOCLISTTYPE
→ decode(bytes) = doclist(decodelist(restn(5, bytes)))

decodelist-empty:
decodelist([ ]) = [ ]

decodelist-rec:
   bytes ≠ [ ] ∧ validListEncoding(bytes)
```

```
→ decodelist(bytes) =
            decode(firstn(5 + i2n(bytes2int(sublist(1, 4, bytes))), bytes))
          + decodelist(restn(5 + i2n(bytes2int(sublist(1, 4, bytes))), bytes))
```

In letzterem Axiom ist 5 + i2n(bytes2int(sublist(1, 4, bytes))) gerade die Länge der Dokumentenkodierung in der Liste bytes. Damit ist also firstn(..., bytes) mit genau diesem Parameter genau diese erste Teil-Liste und entsprechend restn(..., bytes) die hinteren bytes, die nicht zum ersten Element gehören.

10.3.4 Korrektheit der Spezifikation

Da wir nur byte Listen sinnvoll dekodieren können, deren Gesamtlänge kleiner 2^{31} bytes ist, müssen wir auch für die Formulierung von Korrektheitseigenschaften der Kodierung die abstrakten Eingabedokumente nur auf solche beschränken, deren Kodierung nicht zu lang wird. Wir spezifizieren dazu einfach ein Prädikat codable : Document:

SPEZIFIKATION:

codable:
codable(doc) $\leftrightarrow$ # encode(doc) $< 2^{31}$

Wir beweisen nun im Folgenden die Korrektheit unserer Kodierung nur für die Dokumente, die codable erfüllen. Die Betrachtung dieser Dokumente reicht, wie initial erklärt, in der Realität sicherlich aus.

Die wesentlichen Korrektheitseigenschaften für die Spezifikation sind (jeweils unter der Voraussetzung, dass alle betrachteten Eingabedokumente codable erfüllen):

- encode ist die Umkehrfunktion von decode für alle Bytelisten, die validEncoding erfüllen
- decode ist die Umkehrfunktion von encode
- Das Prädikat validEncoding beschreibt genau die Bytelisten, die Ergebnis von encode sein können

Diese Eigenschaften können anhand der Spezifikation verifiziert werden. Es gilt:

THEOREM:

encode-decode:
validEncoding(bytes) → encode(decode(bytes)) = bytes

Die Funktionen decode und decodelist sind dabei wechselseitig rekursiv. Zum Beweis dieser Eigenschaft benötigen wir also auch die entsprechende Eigenschaft für Listenkodierungen, da diese im Fall der Doclist verwendet werden. Um dabei dann eine Induktionshypothese anwenden zu können, müssen wir also die Eigenschaft für Listenkodierungen und normale Dokumentenkodierungen gleichzeitig beweisen. Wir beweisen also obige Eigenschaft mittels:

THEOREM:

encode-decode-h:
(validEncoding(bytes) → encode(decode(bytes)) = bytes)
∧ (validListEncoding(bytes) → encode(decodelist(bytes)) = bytes)

Exemplarisch wollen wir den Beweis dieser Eigenschaft etwas erläutern. Alle formalen Beweise finden sich in der Spezifikation *TLV-encoding-spec* in der Web-Präsentation [49].

Zum Beweis dieser Eigenschaft verwenden wir nun Induktion über # bytes. Für die Korrektheitsaussage der Listenkodierung expandieren wir die Definitionen für decodelist. Im Fall einer nicht-leeren byte Liste verwenden wir dann die Induktionshypothese zweimal (einmal für den ersten Aufruf von decode und einmal für den rekursiven Aufruf von decodelist) und schließen so den Beweis. Für die Korrektheitsaussage der normalen Kodierung führen wir eine Fallunterscheidung über das erste byte der Liste durch. Wegen validEncoding-nodoctype (s. oben) kann hier nur eines der entsprechenden Typ-Bytes stehen. Wir expandieren jeweils die Dekodierungsfunktion anhand der entsprechenden Axiome und wenden anschließend die entsprechende Kodierungsfunktion an. Schließlich bleiben dann Aussagen über die Werte der Byte-Folgen übrig. Eine entsprechende Beweisverpflichtung für den IntDoc Typ ist etwa:

```
    5 ≤ # bytes
  ∧...
→   fillsgn(4, int2bytes(bytes2int(sublist(1, 4, INTDOCTYPE + bytes))))
  + restn(5, INTDOCTYPE + bytes) = bytes
```

Diese Beweisverpflichtungen folgen nun mittels Eigenschaften über fillsgn, int2bytes und bytes2int. Es gilt z.B.

$$\begin{aligned} &\#\,\mathsf{bytes} = \mathsf{n} \wedge \mathsf{bytes} \neq [\,] \\ \rightarrow\ &\mathsf{fillsgn}(\mathsf{n}, \mathsf{int2bytes}(\mathsf{bytes2int}(\mathsf{bytes}))) = \mathsf{bytes} \end{aligned}$$

Damit lässt sich z.B. obige Beweisverpflichtung schließen. Schließlich verbleibt der Fall, dass es sich um eine Kodierung einer Doclist handelt. Dann benötigen wir die verallgemeinerte Induktionshypothese für die enthaltene Listenkodierung. Damit lässt sich dann auch dieser Fall schließen.

Damit kommen wir zur zweiten anfangs genannten Korrektheitseigenschaft: decode ist die Umkehrfunktion von encode:

THEOREM:

decode-encode:
$\mathsf{codable}(\mathsf{doc}) \rightarrow \mathsf{decode}(\mathsf{encode}(\mathsf{doc})) = \mathsf{doc}$

Hier gilt das Gleiche wie bereits für die letzte Eigenschaft: Wir benötigen eine entsprechend allgemeinere Induktionshypothese, die die gleiche Aussage auch für Listen von Documents ausdrückt. Dann beweisen wir durch strukturelle Induktion über den Aufbau von doc. Die entsprechenden nicht-rekursiven Fälle folgen wie oben direkt aus den Definitionen der Funktionen encode und decode, wieder zusammen mit Hilfseigenschaften über die byte Listen. Der Listenfall lässt sich wieder mit der verallgemeinerten Induktionshypothese schließen. Wichtig für den Beweis ist dabei noch die Eigenschaft, dass sämtliche Ergebnisse der encode Funktion das validEncoding Prädikat erfüllen. Ohne eine solche Eigenschaft können wir die entsprechenden Axiome für die Dekodierung nicht anwenden, da diese die Wohlgeformtheit fordern. Wir benötigen also:

THEOREM:

validEncoding-encode:
$\mathsf{codable}(\mathsf{doc}) \rightarrow \mathsf{validEncoding}(\mathsf{encode}(\mathsf{doc}))$

Dies ist schon fast die dritte Korrektheitseigenschaft: Das Prädikat validEncoding beschreibt genau die Bytelisten, die Ergebnis von encode sein können. Dazu benötigen wir allerdings noch die Umkehrung obiger Aussage:

THEOREM:

validEncoding-exdoc-h:
validEncoding(bytes) $\rightarrow \exists$ doc. encode(doc) = bytes $\wedge$ codable(doc)

Diese Aussage folgt mit der ersten Korrektheitseigenschaft encode-decode, wenn wir für doc in obigem Existenzquantor decode(bytes) einsetzen. Wir erhalten dann lediglich noch die Beweisverpflichtung, dass validEncoding(bytes) $\rightarrow$ codable(decode(bytes)). Dies folgt allerdings trivial aus den Axiomen für validEncoding und der Tatsache, dass für bytes wegen encode-decode gilt encode(decode(bytes)) = bytes.

Damit erhalten wir zusammen die Äquivalenz:

THEOREM:

validEncoding-exdoc:
validEncoding(bytes) $\leftrightarrow \exists$ doc. encode(doc) = bytes $\wedge$ codable(doc)

Aus genannten Eigenschaften folgen nun direkt auch andere Eigenschaften, wie etwa die Injektivität von encode und decode. Es gilt:

THEOREM:

inj-encode:
codable(doc) $\wedge$ codable(doc_0)
$\rightarrow$ (encode(doc) = encode(doc_0) $\leftrightarrow$ doc = doc_0)

inj-decode:
validEncoding(bytes) $\wedge$ validEncoding(bytes_0)
$\rightarrow$ (decode(bytes) = decode(bytes_0) $\leftrightarrow$ bytes = bytes_0)

Durch Nachweis der in diesem Abschnitt genannten Eigenschaften haben wir somit sichergestellt, dass unsere Kodierung korrekt ist und für eine Übertragung der Dokumente verwendet werden kann. Insgesamt benötigten wir zum Beweis der Korrektheitseigenschaften auf abstrakter Ebene 51 Theoreme mit ca. 2000 Beweisschritten. Diese relativ hohe Zahl begründet sich hauptsächlich durch die Komplexität der Operationen auf den byte Listen. Es verbleibt nun, eine Implementierung für diese Kodierung anzugeben.

10.4 Eine Implementierung

10.4.1 Gewünschte Eigenschaften

Zur Programmierung der Kodierung müssen wir zunächst überlegen, welche Eigenschaften von unserer Implementierung sichergestellt werden sollen. Dazu ist die Betrachtungsweise aus Kapitel 8.1 wichtig, insbesondere die Zusammenhänge, die bereits in Abb. 8.2 dargestellt wurden: Wir möchten in der untersten Ebene sicherstellen, dass wirklich alle real möglichen Eingaben (die der Angreifer erzeugen kann), im Beweis berücksichtigt werden, um keine realen Angriffe zu übersehen. Insbesondere müssen wir also Eingaben betrachten, die *nicht* validEncoding erfüllen. Zusätzlich soll unsere Kodierungs- und Dekodierungsimplementierung in der Transformationsschicht aber auch nur wohlgeformte Dokumentenstrukturen in Java produzieren. Diese haben wir in Kap. 8.3 und 8.4 bereits durch validDoc charakterisiert. Damit erhalten wir zwei Beweisverpflichtungen für die Implementierungen von `encode` und `decode` jeweils für den Fall einer wohlgeformten Eingabe und für den Fall einer nicht-wohlgeformten Eingabe. Es sollte also gelten:

1. Jedes `byte[ ]` , das wohlgeformt im Sinne von validEncoding ist, wird dekodiert zu genau der Pointerstruktur, die umgewandelt mittels java2doc dem Ergebnis des abstrakten decode auf dem Array entspricht. Insbesondere erfüllen alle diese Ergebnisse validDoc.
2. Jedes `byte[ ]` , das nicht wohlgeformt im Sinne von validEncoding ist, wird von der Dekodierung erkannt und es wird `null` zurückgegeben
3. Jedes Java `Document` Objekt, das eine wohlgeformte Pointerstruktur im Sinne von validDoc ist, wird kodiert zu genau dem `byte[ ]` , welches Ergebnis des abstrakten encode auf dem Ergebnis von java2doc ist
4. Jedes Java `Document` Objekt, das nicht eine wohlgeformte Pointerstruktur im Sinne von validDoc ist, wird erkannt und nicht kodiert

Eigenschaft 4 ist dabei für die Zusammenhänge in Abbildung 8.2 eigentlich nicht wesentlich. Schließlich können wir bei jeder korrekten Protokollimplementierung zeigen, dass diese keine nicht-wohlgeformten Pointerstrukturen erzeugt. Im Sinne einer robusten Implementierung einer Transformationsschicht allerdings, die robust gegen Fehlbenutzung ist, sollten wir allerdings auch diese Eigenschaft fordern.

Wir werden die obigen Eigenschaften später formalisieren und beweisen. Zunächst betrachten wir jedoch die eigentliche Implementierung.

10.4.2 Umsetzung

Im Folgenden wird die Implementierung der obigen Kodierung vorgestellt. Zunächst setzen wir die zwei Funktionen `encode` und `decode` sowie das Prädikat `validEncoding` mittels Methoden in einer Klasse `Coding` um. Wir erhalten somit also Grundgerüst:

```
1  public class Coding{
2      private static final byte EMPTYDOCTYPE = (byte)0;
3      private static final byte INTDOCTYPE = (byte)1;
4      private static final byte KEYDOCTYPE = (byte)2;
5      private static final byte NONCEDOCTYPE = (byte)3;
6      private static final byte SECRETDOCTYPE = (byte)4;
7      private static final byte HASHDOCTYPE = (byte)5;
8      private static final byte ENCDOCTYPE = (byte)6;
9      private static final byte SIGDOCTYPE = (byte)7;
10     private static final byte DOCLISTTYPE = (byte)8;
11
12     ...
13
14     public  Document decode(byte[] b) { ... }
15     private boolean validEncoding(byte[] bytes) { ... }
16     public byte[] encode(Document d){ ... }
17 }
```

Betrachten wir daher zunächst die Dekodierung von byte Arrays. Die Dekodierung soll dabei für nicht wohlgeformte Eingaben einen Fehler signalisieren. Daher verwenden wir zunächst die `validEncoding` Methode, um die Eingabe zu prüfen. Deren Implementierung wird später erläutert. Bei positivem Testergebnis können wir im Anschluss an `validEncoding` von einer wohlgeformten Eingabe ausgehen. Damit wird die Dekodierung für die nicht-rekursiven Dokumente recht einfach. Wir müssen lediglich die Restliste hinter dem Typ-Byte und dem Längenfeld als Wert für die entsprechende Klasse verwenden. Damit haben wir eine einfache Fallunterscheidung über das Typ-Byte:

```
1  public  Document decode(byte[] b){
2   if(b != null && validEncoding(b)){
3     if(b[0] == INTDOCTYPE){
4      return new IntDoc(ByteArray.restn(5, b));
5     }
6     else if(b[0] == KEYDOCTYPE){
7      return new KeyDoc(new Key(ByteArray.restn(5, b);));
8     }
9     else if(b[0] == NONCEDOCTYPE){
10     return new NonceDoc(new Nonce(ByteArray.restn(5, b)));
11    }
12    else if(b[0] == SECRETDOCTYPE){
13     return new SecretDoc(ByteArray.restn(5, b));
14    }
15    else if(b[0] == HASHDOCTYPE){
```

```
16      return new HashDoc(ByteArray.restn(5, b));
17     }
18     else if(b[0] == ENCDOCTYPE){
19      return new EncDoc(ByteArray.restn(5, b));
20     }
21     else if(b[0] == SIGDOCTYPE){
22      return new SigDoc(ByteArray.restn(5, b));
23     }
24     else if(b[0] == DOCLISTTYPE){
25      return decodeList( ByteArray.restn(5, b) );
26     }
27    }
28    return null;
29  }
```

Die Bibliotheksfunktion `ByteArray.restn(n, b)` implementiert dabei das Verhalten der abstrakten `restn(n, bytes)` Funktion entsprechend auf Java dem Java `byte[ ]` Typ. Für die Dokumentenliste müssen wir eine Rekursion starten und schrittweise die einzelnen Listenelemente dekodieren. Dies wird wie folgt implementiert:

```
1  private Doclist decodeList(byte[] bytes){
2   if(bytes.length == 0)
3       return new Doclist();
4
5   int length = ByteArray.getInt(bytes, 1);
6
7   Document doc = decode(ByteArray.firstn(length + 5, bytes));
8   Doclist docs = decodeList(ByteArray.restn(length + 5, bytes));
9   return docs.cons(doc);
10 }
```

Ein leeres `byte[ ]` entspricht der leeren Liste, die gerade Ergebnis der Konstruktor-Aufrufs für `Doclist` ist. Im Falle einer nicht-leeren Liste müssen wir die Länge des ersten Eintrags bestimmen (`ByteArray.getInt(bytes, 1)` liefert gerade den `int` Wert im gegebenen `byte[ ]` beginnend bei Index 1). Schließlich werden rekursiv die `decode` und die `decodelist` Funktion aufgerufen und die Ergebnisse schließlich mittels `cons` verkettet. Die entsprechende `Doclist` Implementierung ist:

```
1  public class Doclist extends Document {
2   private Document[] docs;
3
4   public Doclist{
5    docs = new Document[0];
6   }
7
8   public Doclist cons(Document d) {
9    Document[] res = new Document[docs.length+1];
10   copyDocs(docs, 0, res, 1, docs.length);
11   res[0] = d;
```

```
12   return new Doclist(res);
13  }
14
15  public static void copyDocs(Document[] src,
16                              int srcOff,
17                              Document[] dest,
18                              int destOff,
19                              int len) {
20   for(int i = 0; i < len; i++)
21     dest[destOff + i] = src[srcOff + i];
22  }
23 }
```

Die `cons` Implementierung ist dabei nicht-destruktiv umgesetzt. Dies hat zwar einen höheren Speicherplatzbedarf, ermöglicht jedoch später eine einfachere Verifikation.

Schließlich benötigen wir eine Implementierung für `validEncoding`. Alle nicht-wohlgeformten Kodierungen müssen dabei erkannt erkannt werden. Aus Gründen der Darstellbarkeit findet sich die vollständige Implementierung für `validEncoding` in Anhang 13.1. Wir wollen die wichtigsten Ausschnitte hier betrachten. Für jedes kodierte Dokument `bytes` müssen die folgenden Eigenschaften gelten:

- `bytes` darf nicht den Wert `null` haben und nicht zu lang oder zu kurz sein

```
1  if( !(   bytes != null
2        && bytes.length <= 2147483647
3        && 6 <= bytes.length) )
4   return false;
```

- das Längenfeld muss die richtige Länge enthalten

```
1  int length = ByteArray.getInt(bytes, 1);
2  if(length != bytes.length - 5)
3   return false;
```

- als `byte[ ]` kodierte Zahlenwerte im Wert-Teil von `bytes` (hier beginnend bei Index `5`) müssen in minimaler Darstellung vorliegen

```
1  if( !noLeadingZeros(bytes, 5) )
2   return false;
```

  mit

```
1 private boolean noLeadingZeros(byte[] b, int offset){
2  if(b.length == 0) return false;
3  if(   b[offset] == (byte)0
4     && (offset + 1) != b.length
5     && b[offset + 1] >= (byte)0)
```

```
6       return false;
7    if(   b[offset] == (byte)-1
8       && (offset + 1) != b.length
9       && b[offset + 1] < (byte)0)
10       return false;
11   return true;
12  }
```

Ein `byte[ ]` der Länge 0 kann keine Zahl repräsentieren (kann nicht Ergebnis von int2bytes sein) und muss daher ausgeschlossen werden. Die beiden anschließenden `if` Tests überprüfen das Vorliegen von unnötigen führenden `00` bytes im ersten Fall bzw. bei negativen Zahlen führenden `FF` bytes im zweiten Fall.

Entsprechend der Rekursion beim Dekodieren muss natürlich auch hier für den Listentyp vorgegangen werden. Die entsprechende Implementierung findet sich in Anhang 13.1.

Damit fehlt noch die Implementierung für die Kodierung von `Document` Objekten. Wie in Kap. 10.4.1 bereits bemerkt, müsste diese Implementierung nicht fehlertolerant bezüglich nicht wohlgeformten Eingaben sein, da die Kodierung nur Eingaben erhält, die aus der Implementierung der Protokollfunktionalität eines ehrlichen Agenten stammen. Damit könnte man immer zeigen, dass diese Eingaben wohlgeformt sind, denn ordentliche Implementierungen (und nur diese verifizieren wir ja) produzieren keine nicht wohlgeformten Eingaben. Damit könnten wir die Wohlgeformtheit der Eingaben der Java Kodierungsfunktion eigentlich immer zeigen. Allerdings möchten wir hier eine Implementierung angeben, die allgemein als Bibliothek für Security Protokolle verwendbar ist. Ein Programmierer einer Protokollfunktionalität hätte sicherlich die Erwartungshaltung, dass eine solche Bibliothek auch robust gegenüber Fehlbenutzung ist. Daher prüfen wir in unserer Implementierung die Wohlgeformtheit der Eingabedokumente im Sinne von validDoc und melden entsprechend bei Bedarf einen Fehler durch Rückgabe von `null`.

Im Gegensatz zur Implementierung der Dekodierung, die eine eigene Methode zur Prüfung der Wohlgeformtheit einsetzt, prüfen wir beim Kodieren diese On-the-fly während des Entlanglaufens durch die Struktur. Auch für die Kodierung findet sich die vollständige Implementierung in Anhang 13.1. Wir stellen auch hier der Lesbarkeit wegen nur das Gerüst und die wichtigsten Bestandteile der Implementierung vor. Damit haben wir zunächst folgende Struktur:

```
1  public byte[] encode(Document d){
2   byte[] docValue = null;
3   byte head = 0;
4   byte[] bytes = null;
5
6   if (d == null) {
7       return new byte[]{ EMPTYDOCTYPE, 0, 0, 0, 0 };
8   }
9   else if(d instanceof IntDoc){
10      docValue = ((IntDoc)d).getValue();
11      if(docValue == null || ! noLeadingZeros(docValue,0))
12         return null;
13      head = INTDOCTYPE;
```

```
14  }
15  else if (d instanceof KeyDoc){
16         ...
17  }
18  else if (...)
19  else if (d instanceof Doclist){
20         ... //s. weiter unten
21  }
22  if(docValue.length > 2147483642)
23    return null;
24
25  bytes = new byte[5 + docValue.length];
26  setHead(head, bytes, docValue.length);
27  ByteArray.copy(docValue,0, bytes, 5, docValue.length);
28  return bytes;
29 }
30
31 private void setHead(byte head, byte[] bytes, int length){
32  bytes[0] = head;
33  ByteArray.setInt(bytes,1,length);
34 }
```

Der `null` Wert wird, wie bereits während `java2doc` und `doc2java` als Entsprechung von `bot` verwendet. Daher kodieren wir diesen Wert mittels `new byte[]{ EMPTYDOCTYPE, 0, 0, 0, 0 }`. Für die nicht-rekursiven Typen (für die hier das `IntDoc` stellvertretend behandelt wird), muss für eine wohlgeformte Zeigerstruktur gelten, dass das `byte[ ]` , das den Wert kodiert (hier `.value`, lokale Variable `docValue`) nicht den Wert `null` hat. Ebenso muss die enthaltene Zahlendarstellung (bei `IntDoc`, `KeyDoc`, `NonceDoc` und `SecretDoc`) minimal sein (`noLeadingZeros`). Der enthaltene Wert des jeweiligen Dokumentes wird dann mittels der `byte[ ]` Variablen `docValue` zwischengespeichert. Ist der Wert zu lang, liefern wir `null` zurück. Des Weiteren verwenden wir im Erfolgsfall die Variable `head` verwendet, um das Typ-Byte zwischenzuspeichern. Am Ende der Methode wird dann ein neues `byte[ ] bytes` erzeugt, das Wert und Header (mit Typ-Byte und Länge) enthält. Letzterer wird von der Methode `setHead` gesetzt. Schließlich wird der `docValue` in das `bytes` Array kopiert (Bibliotheksfunktion `ByteArray.copy(source, startindex, destination, destindex, length)`) und der `bytes` Wert zurückgegeben. Im Fall der Doclist benötigen wir eine Rekursion:

```
1  else if (d instanceof Doclist){
2    docValue = new byte[0];
3    head = DOCLISTTYPE;
4    Document[] docs = ((Doclist)d).getDocs();
5    if(docs == null  || ! docArray(docs))
6      return null;
7    for(int i = 0; i < docs.length; i++)
8    {
9      byte[] b = encode(docs[i]);
10     if(b == null) return null;
11     docValue = ByteArray.append(docValue, b);
```

```
12    }
13  }
```

Das Array `docValue` kann hier zu Beginn noch nicht mit der richtigen Länge initialisiert werden, da noch nicht klar ist, wie viele bytes die Kodierung der enthaltenen Dokumente benötigen wird. Diese Information haben wir erst nach dem Rekursionsaufruf. Daher initialisieren wir zunächst mit dem leeren Array. Im Anschluss folgt nun die Prüfung auf Wohlgeformtheit. Für den `Doclist` Typ darf das enthaltene `Document` Array nicht `null` sein, da dies keiner abstrakten Liste entspricht. Zusätzlich muss der Typ des Arrays richtig sein. Hier kommt nun eine Besonderheit der Java Sprache zum Tragen: In einem Array mit statischem Typ `X[ ]` kann zur Laufzeit auch ein Array `Y[ ]` eines Subtyps `Y extends X` stehen. Daher müssen wir hier prüfen, ob der Typ des Eingabearrays `Document[ ]` ist. Eine solche Überprüfung ist direkt in Java nicht möglich. Wir können lediglich per `instanceof` die Subtypbeziehung prüfen. Damit müssen wir umgekehrt vorgehen und alle Subtyp-Arrays ausschließen. Dies implementieren wir in der Methode `docArray`:

```
1  private boolean docArray(Document[] r){
2   if(r instanceof IntDoc[] ||
3      r instanceof SecretDoc[] ||
4      r instanceof NonceDoc[] ||
5      r instanceof KeyDoc[] ||
6      r instanceof HashDoc[] ||
7      r instanceof EncDoc[] ||
8      r instanceof SigDoc[] ||
9      r instanceof Doclist[] )
10      return false;
11   return true;
12  }
```

Danach wird in `encoderec` innerhalb einer Schleife für jedes Element des Arrays die Kodierungsfunktion aufgerufen und dann das Ergebnis mittels `ByteArray.append` an den zwischenzeitlichen Ergebniswert angehängt. Auch `append` ist dabei nicht-destruktiv programmiert und kopiert die beiden Eingabearrays in ein neues Ausgabearray.

Damit ist die Grundstruktur programmiert. Es fehlt allerdings noch der Test auf Zyklizität der Eingabestruktur. Dieser könnte nun destruktiv programmiert werden (Streichen bereits besuchter Elemente) oder durch Aufsammeln aller besuchten Referenzen in einer Listenstruktur. Wir wählen hier einen anderen Ansatz. Bei zyklischen Strukturen würde die bisherige Implementierung trivial nicht terminieren. Wir versehen daher die `encode` Methode mit einem zusätzlichen Parameter, der die aktuelle Tiefe in der Dokumentenstruktur angibt. Bei jedem rekursiven Aufruf inkrementieren wir den Zähler um eins. Überschreitet dieser Zähler einen großen Wert, dann liegt entweder ein Zyklus vor, oder wir haben eben ein recht großes Dokument als Eingabe erhalten. Die Minimallänge eines kodierten Dokumentes ist dabei 5 bytes (für den `null` Wert bzw. $\perp$). Der schlimmste Fall (also der Fall mit maximaler Rekursionstiefe im Sinne obigen Zählers bei gleichzeitig minimaler Länge des kodierten Ergebnis-Arrays) ist dabei eine Eingabe der Form Doclist(Doclist(Doclist(...(Doclist($\perp$))))). Jetzt nutzen wir aus, dass wir Dokumente mit einer kodierten Gesamtlänge größer 2^{31} sowieso nicht mehr kodieren können. Damit müssen wir nur einen so großen Wert als Abbruchbedingung für den Rekursionstiefenzähler wählen, damit im Falle der schlimmst möglichen obigen Eingabe ein Array als

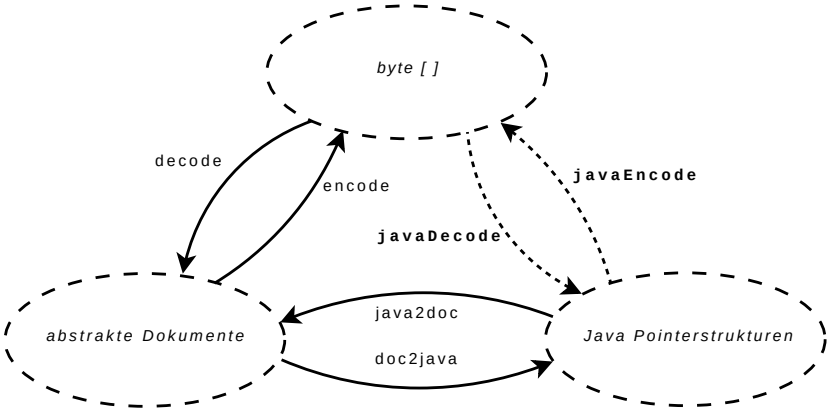


Abbildung 10.1: Zusammenhänge zwischen den Datentypen

Ergebnis geliefert werden würde, das länger als 2^{31} ist. Wir verwenden hier der Einfachheit halber `1000000000` als Abbruchbedingung. Wir geben dann `null` zurück. Es ist für die Korrektheit der Implementierung egal, ob wir wegen Erreichen eines Zyklus oder wegen zu langer Eingabe einen Fehler signalisieren müssen. Damit verändern wir die bisherige Implementierung zu:

```
1  public byte[] encode(Document d) {
2    return encoderec(d, 0);
3  }
4
5  public byte[] encoderec(Document d, int c){
6    if (c > 1000000000) return null;
7
8    //Rest der Methode wie gehabt
9    //lediglich im Doclist-Fall:
10   ...
11   else if (d instanceof Doclist){
12     ...
13     for(int i = 0; i < docs.length; i++)
14     {
15       byte[] b = encoderec(docs[i], c + 1);
16       ...
17     }
18   }
```

Abschließend noch als Anmerkung und unabhängig von der Betrachtung hier: Real wird dieser Wert im Fall nicht zyklischer Dokumente nie erreicht werden, da keine reale Eingabe für die betrachteten Anwendungsdomänen wie z.B. Mobiltelefone oder Chipkarten diese Größe erreichen wird.

10.5 Korrektheit der Implementierung

Nun muss gezeigt werden, dass diese Implementierung bezüglich der gegebenen Spezifikation aus Kap. 10.3 korrekt ist. Dazu müssen wir die Eigenschaften 1 - 4 aus Kap. 10.4.1 formal anhand der Implementierung nachweisen. Wir haben damit die in Abbildung 10.1 dargestellten Zusammenhänge zwischen Java Pointerstrukturen, abstrakten Dokumenten und Arrays von bytes. In der Darstellung ist dabei der `byte[ ]` Datentyp sowohl synonym für die (abstrakten) Bytelisten als auch die Java `byte[ ]` Arrays, die mittels eines Speichers und einer Referenz auf ein solches Array repräsentiert werden. Die Umwandlung zwischen den beiden Darstellungsarten ist dabei durch die Bibliotheksfunktionen addbytearray und getbytearray uniform möglich.

Die Korrektheitseigenschaften der Implementierung beschreiben intuitiv, dass die Ergebnisse der gepunkteten Übergänge zwischen den Datentypen in Abb. 10.1 gerade auch durch entsprechende Übergänge entlang der durchgezogenen Übergänge entstehen können. Die gepunkteten Linien sind dabei gerade die beiden Implementierungen decode und encode aus dem letzten Abschnitt. Unter Vernachlässigung der Signaturen der jeweiligen Funktionen (und des Speichers) soll also z.B. gelten, dass javaEncode(d) = encode(java2doc(d))

Wir wiederholen nun die Eigenschaften aus Kapitel 10.4.1, stellen sie jeweils formal dar und erläutern kurz die Kernideen der Beweise. Die vollständigen formalen Beweise finden sich in der Spezifikation *encoding-correct* im Projekt *javadoc*.

10.5.1 Korrektheitseigenschaft 1: Dekodierung im Erfolgsfall

Jedes `byte[ ]` , das wohlgeformt im Sinne von validEncoding ist, wird dekodiert zu genau der Pointerstruktur, die umgewandelt mittels java2doc dem Ergebnis des abstrakten decode auf dem Array entspricht. Insbesondere erfüllen alle diese Ergebnisse validDoc. Es muss also gelten:

THEOREM:

decode-ok:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
bytes = getbytearray(r, st),
validEncoding(bytes),
validrefnotnull('encoding-impl', r, mkarraytype(byte_type), st),
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0
⊢ ⟨st; `r1 = r2.decode(r);` ⟩
(java2doc(r_1, st) = decode(bytes)
∧ validDoc(r_1, st)
∧ $st_0 \subseteq$ st
∧ validref(r_1, Document, st))

Die beiden zusätzlichen Formeln in der Nachbedingung der Beweisverpflichtungen besagen dabei noch zusätzlich, dass der bisherige Speicherinhalt unverändert geblieben sein muss ($st_0 \subseteq st$) und der Rückgabewert eine typkorrekte Pointerstruktur sein muss (validref(...)).

Bei Betrachtung der Formel fällt auf, dass in Hinblick auf die Zusammenhänge in Abbildung 10.1 eigentlich eine Nachbedingung der Form $r_1 \times st = doc2java(decode(bytes), st_0)$ naheliegender gewesen wäre. Das Ergebnis der Dekodierung in Java sollte schließlich genau dem Ergebnis der Wandlung des Resultats der abstrakten Dekodierung nach Java entsprechen. Dies ist allerdings nicht beweisbar. Der Grund liegt in der Tatsache, wie doc2java eine Pointerstruktur zu einem abstrakten Eingabedokument erzeugt. Wie in Kap. 6.4 erläutert, wird dazu die algebraische Funktion newref_list verwendet, die einige neue Referenzen für die neue Struktur liefert. Deren Werte sind dabei deterministisch die nächsten Referenzen, die noch nicht im Speicher enthalten sind (die Konstruktion einer neuen Referenz erfolgt durch Erhöhung der bisherigen Referenz mit maximalem Wert). Die Kalkülregeln des Java Kalküls in KIV sehen jedoch für die Objekterzeugung lediglich irgendwelche neuen Referenzen vor. Schließlich ist auch für die echte Virtual Machine nicht spezifiziert, welche Werte neue Objektreferenzen haben sollen. Damit wird allerdings nicht mehr beweisbar, dass genau die Pointerstrukturen im echten Programm entstehen, die auch per doc2java entstanden wären, sondern lediglich noch im Sinne von pseq (s. Kap. 8.5) isomorphe Strukturen. Im Sinne einer einfacheren Verwendbarkeit der Eigenschaften der Implementierung der Transformationsschicht in einer späteren Protokollimplementierung haben wir daher hier die Korrektheitseigenschaft über java2doc formuliert und verwenden doc2java nicht mehr.

Beweisidee:

Zum Beweis der Eigenschaft *decode-ok* machen wir zunächst eine Fallunterscheidung über das Typ-Byte des vorliegenden Arrays bytes. Stellvertretend für die nicht-rekursiven Fälle betrachten wir hier das `IntDoc`. Die Beweisstrategie ist dabei immer im Folgenden die symbolische Ausführung des Java Programms und der Beweis der resultierenden prädikatenlogischen Beweisverpflichtung. Der relevante Codeausschnitt ist hier, wie auch im Code auf Seite 151 bzw. in Anhang 13.1 angegeben:

```
1  if(r != null && validEncoding(r)){
2    if(r[0] == INTDOCTYPE){
3     return new IntDoc(ByteArray.restn(5, r));
4    }
```

Die Eingabe `r` ist nach Voraussetzung der Beweisverpflichtung ungleich `null` (validrefnotnull(...) und erfüllt validEncoding. Es verbleibt also ein Lemma anzuwenden, dass bei Gültigkeit des abstrakten Prädikates validEncoding auch einen positiven Rückgabewert für die Java Methode `validEncoding` für das entsprechende `byte[ ]` folgern lässt. Ein solches Lemma ist:

THEOREM:

validencoding-def:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
bytes = getbytearray(r, st),
validrefnotnull('encoding-impl', r, mkarraytype(byte_type), st),
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0
⊢ ⟨st; `boolvar = r2.validEncoding(r);` ⟩
(boolvar ↔ validEncoding(bytes)
∧ st_0 ⊆ st)

Der Beweis dieses Hilfslemmas wird später bei der nächsten Eigenschaft beschrieben.

Im Falle r[0] = INTDOCTYPE wird jetzt die Bibliotheksmethode `ByteArray.restn(...)` verwendet, um den Abschnitt von bytes zu separieren, der den Wert des IntDocs repräsentiert. Als Beweisziel müssen wir zeigen, dass das entsprechende `IntDoc` mit diesem Wert umgewandelt mittels java2doc dem Ergebnis von decode(bytes) entspricht. Da decode ebenso restn (diesmal die abstrakte Funktion auf Byte-Listen) verwendet, benötigen wir also einen entsprechenden Zusammenhang zwischen `ByteArray.restn(...)` und restn(...). Dieser ist:

THEOREM:

ByteArray-restn:
init(ByteArray, st), .mode ∈ st, st[.mode] = noval,
validrefnotnull('encoding-correct', r_0, mkarraytype(byte_type), st),
0 ≤ i,
i ≤ st[r_0.length].intval,
st = st_0
⊢ ⟨st; `r1 = ByteArray.restn(i, r0);` ⟩
(st_0 ⊆ st
∧ getbytearray(r_1, st) = restn(i2n(i), getbytearray(r_0, st_0))
∧...) // *und einige Wohlgeformtheitsbedingungen*

Damit wissen wir also, dass die implementierte und die spezifizierte restn Funktion sich gleich verhalten. Die Vorbedingungen bezüglich des Parameters i für `ByteArray.restn(...)` lassen sich aus dem validEncoding Prädikat folgern (nur `byte[ ]` Arrays mit einer Länge

über 5 bytes erfüllen überhaupt validEncoding). Zusätzlich gilt:

THEOREM:

restn-getba:
restn(n, getbytearray(r,st)) = getbytearray(r, n2i(n), st[r.length] - n, st)

wobei getbytearray(r,i,j,st) das Array an Referenz r beginnend bei Index i und Länge j zurückliefert.

Damit verbleibt nach Ausführung des `IntDoc` Konstruktors, der ein neues Objekt (mittels addobj(...) anlegt, als Beweisverpflichtung schematisch eine Sequenz mit u.a. folgenden Formeln:

...
st_1 = addobj(r_1, IntDoc, .value × refval(jvmref), st_0),
⟨st_1; r_2 = ByteArray.restn(5,r); ⟩ (st_1 = st ∧ r_2 = r_3),
st_1 ⊆ st
getbytearray(r_3, st) =
getbytearray(r, n2i(5), st_0[r.length] - 5, st_1)
⊢ java2doc(r_1, st[r_1.value, refval(r_3)]) = decode(getbytearray(r, st_0)),
...

Durch Anwenden der Definitionen für java2doc und decode jeweils für den IntDoc-Fall und dem Zusammenhang st[r_1.value, refval(r_3)][r_1.value].refval = r_3 und obiger Eigenschaft *restn-getba* erhalten wir daraus eine Sequenz der Form:

...
addobj(addobj(r_1, IntDoc, .value × refval(jvmref), st_0)) ⊆ st
getbytearray(r_3, st) =
  getbytearray(r, n2i(5), st_0[r.length] - 5,
    addobj(r_1, IntDoc, .value × refval(jvmref), st_0))
⊢ IntDoc(bytes2int(getbytearray(r_3, st))) =
  IntDoc(bytes2int(getbytearray(r, n2i(5), st_0[r.length] - 5, st_0))),
...

Dies können wir nun schließen, indem wir ausnutzen, dass das Hinzufügen eines anderen Objektes keine `byte[ ]` Objekte ändern kann:

THEOREM:

$$r \neq r_1 \rightarrow \text{getbytearray}(r_1, i, j, \text{addobj}(r, \text{class}, \text{fis}, st)) = \text{getbytearray}(r_1, i, j, st)$$

Die anderen nicht rekursiven Fälle lassen sich genauso schließen.

Betrachten wir nun den komplizierteren rekursiven Fall, wenn bytes[0] = DOCLISTTYPE. Hier ist nun der Inhalt des kodierten Dokumentes wiederum eine Liste von kodierten Dokumenten. Wir müssen also später eine Induktion verwenden. Zunächst geht der Beweis aber analog zum nicht-rekursiven Fall über die entsprechenden Eigenschaften für validEncoding und seine Implementierung. Anschließend kommen wir allerdings an die rekursive Routine `decodelist`:

```
1   else if(b[0] == DOCLISTTYPE){
2    return decodeList( ByteArray.restn(5, b) );
3   }
```

Hier verwenden wir nun eine Korrektheitseigenschaft für `decodeList`, die die rekursive Java Methode mit der rekursiven Funktion decodelist für Listenkodierungen in Beziehung setzt:

THEOREM:

decodeList-def:
$\text{init}(\text{'encoding-correct'}, st), .\text{mode} \in st, st[.\text{mode}] = \text{noval},$
$\text{bytes} = \text{getbytearray}(r, st),$
$\text{validEncoding}(\text{bytes}),$
$\text{validrefnotnull}(\text{'encoding-impl'}, r, \text{mkarraytype}(\text{byte\_type}), st),$
$\text{validrefnotnull}(\text{'encoding-impl'}, r_2, \text{Coding}, st),$
$st = st_0$
$\vdash \langle st;$ `r1 = r2.decodeList(r);` $\rangle$
$\quad (\quad \text{java2doc}(r_1, st) = \text{doclist}(\text{decodelist}(\text{bytes}))$
$\quad \wedge\ st_0 \subseteq st$
$\quad \wedge\ \text{validDoc}(r_1, st)$
$\quad \wedge\ \text{validref}(\text{'encoding-correct'}, r_1, \text{Document}, st))$

Für den Beweis dieser Eigenschaft benötigen wir nun eine Induktion. Wir induzieren über die Länge des zu dekodierenden Arrays i2n(st[r.length].intval). Die Implementierung für `decodeList` sieht nun eine Fallunterscheidung vor, ob das (Rest-)Array im aktuellen Parameter schon leer ist oder noch mindestens eine Kodierung enthalten ist. Der entsprechende Code ist:

```
1  if(bytes.length == 0)
2      return new Doclist();
3
4  int length = ByteArray.getInt(bytes, 1);
5
6  Document doc =
7    decode( ByteArray.firstn(length + 5, bytes));
8  Doclist doclist =
9    decodeList(ByteArray.restn(length + 5, bytes));
10 return doclist.cons(doc);
```

Der Fall `bytes.length == 0` kann einfach geschlossen werden, schließlich liefert dann sowohl java2doc(r_1, st) nach Definition von java2docs für leere Arrays Doclist([]) und ebenso ist decodelist([]) = []. Die anderen Nachbedingungen gelten ebenso einfach.

Im Fall `bytes.length != 0` wird nun sowohl `decode` für das erste Kodierungselement (`ByteArray.firstn( length + 5, bytes)`) also auch `decodeList` für den Rest aufgerufen. Mit der Induktionshypothese können wir allerdings nur über `decodeList` Aussagen treffen, `decode` kann mit ihr nicht behandelt werden. Hier müssen wir uns also eine andere Strategie überlegen. Wir verwenden zur Lösung eine Fallunterscheidung über den Typ des ersten Kodierungselements. Handelt es sich hierbei um keine Doclist, dann können wir ein eigenes Lemma für die Dekodierung von nicht-rekursiven Kodierungen verwenden und damit `decode` behandeln. Falls es sich allerdings um eine Doclist handelt, führen wir `decode` einfach symbolisch aus (also ein zweites Mal) und erhalten aus dem entsprechenden Methodenrumpf (wiederum) einen Aufruf für die enthaltene Doclist mittels `decodeList`. Diesen Aufruf können wir allerdings nun mit unserer (originalen) Induktionshypothese behandeln. Schließlich verbleibt noch der eigentliche Aufruf von `decodeList` im eigentlichen Methodenrumpf, welche wir in beiden Fällen mittels der Induktionshypothese schließen können. Für den Fall Doclist(Doclist(...)) verwenden wir also die Induktionshypothese *zweimal*. Abb. 10.2 stellt die Zusammenhänge im Beweis nochmals grafisch dar:

Am Ende müssen wir noch das Ergebnis des Aufrufs von `decode` und des rekursiven Aufrufs von `decodeList` mittels `cons` zusammensetzen. Hier ist dann die wesentliche Eigenschaft:

THEOREM:

cons-def:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
validDoc(r, st), validDoc(r_1, st), validref('encoding-impl', r, Doclist, st),
validrefnotnull('encoding-impl', r_1, Document, st),
st = st_0
⊢ ⟨`st; r2 = r.cons(r1);` ⟩
(java2doc(r_2, st) = doclist(java2doc(r_1, st_0) + java2doc(r, st_0).list)

$\wedge\ st_0 \subseteq st$
$\wedge \ldots$

Auch hier zeigt sich wieder die allgemeine Beweistaktik, Eigenschaften der Java Programme über Eigenschaften der abstrakten Entsprechungen auszudrücken. Im Wesentlichen besagt obige Eigenschaft, dass sich die Java Methode `cons` genauso verhält wie die Listenkonkatenation auf den mittels java2doc abstrahierten Eingabeparametern. Wir eliminieren mit solchen Lemmata also Java Programme und führen prädikatenlogische Eigenschaften über den Speicher und dessen Abstraktionen ein.

Anschließend folgt ähnlich zum nicht-rekursiven Fall mit den Definitionen von java2doc und decode die Beweisverpflichtung.

10.5.2 Korrektheitseigenschaft 2: Dekodierung im Fehlerfall

Jedes `byte[ ]` , das nicht wohlgeformt im Sinne von validEncoding ist, wird von der Dekodierung erkannt und es wird `null` zurückgegeben. Formal bedeutet dies:

THEOREM:

decode-fail:
init('encoding-correct', st), .mode $\in$ st, st[.mode] = noval,
bytes = getbytearray(r, st),
$\neg$ validEncoding(bytes),
validrefnotnull('encoding-impl', r, mkarraytype(byte_type), st),
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0
$\vdash \langle$st; `r1 = r2.decode(r);` $\rangle$
($st_0 \subseteq$ st
$\wedge\ r_1$ = jvmref)

Auch hier darf alter Speicherinhalt nicht verändert werden ($st_0 \subseteq st$).

Beweisidee:

Wesentlich für diese Eigenschaft ist nun nicht mehr die eigentliche Implementierung der `decode` Methode, sondern der initiale Check der Eingabe mittels `validEncoding`. Diese Methode muss für jede Eingabe, die nicht validEncoding erfüllt, `false` zurückgeben. Dies

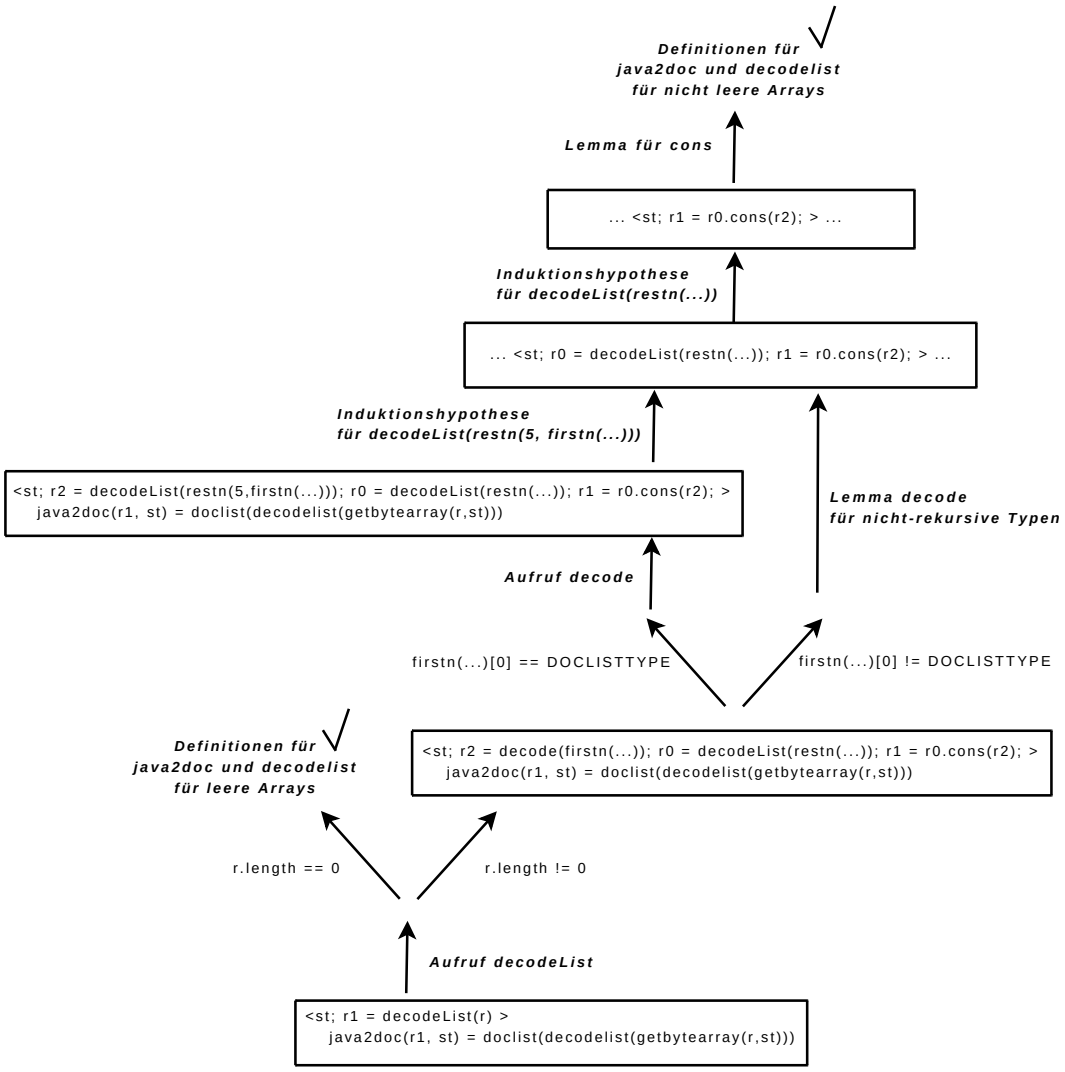


Abbildung 10.2: Beweisstrategie für decodeList

folgt direkt aus dem oben bereits angegebenen Theorem validEncoding-def. Somit reduziert sich der Beweis von decode-fail im wesentlichen auf die Korrektheit von validEncoding-def. Hierzu verwenden wir nun wiederum die gleiche Beweisstrategie wie bereits für die Dekodierung im Erfolgsfall. Die nicht-rekursiven Fällen ergeben sich durch symbolische Ausführung und Expansion der entsprechenden Definitionen für validEncoding. Die jeweiligen Checks im Java Code entsprechen denen der jeweiligen abstrakten Definition für validEncoding. Der eigentlich interessante Fall ist wiederum die rekursive Listenkodierung. Auch hier wird wieder (analog zu oben) eine Induktion über die Länge der kodierten Liste gestartet. Im Rekursionsfall wird ebenso analog zu oben `validEncoding` als auch `validListEncoding` aufgerufen. Wir müssen also wieder eine Fallunterscheidung über den Typ des ersten Kodierungselementes machen. Im Fall einer Doclist wird wieder ausgeführt, bis wieder `validListEncoding` benutzt wird und wir die Induktionshypothese verwenden können. Insgesamt benötigen wir also in diesem Fall auch wieder zwei Anwendungen der Induktionshypothese.

10.5.3 Korrektheitseigenschaft 3: Kodierung im Erfolgsfall

Jedes Java `Document` Objekt, das eine wohlgeformte Pointerstruktur im Sinne von validDoc ist, wird kodiert zu genau dem `byte[ ]` , welches Ergebnis des abstrakten encode auf dem Ergebnis von java2doc ist.

THEOREM:

encode-ok:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
validDoc(r_1, st),
codable(java2doc(r_1,st)),
validrefnotnull('encoding-impl', r_1, Document, st),
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0
⊢ ⟨st; `r = r2.encode(r1);` ⟩
(addarray(r, byte_type, encode(java2doc(r_1, st_0))) ⊆ st
∧ newref(r, st_0))

In die Vorbedingung müssen wir hier wegen der beschränkten Maximallänge der Kodierung ebenso codable(...) für die Eingabe fordern.

Die Nachbedingung ist hierbei so formuliert, dass sowohl der richtige Rückgabewert als auch die Unveränderthеit des restlichen Speichers in einer Bedingung vereint ist. Der richtige Rückgabewert ist die Byteliste encode(java2doc(r_1, st_0)). Dieser soll als Java `byte[ ]` während der Ausführung von `encode` in den Speicher an einer neuen Referenz (newref(r, st_0)) eingefügt werden (addarray(...)).

Beweisidee:

Die Beweisidee bei der Kodierung von Dokumenten ist nun wiederum abhängig vom Typ des jeweils zu kodierenden Dokumentes. Wir machen also eine Fallunterscheidung über den Typ der zu kodierenden Pointerstruktur im Parameter r_1. Für alle Typen außer der `Doclist` ist `encode` (und insbesondere die Hilfsmethode `encoderec`) nicht rekursiv. Somit können wir den Methodenrumpf durch symbolische Ausführung behandeln und müssen schließlich für das resultierende Java `byte[ ]` zeigen, dass genau die gleichen Werte wie in der abstrakten Spezifikation enthalten sind. Der relevante Code wurde bereits auf Seite 154 angegeben. Exemplarisch betrachten wir zunächst wieder das `IntDoc`. Beim Kodieren wird im Gegensatz zum Dekodieren während des Ablaufens der Struktur die Wohlgeformtheit (entsprechend der Gültigkeit von validDoc) geprüft. Somit wird zunächst ein Null-Check auf das `.value` Feld des `IntDoc`s und danach für dieses Feld die Methode `noLeadingZeros` verwendet. Wir können mittels folgender Eigenschaft die Methode `noLeadingZeros` behandeln:

THEOREM:

noLeadingZeros-def:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
codable(java2doc(r_1,st)),
validrefnotnull('encoding-impl', r_1, Coding, st),
validrefnotnull('encoding-impl', r_2, mkarraytype(byte_type), st),
st = st_0
⊢ ⟨st; `boolvar = r1.noLeadingZeros(r2);` ⟩
(boolvar
↔ int2bytes(bytes2int(restn(i2n(i), getbytearray(r_2, st_0))))
= restn(i2n(i), getbytearray(r_2, st_0))
∧ st = st_0)

Wir verwenden hier eine weiter parametrisierte Version von `noLeadingZeros`, die zusätzlich einen Index besitzt, ab dem die gewünschte Eigenschaft im vorliegenden `byte[ ]` geprüft werden soll. Dies liegt daran, dass `noLeadingZeros` sowohl innerhalb `validEncoding` für die Werte ab Index 5 als auch hier innerhalb `encode` für die Werte ab Index 0 benutzt wird (s. auch Code in Anhang 13.1).

Im Anschluss an die Checks wird der Wert des `IntDoc`s lokal in einer Variablen zwischengespeichert und am Ende der Methode `encoderec` schließlich ein entsprechend großes Array für Wert sowie Typ-Byte und Längenfeld reserviert. Dann wird mittels der `ByteArray.copy` Methode und `setHead` alles in dieses Array kopiert. Wir erhalten dadurch nun eine Beweisverpflichtung nach symbolischer Ausführung und entsprechender Lemmas für `copy` und `setHead` der Art:

...
⊢ addarray(r, byte_type, mkjavavalues(encode(java2doc(r_1, st))))
⊆ putByteArray(
getbytearray(st[r_1.value].refval, st), // **Wert des IntDocs**
r, 5, // **kopieren in Array r ab Index 5**
putByteArray(i2b(1) // **und davor Typ Byte**
+ fillsgn(4, int2bytes(st[r_1.length].intval)), //**und Länge**
0, addarray(r, byte_type, 5 + st[r_1.value].length, st))
// **in ein leeres Array r ab Index 0**

Wenn wir also zeigen können, dass

```
mkjavavalues(encode(java2doc(r1, st))) =
          i2b(1)
        + fillsgn(4, int2bytes(st[r1.length].intval))
        + getbytearray(st[r1.value].refval, st)
```

dann lässt sich dieses Ziel schließen, da dann die Teilmengenbeziehung aufgrund von Identität der linken und rechten Seite trivial erfüllt ist. Obige Gleichung folgt aus der Definition von encode und java2doc.

Im rekursiven Fall einer `Doclist` ist es wiederum komplizierter. Die entsprechende Beweisverpflichtung ist:

THEOREM:

```
encoderec-doclist:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
validDoc(r1, st),
codable(java2doc(r1,st)),
validrefnotnull('encoding-impl', r1, Doclist, st),
validrefnotnull('encoding-impl', r2, Coding, st),
st = st0
⊢ ⟨st; r = r2.encoderec(r1, 0); ⟩
      (   addarray(r, byte_type, encode(java2doc(r1, st0))) ⊆ st
        ∧ newref(r, st0))
```

Zunächst benötigen wir hierzu einen passenden Induktionsterm. Die Idee ist, dass die Rekursion zur Kodierung entweder abbricht, wenn ein maximaler Rekursionszähler (hier 1.000.000.000, s. Implementierung vorne) erreicht ist oder die Struktur nicht zyklisch und klein genug für eine ordentliche Kodierung ist. Wenn der jeweils aktuelle Methodenparameter für `encoderec` dabei den Wert `j` hat, können wir somit über 1.000.000.000 - j induzieren. Hierfür müssen wir zunächst die Beweisverpflichtung verallgemeinern:

THEOREM:

```
encoderec-codable-validDoc:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
validDoc(r1, st),
codable(java2doc(r1, st)),
```

validrefnotnull('encoding-impl', r_1, Doclist, st),
validrefnotnull('encoding-impl', r_2, Coding, st),
$0 \leq$ j,
j $\leq$ 1000000000,
n2i(# encode(java2doc(r_1, st))) $\leq$ (1000000000 - j) * 5
st = st_0
$\vdash \langle$st; `r = r2.encoderec(r1, j);` $\rangle$
(addarray(r, byte_type, encode(java2doc(r_1, st_0))) $\subseteq$ st
$\wedge$ newref(r, st_0))

Hiermit starten wir die besagte Induktion. Der Beweis gewinnt wesentlich an Komplexität durch die Tatsache, dass die Kodierung einer Dokumentenliste durch eine `for`-Schleife geschieht, die nach und nach die Listenelemente abarbeitet. Hier wird eine Invariante benötigt, um den Beweis schließen zu können. Innerhalb der Schleife wird allerdings wieder die Methode `encoderec` aufgerufen. Der relevante Code ist:

```
1  public byte[] encoderec(Document d, int j) {
2   ...
3   else if (d instanceof Doclist){
4     docValue = new byte[0];
5     head = DOCLISTTYPE;
6     Document[] docs = ((Doclist)d).getDocs();
7     if(docs == null  || ! docArray(docs))
8       return null;
9     for(int i = 0; i < docs.length; i++)
10    {
11      byte[] b = encoderec(docs[i], j + 1);
12      if(b == null) return null;
13      docValue = ByteArray.append(docValue, b);
14    }
15  }
16   ...
17 }
```

Wir müssen also eine Invariante verwenden, die die Behandlung dieses rekursiven Aufrufs erlaubt. Dies geht, indem die *Induktionshypothese* des bisherigen Beweises in die Invariante mit aufgenommen wird. Zusätzlich müssen wir angeben, dass das in der `Doclist` enthaltene Array bis zum aktuellen Laufindex richtig kodiert ist. Es muss also (mit den Variablenbezeichnern des obigen Sourcecode-Stücks) in der Invariante gelten:

getbytearray(docValue, st) =
encode(java2docs(getarray(st[d.docs].refval, 0, j, st), st))

Die Invariante beinhaltet daneben noch ca. 30 andere Eigenschaften über Speicher und lokale Variablen, die Wohlgeformtheitseigenschaften und die ursprüngliche Induktionshypothese beinhalten. Terminierung der Schleife wird einfach sichergestellt, indem man $st[d.docs.length]-j$ als Terminierungsfunktion verwendet.

Es verbleibt damit nun zu zeigen, dass bei Betreten der Schleife und Aufruf von `encoderec` darin dessen Rückgabewert die korrekte Kodierung des nächsten Listenelementes beinhaltet. Schließlich müssen wir nach dem jeweiligen Schleifendurchlauf die Invariante wieder etablieren. Eigentlich folgt dies aus der Induktionshypothese, allerdings muss noch codable und validDoc von der äußeren `Doclist` auf deren Inhalte übertragen werden (die Induktionshypothese fordert dies). Zusätzlich benötigen wir hier die Abschätzung n2i(# encode(java2doc(r_1, st))) ≤ (1000000000 - j) * 5 von encoderec-codable-validDoc, um zeigen zu können, dass der rekursive Aufruf noch nicht zum Abbruch wegen zu tiefer Rekursion führt.

10.5.4 Korrektheitseigenschaft 4: Kodierung im Fehlerfall

Jedes Java `Document` Objekt, das nicht eine wohlgeformte Pointerstruktur im Sinne von validDoc ist, wird erkannt und nicht kodiert. Dies bedeutet formal:

THEOREM:

encode-fail:
init('encoding-correct', st), .mode ∈ st, st[.mode] = noval,
¬ (validDoc(r_1, st) ∧ codable(java2doc(r_1, st))),
validrefnotnull('encoding-impl', r_1, Document, st),
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0
⊢ ⟨st; `r = r2.encode(r1);` ⟩
($st_0 \subseteq$ st
∧ r = jvmref)

Ein Fehlerfall soll dabei signalisiert werden, wenn entweder die Eingabe nicht wohlgeformt ist (validDoc(...)) oder aber das Eingabedokument zu lang ist (codable(...)).

Beweisidee:

Der Fehlerfall bei der Kodierung ist schwierigste aller 4 Fälle. Die Beweisstrategie ist dabei immer noch die gleiche wie im Erfolgsfall der Kodierung. Das bedeutet dass auch hier die nichtrekursiven Fälle durch symbolische Ausführung und entsprechende Hilfslemmata geschlossen werden können und auch hier im rekursiven Fall das oben genannte Problem der Rekursionsaufrufe innerhalb einer `for` Schleife auftritt. Auch hier muss daher eine geeignete Invariante gefunden werden und insbesondere die Induktionshypothese für die Hauptfunktion in die Invariante für die Schleife aufgenommen werden. Problematisch wird der Beweis hauptsächlich

dadurch, dass es mehr als einen Fehlerfall geben kann. Die Vorbedingung des obigen Lemmas erlaubt die Fehlerfälle $\neg$ validDoc(r_1, st) oder $\neg$ codable(java2doc(r_1, st)). Wir separieren diese zunächst, in dem wir ein Lemma für $\neg$ validDoc(r_1, st) formulieren und ein anderes für validDoc(r_1, st) $\wedge \neg$ codable(java2doc(r_1, st)).

Im Falle $\neg$ validDoc(r_1, st) kann nun innerhalb der Rekursion der Fehler (sozusagen der verantwortliche Listeneintrag für die Ungültigkeit von validDoc) im aktuellen Schleifendurchlauf auftreten oder erst später. Ebenso kann (durch Zyklizität) der Rekursionszähler zu groß werden. Damit müssen wir innerhalb des Schleifenrumpfs der `for` Schleife diese Fälle unterscheiden. Zunächst können wir uniform für zu große Rekursionszähler folgendes Lemma verwenden und damit unmittelbar die Behauptung zeigen:

THEOREM:

encoderec-big:
st[.mode] = noval,
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0,
1000000000 < j
$\vdash \langle$st; `r = r2.encoderec(r1, j);` $\rangle$
($st_0 \subseteq$ st
$\wedge$ r = jvmref)

Im Falle eines kleineren Rekursionszählers kann es nun sein, dass der nächste Listeneintrag $\neg$ validDoc(...) ist oder eben schon. Dies ist unabhängig von validDoc(...) für die Gesamtliste, man kann es zum Zeitpunkt der aktuellen Schleifendurchlaufs nicht wissen. Der in diesem Sinne "schuldige" Eintrag könnte der aktuelle Eintrag sein oder auch hinter dem aktuellen Index auftreten. Um eine sichere Terminierung der Schleife durch ein `return null;` sicherzustellen, müssen wir lediglich ausschließen, dass die Schleife alle Listeneinträge richtig kodiert. Dies wird nicht passieren, denn irgendwann wird der Index mit dem Teildokument erreicht sein, das nicht validDoc ist. Daher verwenden wir ein Lemma für validDoc, das die Existenz eines solchen Index innerhalb des Arrays folgern lässt. Wir nehmen dann in die Schleifeninvariante auf, dass der aktuelle Laufindex der Schleife höchstens kleiner oder gleich als dieser Index ist (schließlich wird die Kodierung bei Erreichen des Index abbrechen). Damit können wir beweisen, dass die Schleife nicht normal terminieren wird, sondern sicher durch ein `return` terminieren wird (womit dann die Nachbedingung der Beweisverpflichtung gilt)

Allerdings können wir nun für den Rekursionsaufruf von `encoderec` nicht mehr die Induktionshypothese verwenden, da diese nur für $\neg$ validDoc anwendbar ist. Es ist allerdings nicht mehr klar, ob der aktuelle Laufindex der "schuldige" Index für $\neg$ validDoc ist oder dieser erste später auftreten wird. Wir müssen also entsprechend verallgemeinern und `encoderec` unabhängig von validDoc charakterisieren. Wir verwenden hier lediglich die Eigenschaft, dass `encoderec` *bisherige* Speicherinhalte unverändert lässt und ein beliebiges Array (möglicher-

weise `null`) zurückliefert:

THEOREM:

encoderec-subset:
st[.mode] = noval,
validrefnotnull('encoding-impl', r_2, Coding, st),
st = st_0,
1000000000 < j
$\vdash \langle$st; `r = r2.encoderec(r1, j);` $\rangle$
($st_0 \subseteq$ st
$\wedge$ validref('encoding-correct', r, mkarraytype(byte_type), st))
$\wedge$...)

Damit lässt sich dann die Invariante nach der Schleife wieder zeigen, da sich alle Eigenschaften der Invariante über die Teilmengenbeziehung vererben.

Für den Fall validDoc(r_1, st) $\wedge \neg$ codable(java2doc(r_1, st)) müssen wir wiederum eine andere Strategie für den Beweis wählen. Hier kann zwar kein Zyklus in der Eingabe auftreten, allerdings wird irgendwann die Maximallänge für kodierte Dokumente überschritten. Problematisch hierbei ist wiederrum, dass der Grund für die Überschreitung der Maximallänge nun lokal im nächsten Dokument innerhalb der `Doclist` zu suchen sein kann (dieses also für sich allein bereits zu lang ist) oder aber erst durch Zusammensetzen aller kodierten Einzeldokumente entsteht. Wir müssen daher in der Invariante für die Schleife wieder die Induktionshypothese aufnehmen (die für ersteren Fall eine Lösung bereitstellt) und zusätzlich angeben, dass das bisher berechnete `docValue` Array bei Wiedereintritt in die Schleife die richtige Kodierung des bisherigen Listen-Präfixes beinhaltet. Mit letzterer Bedingung können wir dann zeigen, dass nach Verlassen der Schleife die Maximallänge sicher überschritten wird (da ja die Gesamtliste $\neg$ codable war). Wenn wir allerdings in der Schleife bleiben, müssen wir den rekursiven Aufruf wiederum richtig behandeln können, und zwar auch dann, wenn dieser nicht zum Abbruch der Schleife führt (es könnten schließlich alle einzelnen Einträge in der Liste codable sein und erst ihre Konkatenation könnte die Maximallänge überschreiten). Also müssen wir auch hier wieder eine verallgemeinerte Eigenschaft für `encoderec` ähnlich zu encoderec-subset oben anwenden (und beweisen), die angibt, dass der Rückgabewert der `encoderec` Methode im validDoc-Fall entweder `jvmref` oder aber eine ordentliche Kodierung des aktuellen Parameters ist. Im ersten Fall gilt die Nachbedingung der Schleife (wir terminieren abrupt mit `return null`), im zweiten Fall können wir die Invariante wieder etablieren.

10.6 Statistiken

Die Verifikation des Kodierungsschemas benötigte:

- 51 Axiome mit 358 Zeilen
- 223 Theoreme mit 11301 Beweisschritten
- 3375 Benutzerinteraktionen (Automatisierungsgrad: 70 %)

Die Verifikation benötigte dabei beginnend mit Spezifikation der Kodierung auf abstrakter Ebene und endend bei den eben vorgestellten Eigenschaften ca. 2 Monate. Die Hauptschwierigkeit war die korrekte Findung der ebenfalls vorgestellten Invarianten für die Implementierung.

10.7 Eigenschaften für den Verfeinerungsbeweis

Ausschließlich die Eigenschaften für die `encode` und `decode` Methoden sind für den eigentlichen Verfeinerungsbeweis noch nicht nützlich. Vielmehr müssen wir nun zeigen, dass damit Implementierungen für `send` und `receive` umsetzbar sind, deren Benutzung zu einer korrekten Verfeinerung der Ebene führt, die keine Übertragungsformat vorsieht und `send` und `receive` als Identitätsfunktionen vorsieht. Insbesondere müssen die Implementierungen die Eigenschaften TOSTORE-receive und FROMSTORE-send, die bereits in Kap. 6.9 eingeführt wurden, erfüllen. Kap. 8.7 hat dazu bereits Beweisideen mit Implementierungen unter Benutzung eines beliebigen Übertragungsformats geliefert. Kap. 8.7 rechtfertigt auch, dass wir für die Gültigkeit von TOSTORE-receive und FROMSTORE-send lediglich die lokalen Eigenschaften receive-ok und receive-fail sowie send-ok und send-fail aus Kap. 8.6 beweisen müssen.

Wir verwenden nun folgende Implementierungen:

```
1  public class Coding implements CommInterface {
2  ...
3  private BAComm bacomm;
4
5  public void send(Document d, int port){
6   bacomm.send(encode(d), port);
7  }
8
9  public Document receive(int port){
10  byte[] bytes = bacomm.receive(port);
11  if(bytes != null)
12      return decode(bytes);
13  return null;
14 }
15 }
16
17 public interface BAComm {
18     byte[] receive(int port);
19     void send(byte[] b, int port);
20 }
```

Wir setzen also das Versenden von Dokumentenstrukturen mittels `send(Document)` um, indem wir die Eingabe kodieren und diese dann mit einer Implementierung für das Versenden von `byte[ ]` Arrays verschicken (`BAComm.send(byte[])`). Analog wird Empfangen umgesetzt, indem man ein `byte[ ]` Array aus dem Netzwerk empfängt (`BAComm.receive()`) und dann (falls eine Eingabe vorliegt - `bytes != null`) und dieses dekodiert sowie das Ergebnis zurückliefert.

Beweisidee:

Mittels der Eigenschaften encode-ok und encode-fail von oben lässt sich nun leicht zeigen, dass eine solche Implementierung von send die Eigenschaften send-ok und send-fail erfüllt. Lediglich das eigentliche Versenden eines `byte[ ]` Arrays mit Implementierungen für das Interface `BAComm` lässt sich nun nicht mehr implementieren oder verifizieren, hier muss eine entsprechende Axiomatisierung vorgenommen werden (diese Methoden sind nicht mehr in Java implementiert und kommunizieren direkt mit der jeweiligen Netzwerkschicht). Wir müssen also axiomatisieren, dass nun `BAComm.send(byte[] b, int port)` entsprechend das Feld st[.outs] auf das Array b setzt und den st[.outport] auf den gegebenen port setzt. Spiegelbildlich muss `BAComm.receive(int port)` den Wert von $\mathsf{st}[.\mathsf{ins}_{port}]$ zurückgeben. Damit lassen sich dann auch einfach mittels der Eigenschaften decode-ok und decode-fail der Kodierung die Eigenschaften receive-ok und send-ok beweisen.

10.8 Behandlung der Kryptographie

Schließlich haben wir auch bezüglich der Kryptographie in Kap. 9.2 bereits Annahmen an das Verhalten von Methoden wie `encrypt` oder `decrypt` gemacht (und analog für Hashing und Signaturen). Dies waren die Eigenschaften decrypt-ok, decrypt-fail, encrypt-ok und encrypt-fail. Auch die Gültigkeit dieser Eigenschaften können wir nun, da wir eine Implementierung auf `byte[ ]` Ebene haben, formal beweisen. Schließlich arbeitet die tatsächliche Kryptographie in Java in einer Implementierung der Java Cryptographic Architecture auf der `byte[ ]` Ebene. Zunächst betrachten wir die Implementierung:

```
1  public class Coding implements CryptoInterface {
2    ...
3    private BACrypto bacrypto;
4
5    public byte[] encrypt(Key k, Document d){
6        byte[] b = encode(d);
7        if(b != null)
8            return bacrypto.encrypt(b, k.getBA());
9        return null;
10   }
11
12   public  Document decrypt(Key k, Document d){
13       Document d0 = null;
14       if(d instanceof EncDoc)
15           d0 = decode(
16                 bacrypto.decrypt(k.getBA(),
```

```
17                    ((EncDoc) d).getEncrypted()
18                      ));
19        if(d0 instanceof IntDoc)
20            return null;
21        return d0;
22    }
23  }
24
25  public interface BACrypto {
26      byte[] encrypt(byte[] b, byte[] b1);
27      byte[] decrypt(byte[] b, byte[] b1);
28  }
```

Verschlüsselung bedeutet also, zunächst eine Kodierung des zu verschlüsselnden Dokumentes herzustellen und anschließend eine Kryptographie-Bibliothek auf dem entstandenen `byte[ ]` zu benutzen. Dies wird nur durchgeführt, wenn die Eingabe wohlgeformt war (erkennbar am einem Rückgabewert von `encode` ungleich `null`).

Entschlüsselung wird umgesetzt, indem zunächst geprüft wird, ob die Eingabe ein `EncDoc` ist, da nur diese ein sinnvolles Ergebnis bei Entschlüsselung liefern können. Dann wird auf dem `byte[ ]` Inhalt des `EncDoc` und dem `byte[ ]` Wert des verwendeten Schlüssels `k` eine Entschlüsselungsoperation aus einer Kryptobibliothek aufgerufen. Wir erhalten daraus ein `byte[ ]` Array, das noch dekodiert werden muss. Nun benötigen wir lediglich noch die Annahme, dass die Kryptographie auf `byte[ ]` Arrays ebenso die Eigenschaften der perfekten Kryptographie erhält, genauso wie dies bereits in Kap. 9.2 für die Kryptographie auf Dokumenten erläutert wurde. Wir benutzen dazu folgende Eigenschaften für `encrypt` auf `byte[ ]` Arrays:

SPEZIFIKATION:

encrypt-bytes:

$$
\begin{aligned}
&\text{validref("docs2store", } r_1\text{, mkarraytype(byte\_type), st)},\\
&\wedge\ \text{validref("docs2store", } r_2\text{, mkarraytype(byte\_type), st)},\\
&\wedge\ \text{validref("docs2store", } r_3\text{, BACrypto, st)},\\
&\wedge\ \text{bytes}_1 = \text{getbytearray}(r_1, \text{st})\\
&\wedge\ \text{bytes}_2 = \text{getbytearray}(r_2, \text{st})\\
&\wedge\ \text{st[.mode]} = \text{noval}\\
&\wedge\ \text{st} = \text{st}_0\\
\rightarrow\ &\langle \text{st};\ \texttt{r0 = r3.encrypt(r1, r2);}\ \rangle\\
&\quad (\text{st}_0 \subseteq \text{st} \wedge \text{getbytearray}(r_0, \text{st}) = \text{baencrypt}(\text{bytes}_1, \text{bytes}_2))
\end{aligned}
$$

wobei für baencrypt gilt:

SPEZIFIKATION:

baencrypt:
baencrypt(bytes$_1$, encode(doc)) = doc2cryptbytes(mkkey(bytes$_1$), doc)

Dies bedeutet also, dass wir annehmen, dass sich die Kryptographie auf `byte[ ]` Arrays genauso verhält wie unsere Repräsentationsfunktion für die Abbildung der kryptographischen Dokumente auf `byte[ ]` Arrays, die wir bereits in Kap. 9.2 eingeführt haben.

Analog gilt das selbe für decrypt und cryptbytes2doc:

SPEZIFIKATION:

decrypt-bytes:
validref("docs2store", r$_1$, mkarraytype(byte_type), st),
$\wedge$ validref("docs2store", r$_2$, mkarraytype(byte_type), st),
$\wedge$ validref("docs2store", r$_3$, BACrypto, st),
$\wedge$ bytes$_1$ = getbytearray(r$_1$, st)
$\wedge$ bytes$_2$ = getbytearray(r$_2$, st)
$\wedge$ st[.mode] = noval
$\wedge$ st = st$_0$
$\rightarrow$ $\langle$st; `r0 = r3.decrypt(r1, r2);` $\rangle$
(st$_0$ $\subseteq$ st $\wedge$ getbytearray(r$_0$, st) = badecrypt(bytes$_1$, bytes$_2$))

wobei

SPEZIFIKATION:

badecrypt:
badecrypt(bytes$_1$, bytes$_2$) =
encode(decrypt(mkkey(bytes2int(bytes$_1$), cryptbytes2doc(bytes$_2$))))

Damit haben wir die Eigenschaften der abstrakten Kryptographie direkt auf die `byte[ ]` Arrays abgebildet.

Kapitel 11

Anwendung auf Chipkarten

Dieses Kapitel zeigt die Umsetzung des Verfeinerungskonzeptes für Programme auf Chipkarten. Dazu werden zunächst die Besonderheiten und Restriktionen der verwendeten JavaCard Plattform sowie die Hardware kurz beschrieben. Im Anschluss werden nötige Anpassungen an der Implementierung der Kodierungs- und Dekodierungsschicht sowie für die Einbettung in die JavaCard Laufzeitumgebung erklärt. Die in diesem Kapitel beschriebenen Arbeiten wurden - illustriert am Mondex Beispiel - in [67] publiziert.

11.1 Chipkarten und Smart Cards

Chipkarten haben inzwischen eine sehr hohe Verbreitung erreicht. Ihre Einsatzgebiete variieren von einfachen Kundenkarten bis hin zu komplexen Bezahlsystemen wie z.B. der Geldkarte [59], oder auch der elektronischen Gesundheitskarte [29] und Anwendungen im Pay-TV. Auch als Identifikations- und Zugangsicherungssysteme oder zur Zeiterfassung sind Chipkarten beliebt. Die größte Verbreitung hat sicherlich die SIM-Karte[1] in allen Mobiltelefonen [90].

Chipkarten enthalten dabei in ihrer einfachsten Form lediglich einen Speicher zur persistenten Hinterlegung von Daten auf der Karte. Die derzeitige Krankenversichertenkarte oder die Telefonkarte fällt in diese Kategorie. Maximal ein Passwortschutz zum Schutz vor unbefugtem Zugriff auf die Daten ist hier noch zusätzlich möglich. Während solche Karten auch z.B. für einfache Zeiterfassungssysteme noch ausreichend sein können, werden bei allen ernsthafteren sicherheitskritischeren Anwendungen auch Rechenkapazitäten auf dem Chip benötigt. Insbesondere kryptographische Operationen müssen dabei unterstützt werden. Karten, die solche Rechenkapazitäten bereitstellen, werden dabei als Smart Cards bezeichnet. Smart Cards besitzen auf engstem Raum alle wesentlichen Bestandteile eines Computers: Prozessor, persistenten Speicher und Programmspeicher sowie einen Eingabe- und einen Ausgabekanal. Die Karten besitzen dabei keine eigene Stromversorgung, sind also bei ihren Berechnungen auf die Stromversorgung durch ein Terminal angewiesen.

Die Kommunikation mit Smart Cards kann dabei kontaktbehaftet oder kontaktlos stattfinden.

[1] Subscriber Identity Module

Der elektronische Reisepass [46] ist dabei das bekannteste Beispiel kontaktloser Chipkartenanwendungen.

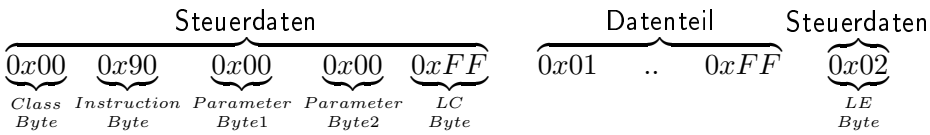
Der wesentliche Vorteil beim Einsatz von Smart Cards für sicherheitskritische Anwendungen ist dabei ihre Eigenschaft, Daten manipulationsresistent und nicht auslesbar speichern zu können ("tamper-proof"). Im Gegensatz zu herkömmlichen Speichermedien und Endgeräten kann bei Chipkarten im Allgemeinen davon ausgegangen werden, dass auf der Karte gespeicherte Informationen auf einem anderen als über die Eingabe-/Ausgabeschnittstelle vorgesehenem Wege nicht ausgelesen werden können. Zwar existieren einige physikalische Angriffsmethoden auf Chipkarten (wie z.B. die Differential Power Analysis [108]). Der Aufwand für diese Angriffe ist allerdings meist sehr hoch. Smart Cards können daher als manipulationssichere Endgeräte angenommen werden. Dies prädestiniert diese Karten als Endgeräte für sicherheitskritische Anwendungen, da ein Angreifer z.B. nicht den gespeicherten Geldbetrag auf der Geldkarte, den Schlüssel zum Zugang zum Bezahlfernsehen oder die elektronischen Passdaten einfach auslesen oder (schlimmer noch) ändern kann. Lediglich das jeweils verwendete Kommunikationsprotokoll bietet Möglichkeiten zum Angriff. Dies ist genau der Fokus der hier vorgestellten Arbeiten.

11.2 Besonderheiten von Smart Cards

Weitführende technische Details zu Chipkarten sind für die folgenden Betrachtungen nicht wesentlich und werden daher hier nicht aufgeführt, [147] gibt dazu z.B. einen umfassenden Überblick. Im Folgenden beschreiben wir lediglich die Charakteristika von Chipkarten, die für unsere Anwendungsdomäne und Abstraktionsebene wichtig sind:

11.2.1 Kommunikation

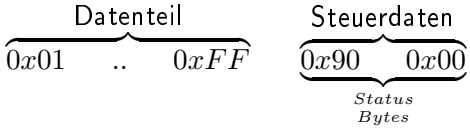
Die Kommunikation mit Smart Cards findet mittels eines standardisierten Protokolls statt, das auf Byte Folgen beruht. Smart Cards arbeiten dabei lediglich reaktiv, beginnen also von sich aus keine eigenen Berechnungen. Sämtliche Berechnungen finden als Reaktion auf eine Eingabe von einen Terminal statt. Das spiegelt sich auch im Kommunikationsverhalten wieder: Ein Terminal schickt zunächst eine Eingabe an die Karte, diese rechnet mittels dieser Eingabe und schickt eine Ausgabe zurück. Die Kommunikation findet damit mittels Application Protocol Data Units (APDU) statt. Eine APDU ist eine Dateneinheit, die vom Terminal zur Karte oder umgekehrt verschickt wird. Die APDUs, die das Terminal verschickt, heißen Command APDUs. Die Antworten der Karte werden als Response APDUs bezeichnet. Eine Command APDU hat dabei exemplarisch die folgende Form:



Das Class Byte gibt dabei an, für welche Anwendungsdomäne die APDU intendiert ist. Es spezifiziert damit den verwendeten Befehlssatz genauer. Z.b. beschreibt das Class Byte `0x00` Systemkommandos nach dem Standard ISO7816. Das folgende Instruction Byte gibt dann das eigentlich Kommando an, das verarbeitet werden soll (bei einer elektronischen Geldbörse

wäre das z.B. typischerweise "Laden" oder "Abheben"). Es folgen zwei Parameter Bytes, die das Kommando genauer beschreiben können. Schließlich folgt das LC Byte, das die Länge der übertragenen Nutzdaten angibt. Diese Nutzdaten folgen im Anschluss. Es können bis zu 255 Byte Nutzdaten übertragen werden. Das letzte Byte (genannt LE Byte) gibt die Länge der erwarteten Antwort an (oder wird weggelassen, wenn keine Antwort erwartet wird). Werden keine Nutzdaten übertragen, ersetzt das LE Byte das LC Byte.

Response APDUs der Chipkarte sind lediglich aus Daten und einer Statusangabe zusammengesetzt:



Die letzten beiden Bytes geben dabei den Status der Antwort an. Im ISO7816 Standard sind hier diverse Statusangaben vordefiniert. `0x90 0x00` bezeichnet z.B. eine erfolgreiche Verarbeitung der Eingabe, `0x6A 0x84` gibt dagegen exemplarisch für die Fehlerfälle an, dass der Karte bei Verarbeitung der Eingabe der interne Speicher ausgegangen ist.

11.2.2 Ressourcenbeschränkung

Chipkarten sind sehr ressourcenbeschränkte Endgeräte. Die Mikroprozessoren auf den Karten arbeiten üblicherweise mit 8 oder 16 bit. Dementsprechend stehen Integer Operationen nicht zur Verfügung.

Entsprechend dem beschränkten Kapazitäten des Prozessors ist auch der verfügbare Speicher sehr beschränkt. Auf derzeitigen Chipkarten wie z.B. der Giesecke & Devrient Smart Cafe Expert 3 Karten - welche auch für die hier vorgestellte Mondex Implementierung verwendet wurden - sind z.B. 72 kB EEPROM Speicher enthalten. Das bedeutet, dass Programme für Chipkarten keine großen Datenmengen speichern können. Auch der Arbeitsspeicher ist mit nicht einmal 5 kB sehr knapp.

Gerade im Hinblick auf die Datenübertragungsmethodik in dieser Arbeit, die Pointerstrukturen dekodiert und kodiert, muss also Rücksicht auf die Ressourcenbeschränkung der Chipkarten genommen werden. Eine entsprechende Adaption findet sich in Kap. 11.3.

Schließlich wird für kryptographische Operationen, die im Umfeld von Chipkartenanwendungen naturgemäß häufig eingesetzt werden, zusätzlich spezielle Hardware auf dem Chip integriert, die z.B. die RSA oder 3DES Algorithmen (s. [166]) effizient berechnen kann.

11.2.3 Programmierung

Die Programmierung von Chipkarten kann auf diverse Arten erfolgen. In der Praxis eingesetzt werden häufig Implementierungen in C oder auch in Maschinencode. Dazu kommt die Möglichkeit, Chipkarten in JavaCard (momentan ist Version 2.2.2 aktuell [175]) zu programmieren, einem Dialekt der Java Sprache. JavaCard besitzt alle Sprachkonstrukte der Java Standard Edition in Version 1.4 [100] (also z.B. Klassen- und Interfacedeklarationen, Methodendeklarationen, Schleifen, ...) , mit folgenden Ausnahmen:

- Die primitiven Typen beschränken sich auf `boolean`, `byte` und `short` (insbesondere gibt es also keinen `int` Typ).
- Es gibt keine Unterstützung für Multithreading (kein `synchronized` bzw. keine `Thread` Klasse).
- Die API ist sehr eingeschränkt und definiert lediglich einige Exceptions (wie z.B. eine `NullPointerException`) sowie Klassen zur Kommunikation und Kryptographie. Es gibt z.B. keinen `String` Typ oder komplexe Container wie `Vector`.
- Es gibt keine Garbage Collection (weder implizit noch explizit). Es ist also nicht möglich, zur Laufzeit einer Chipkartenanwendung beliebig viele Objekte dynamisch zu allokieren, da sonst früher oder später der Speicher der Karte erschöpft sein wird.

Für JavaCard Anwendungen werden nun spezielle Karten benötigt, die ein Betriebssystem enthalten, das in der Lage ist, Java Bytecode zu interpretieren und auch die Ein- und Ausgabefunktionen der Karte anzusteuern. Für eine einheitliche Programmierung dieser Karten sieht dabei die JavaCard API eine generische Programmierschnittstelle für Chipkartenanwendungen vor. Alle Chipkartenanwendungen müssen Subklassen der generischen API Klasse `Applet` sein. `Applet` definiert insbesondere eine `process` Methode, die eine `APDU` (als Java Objekt, das das Eingabe/Ausgabe `byte[ ]` kapselt) als Parameter bekommt. Alle konkreten JavaCard Anwendungen müssen dabei diese `process` Methode überschreiben:

```
1 package javacard.framework;
2
3 public abstract class Applet{
4    ...
5    public void process(APDU apdu);
6 }
```

Das Betriebssystem der Karte kann dabei nun mehrere Anwendungen auf einer Karte verwalten. Jeweils eine Anwendung ist dabei immer die derzeit aktive Anwendung. Eine spezielle *select* APDU kann zur Wechsel der jeweils aktiven Anwendung benutzt werden. Das Betriebssystem der Karte ruft bei Empfang einer Command APDU von einem Terminal die `process` Methode der derzeit aktiven Anwendung auf.

Alle Felder einer Klasse werden dabei (im Gegensatz zu lokalen Variablen von Methoden) im EEPROM abgespeichert. Insbesondere bedeutet dies, dass Werte, die in Feldern gespeichert werden, persistent vorhanden sind. Wird die Stromversorgung der Karte unterbrochen und später wieder hergestellt, beinhalten diese Felder immer noch die gleichen Werte. Ein Speichern, z.B. in ein Dateisystem, ist daher bei Chipkarten nicht nötig.

11.3 Adaption der Methodik

Die bisherige Implementierungsmethodik für Kommunikationsprotokolle, die in dieser Arbeit verwendet wurde, stellte die abstrakten `Document` Typen in der Implementierung durch `Document` Klassen dar. Zusätzlich wurde im letzten Kapitel eine Kodierungsschicht eingeführt, die eine Transformation dieser Klassen in `byte[ ]` vornehmen kann. Da auch Chip-

karten mit Byte Sequenzen kommunizieren, erscheint eine Benutzung dieser Transformationsschicht auch für Chipkarten interessant. Allerdings allokiert die bisherige Transformationsschicht beim Kodieren und Dekodieren laufend neuen Speicher. Da Chipkarten keine Garbage Collection besitzen, führt dies zu Speicherüberläufen. Auch ist die Maximallänge des Datenteils einer APDU mit 255 Byte beschränkt. Lange Dokumente können so nicht auf einmal verschickt werden. Darüber hinaus benutzen einige Operationen auf den Java `Document` Klassen `int` Werte z.B. für Indizierung oder Iteration. Es müssen also einige Anpassungen vorgenommen werden.

Dabei ist folgendes zu beachten: die nächste Version des JavaCard Standards (JavaCard 3.0) befindet sich derzeit in der Ausarbeitung durch das JavaCard Forum ([56]). Für diese Version sind dutzende neuer Features angekündigt, unter anderem sind TCP/IP artige Kommunikation mit der Chipkarte, Remote Method Invocation, Multi-Threading, String Support sowie eine Überarbeitung des Speichermodells angedacht. Diese Änderungen würden (sollten sie tatsächlich umgesetzt werden), die gesamte Chipkartenprogrammierung ändern und insbesondere auch oben genannte Schwierigkeiten beseitigen. Insbesondere wäre mit Umsetzung dieser Features die in bisher vorgestellte Programmiermethodik für Kommunikationsprotokolle auch direkt für JavaCard nutzbar. Da JavaCard 3.0 jedoch noch nicht veröffentlicht wurde, geben wir in diesem Kapitel die Adaptionen an, die am bisherigen Konzept vorgenommen werden müssen, geben jedoch keine formalen Beweise an.

11.3.1 Integration in die JavaCard API

Für die Cindy Anwendung haben wir bereits ein Kommunikationsinterface bereitgestellt, das `send` und `receive` Methoden für Java `Document` Objekte bereitstellt (s. dazu Kap. 5.3.1). Insbesondere haben wir dort verschiedene Eingabeports (z.B. für MMS oder Benutzereingaben) vorgesehen. Chipkarten besitzen demgegenüber nur eine Eingabe- und Ausgabeschnittstelle über APDUs. Daher können wir eine einfachere Variante des Kommunikationsinterfaces für Chipkarten verwenden. Dies ist das `SimpleComm` Interface:

```
1 public interface SimpleComm {
2     public Document receive();
3     public void send(Document d);}
```

Dieses Interface kann nun mittels einer leicht adaptierten Implementierung für die Transformationsschicht für Kodierung und Dekodierung implementiert werden, s. dazu weiter unten. Zusätzlich müssen wir aber für die JavaCard Plattform unsere Protokollfunktionalität auch in eine Subklasse der `javacard.framework.Applet` Klasse einbetten, die über APDUs kommuniziert. Die bisherige Implementierung der Protokollfunktionalität mittels einer Klasse `Protocol` mit einer `step` Methode, die dann obige `send` und `receive` Methoden benutzt, kann dabei unverändert übernommen werden, wenn wir einen Wrapper für die JavaCard spezifischen Besonderheiten angeben, der APDU Kommunikation auf Dokumenten-orientierte Kommunikation umsetzt. Die Hauptklasse unserer JavaCard Protokollimplementierungen ist daher ein Wrapper der folgenden (schematischen) Form:

```
1 public class JavaCardAppletWrapper
2   extends javacard.framework.Applet
3   implements SimpleComm{
```

```
4
5     Document indoc;
6     Protocol protocolimpl;
7
8     public Applet(){
9        protocolimpl = new Protocol(this, ...);
10    }
11
12    public void process(APDU apdu){
13       ...
14       //dekodieren der Command APDU
15       indoc = decode(apdu.getBuffer());
16
17       //Aufruf der Protokollimplementierung
18       protocolimpl.step();
19    }
20
21    public Document receive(){
22       return indoc;
23    }
24
25    public void send(Document d){
26       byte[] outbytes = encode(d);
27
28       //kopieren der kodierten Ausgabe in den APDU-Buffer
29       Util.arraycopy(outbytes, 0,
30                      apdu.getBuffer(),0, outbytes.length)
31       //und senden dieses Buffers
32       apdu.setOutgoingAndSend(0, outbytes.length);
33    }
34 }
```

Die Klasse `JavaCardAppletWrapper` erweitert zunächst die `Applet` Klasse aus der JavaCard API. Sie muss also die Einstiegsmethode `process(APDU)` überschreiben, die bei Vorliegen einer Eingabe vom Betriebssystem aufgerufen wird. Diese Methode erhält als Parameter bereits die Eingabe APDU - hier unterscheidet sich die JavaCard Programmierung von herkömmlichen Ein-/Ausgabemechanismen, es gibt zunächst kein explizites `receive`, die Eingabe wird immer gleich zu Beginn mitgeliefert. Um dies auf unsere Programmiermethodik mit explizitem `send` und `receive` umzusetzen, dekodieren wir die in der APDU enthaltene Eingabe (z.B. mit der gleichen Kodierungsvorschrift, die auch schon in Kap. 10 verwendet wurde) und speichern das Ergebnis temporär im Feld `indoc`. Im Anschluss wird die `step` Methode einer Protokollimplementierung (hier die Klasse `Protocol`) aufgerufen, die dann die eigentliche Funktionalität enthält. Diese Klasse wurde im Konstruktor des `JavaCardAppletWrapper` erzeugt und erhielt die Instanz des `JavaCardAppletWrapper` als Parameter. Da der Wrapper auch das Kommunikationsinterface `SimpleComm` implementiert, kann damit `Protocol` die `receive` und `send` Methoden des Wrappers aufrufen. Innerhalb `receive` wird nun lediglich der initial dekodierte Eingabewert `indoc` zurückgegeben. `send` verwendet nun `encode`,

um eine Kodierung des Ausgabedokumentes zu erstellen und diese mittels der APDU Kommunikation als Response APDU an das Terminal zurückzusenden. Alle Zusammenhänge sind auch nochmals im Sequenzdiagramm in Abb. 11.1 dargestellt. Diese Abbildung illustriert zusätzlich auch die Abläufe einer terminalseitigen Protokollimplementierung, die spiegelbildlich zu den gerade beschriebenen Abläufen funktioniert. Dabei sei explizit auf den bisherigen Kommunikationsablauf für Anwendungen ohne Chipkarte verwiesen, der bereits in Kap. 8.8 beschrieben wurde.

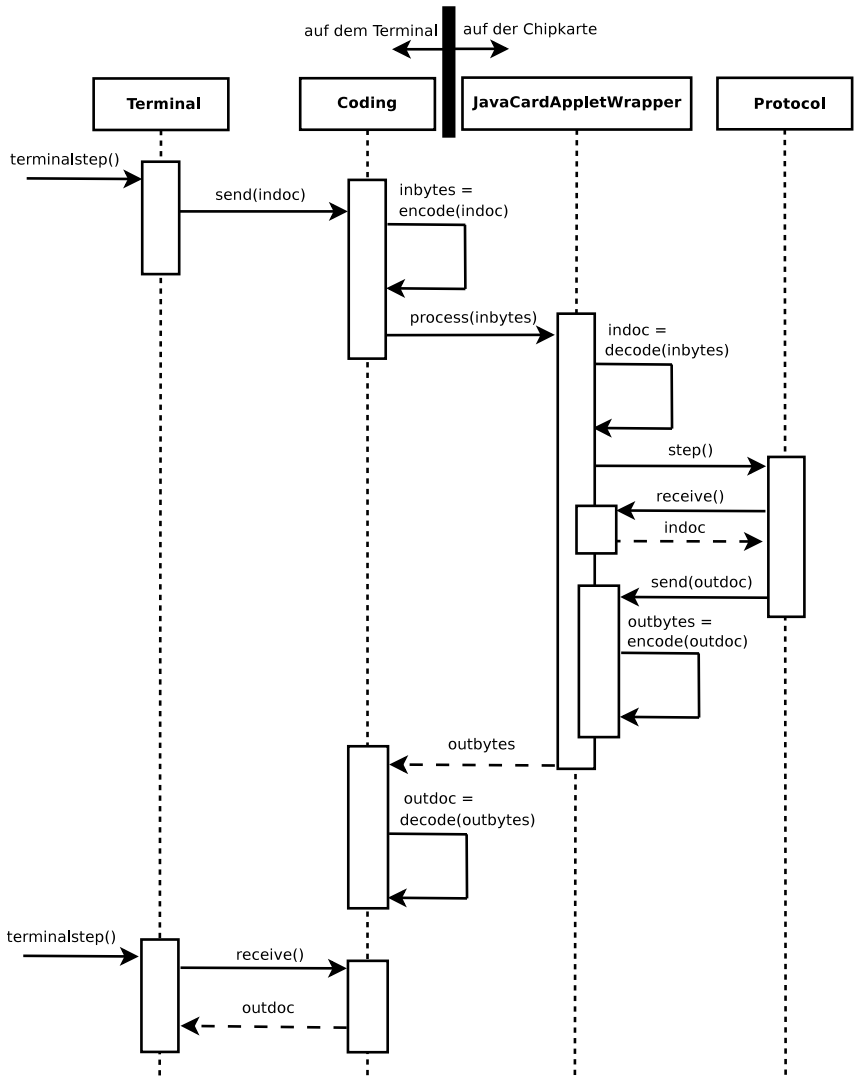


Abbildung 11.1: Kommunikation bei einer Chipkartenimplementierung

Abb. 11.1 vereinfacht dabei leicht und abstrahiert von der Kommunikation über APDUs. Das Senden einer APDU an das Chipkartenprogramm wird als einfacher Aufruf von `process` dargestellt. Die Antwort - die normalerweise von der Karte aktiv mittels `setOutgoingAndSend` verschickt wird - wird als einfache Rückgabe dieses Aufrufs dargestellt.

Es ist insbesondere auch zu beachten, dass die Terminalimplementierung hier nicht so funktioniert, wie es bei herkömmlichen Chipkartenanwendungen zu erwarten ist. Die PROSECCO Spezifikationen funktionieren asynchron. Das bedeutet, dass ein Schritt eines Agenten üblicherweise aus dem Empfangen einer Nachricht und dem Senden einer Nachricht besteht. Die darauffolgende Antwort auf letztere Nachricht wird erst im nächsten Schritt des Agenten empfangen, der nicht unbedingt unmittelbar danach erfolgen muss (es könnte z.B. der Angreifer oder ein anderer Protokollteilnehmer vorher an die Reihe kommen). Ein JavaCard-Terminal schickt demgegenüber eine Nachricht an die Karte und blockiert solange, bis es als direktes Ergebnis dieses Sendens eine Antwort von der Karte bekommt. JavaCard Kommunikation kann also terminalseitig als `response = send(command)` aufgefasst werden. Eine solche Kommunikationsstruktur könnten wir nicht ohne weiteres als Verfeinerung der PROSECCO Ebene auffassen, da hier das Senden einer Nachricht und das Empfangen der Antwort *zwei* Schritte sind und dazwischen insbesondere der Angreifer, aber auch beliebige andere Teilnehmer aktiv sein können und auch müssen (sonst würden wir evtl. dort Angriffe ignorieren). Daher muss auch eine terminalseitige Kommunikationsinfrastruktur (in Abb. 11.1 die Klasse `Coding`) die Antwort der Chipkarte zwischenspeichern - analog zum Zwischenspeichern der Eingabe auf der Chipkarte vor dem Aufruf von `step`. Erst im nächsten Protokollschritt (der zweite `terminalstep()` in Abb. 11.1) wird die Antwort der Karte dann tatsächlich zurückgeliefert.

11.3.2 Speicherallokation

Auf derzeitigen JavaCard Smart Cards wird keine Garbage Collection unterstützt. Alle Objekte, die irgendwann angelegt wurden, bleiben im Speicher bestehen. Insbesondere bedeutet dies, dass die Implementierung der Transformationsschicht, die sowohl bei der Kodierung als auch bei der Dekodierung häufig Speicher allokiert, adaptiert werden muss. Alle Objekte, die zur Laufzeit des Chipkartenprogramms benötigt werden, sollten damit zu Beginn der Laufzeit angelegt werden. Dies ist bei der Transformationsschicht für zwei verschiedene Daten wichtig: die allokierten `byte[ ]` zur Kodierung und die allokierten `Document` Objekte bei der Dekodierung.

Zur Lösung dieses Problems sehen wir jeweils einen Pool vor, der bei Installation der Applikation groß genug initialisiert wird. Bei der Kodierung ist dies einfach:

Auswirkungen auf die Kodierung

Die bisherige Kodierungsfunktion (s. Kap. 10.4) `encode` allokierte für jedes Teildokument einer Liste ein neues `byte[ ]` und hängte diese `byte[ ]` schließlich zusammen, um die Gesamtkodierung zu erhalten. Für eine Version für Chipkarten können wir nur ein `byte[ ]` verwenden, das lang genug sein muss, um alle Dokumente des jeweiligen Kommunikationsprotokolls kodiert darzustellen. Wir verändern also die Kodierungsfunktion so, dass sie statt auf einzelnen Arrays auf diesem einen globalen Puffer Array arbeitet.

Statt einer Konkatenation der Zwischenergebnisse verwenden wir einen Index in den Puffer, um die jeweils nächste Stelle zu finden, ab der die Kodierung des nächsten Dokumentes in einer Liste zu erfolgen hat. Um diesen Index jeweils richtig inkrementieren zu können, geben wir jeweils die Länge des aktuell zu kodierenden Dokumentes als Ergebnis der Kodierungsfunktion zurück.

Zu beachten ist nun, dass die Kodierung nur noch dann korrekt funktioniert, wenn der Puffer für die kodierten Ausgaben groß genug ist.

Auswirkungen auf die Dekodierung

Bei der Dekodierung ist keine so einfache Lösung mittels eines einzigen globalen Puffers möglich. Hier müssen nun einzelne Dokumente auf Vorrat für die Dekodierung angelegt werden.

Dazu verwenden wir einen Dokumentenpool, der groß genug sein muss, um alle im Protokoll vorkommenden Dokumente darstellen zu können. Eine einfache Lösung wäre es, alle Nachrichten des Protokolls im vornherein als `Document` Pointerstruktur zu allokieren und bei Empfang eines `byte[ ]` eine dazu passende Struktur auszuwählen und mit Werten zu füllen. Der Speicherplatzbedarf dazu wäre allerdings recht hoch, pro Verarbeitungsschritt wird nur ein kleiner Teil der vorallokierten Nachrichten verwendet. Wir wählen hier eine andere Lösung. Wir allokieren nur die einzelnen Bestandteile der Nachrichten vor. Die Allokation umfasst dabei genau so viele einzelne Dokumente, wie gleichzeitig in einer beliebigen wohlgeformten Eingabenachricht vorkommen können. Nehmen wir zur Illustration an, ein Protokoll für z.B. eine Geldkarte beinhalte die folgenden Eingabenachrichten für die Karte:

Laden von Geld:	Doclist(IntDoc(LOAD) + IntDoc(betrag))
Abheben von Geld:	Doclist(IntDoc(WITHDRAW) + SecretDoc(password) + IntDoc(betrag))
Abfrage des Kontostands:	Doclist(IntDoc(GETBALANCE) + SecretDoc(password))

wobei LOAD, WITHDRAW und GETBALANCE einfache ein Byte lange Werte sind. Das password soll 8 byte lang sein. Der betrag ist ein `short` Wert, also zwei Byte lang.

Die Einzeldokumente, die in diesem Protokoll innerhalb einer Eingabenachricht potentiell gleichzeitig vorkommen können, sind also:

- eine `Doclist` der Länge 2 Dokumente (LOAD und GETBALANCE Nachricht)
- eine `Doclist` der Länge 3 Dokumente (WITHDRAW Nachricht)
- ein `IntDoc` der Länge 1 Byte (die Kommandos LOAD, WITHDRAW und GETBALANCE)
- ein `IntDoc` der Länge 2 Bytes (der Betrag bei LOAD und WITHDRAW)
- ein `SecretDoc` der Länge 8 Bytes (das Passwort)

Insgesamt benötigen wir also als Dokumentenpool für die Darstellung aller potentiellen wohlgeformten Eingaben also genau diese fünf Dokumente (und insbesondere *nicht* eine Struktur aus 3 Dokumenten für LOAD, 4 Dokumenten für WITHDRAW und 3 Dokumenten für GETBALANCE, insgesamt also mit 10 Dokumenten doppelt so viele Objekte).

Die Dekodierungsimplementierung aus Kap. 10 wird jetzt so angepasst, dass an allen Stellen, an denen zuvor ein `new` Aufruf für ein `Document` erfolgt, jetzt ein Zugriff auf eines der Dokumente aus dem Pool stattfindet. Sollten andere Dokumente als die im Pool enthaltenen empfangen werden, gehören diese offensichtlich nicht zum Protokoll und können analog zum Fehlerdokument $\perp$ behandelt werden. Diese Dokumente können schließlich nur von einem Angriffsversuch auf das Protokoll stammen.

Pooling in der Implementierung

Der Dokumentenpool, der für die Transformationsschicht eingeführt wird, muss bei Initialisierung der Anwendung ebenfalls geeignet initialisiert werden. Dabei muss angegeben werden, welche Dokumente mit welcher Länge für das Programm benötigt werden. Diese Initialsierungsinformation wird für die Verfeinerungsbeweise als korrekt angenommen.

Die Initialisierungsinformationen werden beim Installieren eines Applets auf der Chipkarte als Byte Liste übergeben. Diese besteht dabei jeweils aus Paaren `(DokumentTyp, Länge)`. Unter Verwendung von symbolischen Konstanten wäre also die Initialisierung für obiges Beispiel

```
byte[] initdata = new byte[]{DOCLISTTYPE, 2,
                             DOCLISTTYPE,3,
                             INTDOCTYPE, 1,
                             INTDOCTYPE 2,
                             SECRETDOCTYPE, 8}
```

Konkret wird der Pool implementiert, indem für jeden Dokumententyp ein Array mit den zur Verfügung stehenden Dokumenten angelegt wird. Wir haben also:

```
1 public class DocumentPool{
2   private static IntDoc[] intdocs;
3   private static IntDoc[] noncedocs;
4   private static IntDoc[] secretdocs;
5   ...
6 }
```

Zusätzlich wird in einer weiteren boolschen Liste gespeichert, welche dieser Dokumente bereits benutzt sind und welche noch zur Verwendung stehen. Die Referenzierung erfolgt dabei über den jeweiligen Index in dem Array (`intdoc[i]` hat Status `usedIntDocs[i]`):

```
1   private static boolean[] usedIntDocs;
2   private static boolean[] usedNonceDocs;
3   private static boolean[] usedSecretDocs;
4   ...
```

Die Allokation eines Dokumentes aus dem Speicher erfolgt, indem der Typ des Dokumentes und die gewünschte Länge angegeben wird und im Pool nachgesehen wird, ob ein passendes Dokument noch verfügbar ist. Ist kein solches verfügbar, wird `null` zurückgegeben (die Entsprechung des $\perp$ Dokumentes für den Fehlerfall):

```
1 public static IntDoc newIntDoc(short length) {
2    for(short i=0;i<intdocs.length;i++) {
3       if(   intdocs[i].getValue().length == length
4          && usedIntDocs[i] == UNUSED) {
5             usedIntDocs[i] = USED;
6             return intdocs[i];
7       }
8    }
9    return null;}
```

Die grundlegende Funktionsweise des Dokumentenpools ist nun (s. dazu auch ergänzend Abb. 11.1):

Zu Beginn jedes Protokollschritts wird der Pool komplett zur Verfügung gestellt (alle Dokumente werden als verfügbar markiert). Dann wird die Dekodierungsfunktion für die `byte[ ]` Eingabe aufgerufen. Diese Funktion verwendet nun die verfügbaren Pool-Dokumente für die Darstellung der Eingabe als Pointerstruktur. Dann wird die Protokollfunktionalität aufgerufen (`step` Methode). Diese produziert eine `Document` Ausgabe (wird nicht über den Pool behandelt, sondern von der Anwendung selbst vorgehalten). Die Ausgabe wird in den oben beschriebenen `byte[ ]` Puffer der Transformationsschicht kodiert (dessen Initialisierungsgröße als ausreichend angenommen wird) und gesendet.

Damit benötigen wir insbesondere eine Funktion `clearPool` in `DocumentPool`, die den Status aller Dokumente zurücksetzt. Ebenso muss die Initialisierung des Pools bei Installation der Anwendung einmalig möglich sein. Der `JavaCardAppletWrapper` muss also in der `process` Methode angepasst werden. Bisher haben wir die Header Informationen der Command APDU dort nicht beachtet. Nun führen wir zwei verschiedene Kommando Bytes ein: `STEP` für die Durchführung eines Protokollschrittes und `INIT` für die (einmalige) Initialisierung:

```
1  public class JavaCardAppletWrapper
2    extends javacard.framework.Applet
3    implements SimpleComm{
4
5    ... //s. oben
6
7    public void process(APDU apdu){
8       byte[] buffer = apdu.getBuffer();
9
10      if(buffer[ISO7816.OFFSET_INS] == INIT){
11       DocumentPool.init(buffer);
12      }
13      else if(buffer[ISO7816.OFFSET_INS] == STEP){
14       DocumentPool.clearPool();
15       indoc = decode(apdu.getBuffer());
16       protocolimpl.step();
17      }
18   }
19 }
```

Durch die Funktionsweise des Pools muss zudem sichergestellt sein, dass die Dokumente, die zur Laufzeit zurückgegeben werden, aus Sicht des benutzenden Programmteils nicht von wirklich neuen Dokumenten zu unterscheiden sind. Insbesondere müssen die Pointerstrukturen frei von Sharing sein und auch disjunkt zu allen bisher vorkommenden Referenzen. Dies bedeutet zweierlei:

- Die Dekodierungsfunktion der Transformationsschicht darf keine Dokumente mit Sharing innerhalb der Struktur erzeugen

- Die eigentliche Protokollimplementierung darf keine Referenzen auf Dokumente aus dem Pool speichern

Erstere Anforderung kann auf einer adaptierten Implementierung der Transformationsschicht nachgewiesen werden. Zweitere Anforderung kann syntaktisch auf der Protokollimplementierung überprüft werden (im wesentlichen dürfen Referenzen aus dem Pool nicht auf der rechten Seite einer Zuweisung vorkommen).

Pooling in der Spezifikation

Die beiden genannten Änderungen haben nun eine Auswirkung auf die beiden Eigenschaften der Transformationsschicht, die bereits in vorangegangen Kapiteln vorgestellt wurden. Als Eigenschaften für einen Verfeinerungsbeweis haben wir bisher immer die beiden Eigenschaften FROMSTORE-send und TOSTORE-receive verwendet (s. Kap.6 und 8). TOSTORE-receive besagte, dass die Java `receive` Implementierung genau die Eingabe als Pointerstruktur zurückliefert, die auch in der abstrakten Eingabequeue des Agenten als abstraktes Dokument befand. Dies stimmt nun nur noch, wenn das Eingabedokument mit dem Dokumentenpool darstellbar ist. Ansonsten wird - wie oben beschrieben - `null` als Entsprechung von $\perp$ zurückgeliefert. Analog dazu wird `send` nur noch die Dokumente versenden können, die mit dem `byte[ ]` Puffer der Transformationsschicht darstellbar sind. Hier verwenden wir als Annahme, dass der Puffer groß genug gewählt wurde, um alle Dokumente zu kodieren, die im Protokoll vorkommen (die Maximalgröße einer APDU sollte hier ausreichen - falls nicht wird in Kap. 11.3.3 eine Lösung vorgeschlagen).

Um diese Zusammenhänge formal auszudrücken, führen wir ein neues Prädikat poolrestriction : Document ein, das angibt, ob ein abstraktes Dokument mit dem Dokumentenpool der Applikation darstellbar ist. Dieses Prädikat wird zur zusätzlichen Vorbedingung für die Korrektheitseigenschaften der Transformationsschicht. Die Axiomatisierung ist nun applikationsspezifisch und gibt die Anforderungen des jeweiligen Protokolls wieder. Für eine generische Spezifikationsmethodik empfiehlt sich folgende Art der Axiomatisierung: Für jeden Dokumententyp wird eine Zählfunktion verwendet, die die Zahl der Vorkommen dieses Typs mit gegebener Länge in dem jeweilig zu testenden Dokument angibt. Damit kann ausgedrückt werden, dass nicht mehr als die im Pool verfügbaren Dokumente einer bestimmten Länge verwendet werden können. Dazu wird pro Dokumententyp eine weiteres Prädikat spezifiziert, das testet, ob in einem gegebenen Dokument andere Längen als die erlaubten Längen verwendet werden. Generisch erhalten wir so folgende Spezifikation:

SPEZIFIKATION:

$$
\begin{aligned}
& \mathsf{poolrestriktion(doc)} \\
\leftrightarrow \quad & \mathsf{countInt(intlength}_1\mathsf{, doc)} \leq \mathsf{i}_1 \\
& \wedge \ldots \\
& \wedge \mathsf{countInt(intlength}_n\mathsf{, doc)} \leq \mathsf{i}_n \\
& \wedge \ldots \\
& \wedge \mathsf{countNonce(noncelength}_1\mathsf{, doc)} \leq \mathsf{j}_1 \\
& \wedge \ldots \\
& \wedge \mathsf{countNonce(noncelength}_m\mathsf{, doc)} \leq \mathsf{j}_m
\end{aligned}
$$

$\wedge$ ... // usw. für alle vorkommenden Dokumententypen
$\wedge$ ...
$\wedge$ noOtherInt(intlength$_1$ + ... + intlength$_n$, doc)
$\wedge$ noOtherNonce(noncelength$_1$ + ... + noncelength$_m$, doc)
$\wedge$ ... // usw. für alle Dokumententypen

countInt(i, doc) zählt dabei alle IntDoc Vorkommen innerhalb von doc, deren Länge i ist. Analog countNonce für Nonces und für alle anderen Typen. noOtherInt(ilist, doc) ist true, wenn jedes Vorkommen eines IntDocs innerhalb von doc eine Länge besitzt, die in ilist vorkommt.

Für das obige Geldkartenbeispiel hätten wir damit z.B.:

SPEZIFIKATION:

poolrestriktion$_{\text{geldkarte}}$(doc)
$\leftrightarrow$ countInt(1, doc) $\leq$ 1
$\wedge$ countInt(2, doc) $\leq$ 1
$\wedge$ countSecret(8, doc) $\leq$ 1
$\wedge$ countDoclist(2, doc) $\leq$ 1
$\wedge$ countDoclist(3, doc) $\leq$ 1
$\wedge$ noOtherInt(1 + 2, doc)
$\wedge$ noOtherNonce([ ], doc)
$\wedge$ noOtherSecret(8, doc)
$\wedge$ noOtherKey([ ], doc)
$\wedge$ noOtherHash([ ], doc)
$\wedge$ noOtherEnc([ ], doc)
$\wedge$ noOtherSig([ ], doc)
$\wedge$ noOtherDoclist(2 + 3, doc)

Pooling im Refinement

Wichtig für konkrete Verfeinerungsbeweise ist die in Kap. 6 vorgestellte und in Kap. 8 bewiesenen Eigenschaft TOSTORE-receive. Bei dieser wird nun die poolrestriction als zusätzliche Vorbedingung eingeführt. Für Chipkarten gilt damit:

THEOREM:

TOSTORE-receive-smartcards-ok:
.mode $\in$ st $\wedge$ st[.mode] = noval
$\wedge$ validrefnotnull("SimpleComm", r, SimpleComm, st)
$\wedge$ inputs(agent)(i) $\neq$ [ ]
$\wedge$ $\langle$TOSTORE(agent, inputs; cstore)$\rangle$ (cstore = cstore$_1$)

$$
\begin{array}{ll}
& \wedge\ \text{poolrestriction(inputs(agent)(i).first)} \\
& \wedge\ \text{st} = \text{cstore}_1\text{(agent)} \\
& \langle \text{st};\ r_1 = \text{r.receive(i)} \rangle\ (\text{st} = \text{st}_0 \wedge r_1 = r_2) \\
\rightarrow & \quad \text{java2doc}(r_2,\ \text{st}_0) = \text{inputs(agent)(i).first} \\
& \wedge\ \text{st} \subseteq \text{st}_0 \\
& \wedge\ \text{validref("docs2store"}, r_2, \text{ Document, } \text{st}_0) \\
& \wedge\ \text{validDoc}(r_2,\ \text{st}_0)
\end{array}
$$

Im Fehlerfall wird $\perp$ bzw. `null` zurückgegeben:

THEOREM:

TOSTORE-receive-smartcards-fail:

$$
\begin{array}{ll}
& \text{.mode} \in \text{st} \wedge \text{st[.mode]} = \text{noval} \\
& \wedge\ \text{validrefnotnull("SimpleComm"}, r, \text{ SimpleComm, st)} \\
& \wedge\ \text{inputs(agent)(i)} \neq [\,] \\
& \wedge\ \langle \text{TOSTORE(agent, inputs; cstore)} \rangle\ (\text{cstore} = \text{cstore}_1) \\
& \wedge\ \neg\ \text{poolrestriction(inputs(agent)(i).first)} \\
& \wedge\ \text{st} = \text{cstore}_1\text{(agent)} \\
& \langle \text{st};\ r_1 = \text{r.receive(i)} \rangle\ (\text{st} = \text{st}_0 \wedge r_1 = r_2) \\
\rightarrow & \quad r_2 = \text{null} \\
& \wedge\ \text{st} \subseteq \text{st}_0
\end{array}
$$

Die Beweismethodik für beide Eigenschaften bleibt gleich wie in Kap. 8.7 beschrieben, lediglich die poolrestriction wird (analog zu validEncoding) eine zusätzliche Vorbedingung für die entsprechenden benutzten Theoreme über decode.

Auswirkungen auf die Verfeinerungsbeweise

Wir haben nun einen zusätzlichen Fehlerfall durch die poolrestriction eingeführt. Diese Fehlerquelle existiert lediglich in der konkreten Implementierung, die abstrakte Spezifikation eines Protokolls kann prinzipiell alle möglichen Eingaben verarbeiten. Damit eine Implementierung, die einen Pool verwalten muss, dennoch korrekt bezüglich einer abstrakten Spezifikation sein kann, muss diese bei zu großen Dokumenten (die im konkreten nicht dargestellt werden können) ebenso in einen Fehlerzustand übergehen. Konkret bedeutet dies, dass eine abstrakte PROSECCO Spezifikation bei Empfang eines Dokumentes, das nicht dem Pool genügt, das gleiche Zustandsübergangsverhalten zeigen muss, als hätte sie $\perp$ als Eingabe empfangen. Dies

begründet sich dadurch, dass $\perp$ die abstrakte Entsprechung des Rückgabewertes `null` einer Poolimplementierung ist, falls die Eingabe nicht der poolrestriction genügt. Auf der konkreten Implementierungsebene bemerkt man den Fehler hierbei sofort am Rückgabewert `null` von `receive` und kann sofort den entsprechenden Protokollschritt abbrechen und in einen Fehlerzustand übergehen. Auf der abstrakten Ebene muss man zunächst die Eingabe auf korrekte Struktur überprüfen und wird dann entsprechend einen Fehler bei der Eingabe erkennen, da die poolrestriction ja gerade die gültigen Eingaben des Protokolls charakterisieren soll.

Kryptographische Operationen

Die Transformationsschicht wird neben dem Empfangen von Nachrichten auch bei der Entschlüsselung verschlüsselter Nachrichten verwendet (Verschlüsselung und Entschlüsselung sind Operationen auf kodierten Byte Arrays, s. Kap. 10.8). Deshalb wird auch beim Entschlüsseln ein Pool benötigt. Dementsprechend müssen auch die Eigenschaften über `decrypt` mit der Vorbedingung poolrestriction versehen werden.

Um dies korrekt implementieren zu können, muss der Pool für Entschlüsselungsoperationen getrennt vom Kommunikationspool verwaltet werden. Schließlich könnte der Pool zum Zeitpunkt des Aufrufs von `decrypt` schon halb belegt sein, da die `receive` Operation bereits vorher dran war und den Speicher verbraucht hat. Damit wäre ein Theorem, das besagt, dass das Entschlüsseln immer ordentlich funktioniert, wenn die poolrestriction für das verschlüsselte Dokument gilt, nicht implementierbar.

Im einfachsten Fall ist der `decrypt` Pool eine genaue Kopie des Kommunikationspools und wird entsprechend bei Initialisierung der Anwendung erzeugt.

11.3.3 Beschränkte Größe der Ein- und Ausgabe

Ein weiterer Punkt, der bei der Übertragung des Verfeinerungskonzeptes auf Chipkarten zu beachten ist, ist die Tatsache, dass Chipkarten lediglich eine recht kurze Ein- und Ausgabenachrichtenlänge erlauben. Die Maximallänge des Datenteils einer Command APDU ist 255 Byte, die Maximallänge einer Antwort ist 256 Byte. Längere Eingaben können nicht gesendet werden und längere Ausgaben führen zu einem Laufzeitfehler auf der Chipkarte. Zusätzlich muss in der Command APDU bereits die erwartete Länge der Response APDU angegeben werden (Feld `LC`). Damit ergeben sich zwei Probleme: Das aktuelle Eingabedokument kann kodiert zu lang sein und das Ausgabedokument ebenso.

Zusätzliche Beweisverpflichtungen

Im einfachen Fall ist das Protokoll bereits so entworfen worden, dass diese Längenbeschränkungen bei wohlgeformten Eingabenachrichten immer eingehalten werden. Das in Kap. 13 vorgestellte Mondex Protokoll ist ein solcher Fall. Alle Eingaben sind kleiner als 256 Byte. Die möglichen Eingaben einer Protokollimplementierung wurden bereits mit der poolrestriction charakterisiert. Kann nun gezeigt werden, dass alle solchen Eingaben in kodierter Form kurz genug sind, gibt es mit der Eingabelängenbeschränkung keine Probleme. Wir müssen also zeigen:

THEOREM:

$$\mathsf{poolrestriction(doc)} \rightarrow \#\,\mathsf{encode(doc)} \leq 255$$

Analog dazu müssen die Ausgaben betrachtet werden. Diese sind bisher nicht in der poolrestriction enthalten, da davon ausgegangen wurden, dass die entsprechenden Pointerstrukturen innerhalb der Protokollimplementierung vorgehalten werden. Für die Längenbeschränkung wenden wir nun das gleiche Verfahren wie beim Eingabe-Pooling an und spezifizieren entsprechend einen Ausgabe-Pool mittels $\mathsf{poolrestriction}_{out}$. Analog zu oben muss dann gezeigt werden:

THEOREM:

$$\mathsf{poolrestriction}_{out}\mathsf{(doc)} \rightarrow \#\,\mathsf{encode(doc)} \leq 256$$

Gelten beide Beweisverpflichtungen müssen keine weiteren Adaptionen vorgenommen werden. Sollte die Eingabe oder Ausgabe allerdings länger sein, so müssen mehrere APDUs verwendet werden, um eine kodierte Eingabe an die Chipkarten zu senden oder eine Antwort zu erhalten.

Erweiterung: Lange Ein- und Ausgaben

Die Implementierung der `JavaCardAppletWrapper` Klasse kann entsprechend angepasst werden, wenn längere Nachrichten im Protokoll vorkommen. Die Vorgehensweise ist nun, die Nachrichten dann auf mehrere APDUs aufzuteilen. Ein entsprechendes Protokoll ist schematisch in Abb. 11.2 dargestellt.

Eine zu lange Eingabenachricht indoc wird in ein `byte[ ]` Array message kodiert. Die Länge dieses Arrays ist i * 255 + j (mit j < 255). Demnach sind i + 1 Command APDUs nötig, um dieses Array an die Karte zu senden. Auf der Karte (im `JavaCardAppletWrapper`) muss die message dabei vollständig zwischengespeichert werden. Sind alle Teile übertragen, signalisiert dies das Terminal durch Senden von DO_STEP. Dadurch wird wie bisher die `step()` Methode der eigentlichen Protokollimplementierung aufgerufen, die wie gehabt `receive` und `send` benutzt. Nach Abschluss des Protokollschrittes wird nun nicht mehr einfach die Ausgabenachricht zurückgesendet, sondern auch diese wird aufgeteilt. Unter der Annahme, dass das Ausgabedokument outdoc kodiert als `byte[ ]` Array returnval m * 256 + n Bytes lang ist, benötigen wir für das Senden der Antwort analog zu oben m + 1 Response APDUs. Da die Kommunikation mit Chipkarten immer vom Terminal aus angestoßen werden muss, wird

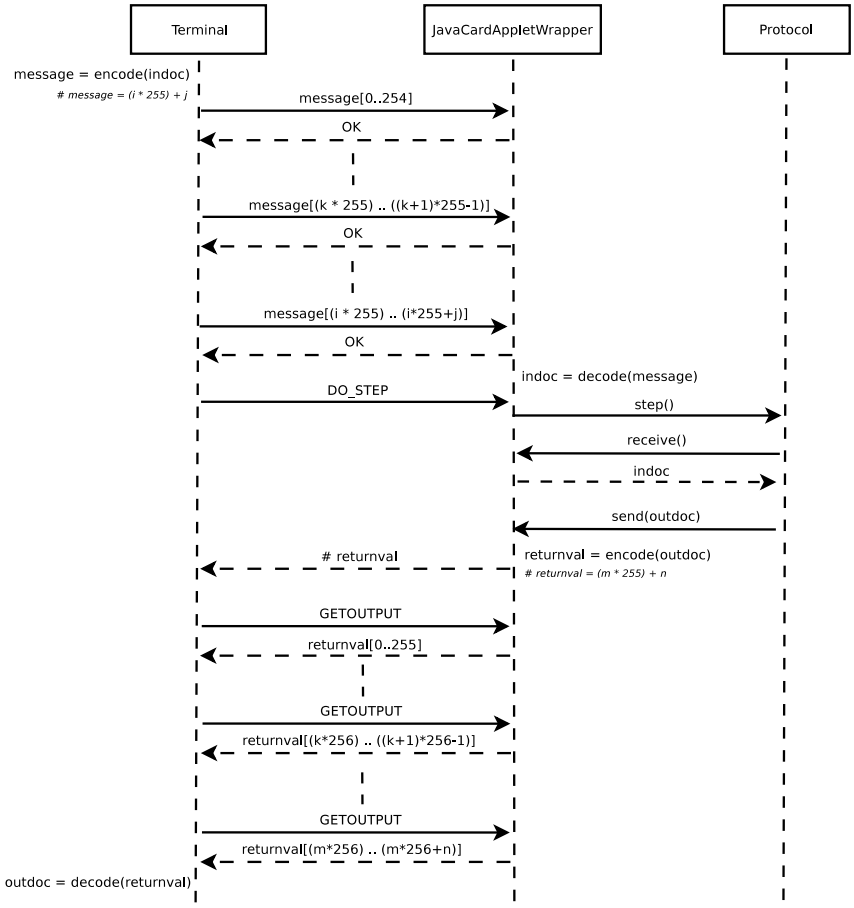


Abbildung 11.2: Protokoll für zu lange Ein- und Ausgabenachrichten

die erwartete Länge (# returnval) der Ausgabe zunächst an das Terminal übertragen. Dieses kann dann mittels des GETOUTPUT Kommandos die einzelnen Teile der Antwort empfangen und schließlich dekodieren.

11.3.4 Primitive Typen

Die letzte Einschränkung, die auf Chipkarten auftritt, ist das Fehlen des `integer` Typs in JavaCard. Zahlen können maximal als `signed short` mit 2 Byte (also im Wertebereich von -32768 bis 32767) dargestellt werden. Dies bedarf nun zweier Anpassungen:

Anpassung der Dokumente

Die Dokumente stellen einige Hilfsfunktionen bereit, die `int` Parameter bekommen. Ein Beispiel ist etwa die Funktion `get-part(int index)` im `Doclist` Typ, die aus einer Liste ein Dokument an gegebenem Index selektiert. Da auch die Maximallänge von Arrays in JavaCard auf die Länge eines `short` Wertes begrenzt ist, bekommt nun auch diese Methode einen `short`

statt des `int` Parameters. Auch Operationen z.B. zum Kopieren von Arrays (wo ein `int` als Schleifenzähler verwendet wurde) müssen entsprechend angepasst werden. Diese Änderungen sind allesamt unkritisch, da in wohlgeformten Eingabenachrichten in Kommunikationsprotokollen (und auf diese können wir uns dank der poolrestriction und obigen Argumentationen ja beschränken) keine Listen mit mehr als 32767 Einträgen vorkommen sollten - insbesondere nicht bei Chipkarten.

Anpassung der Transformationsschicht

Auch die Transformationsschicht basierte auf `int` Typen. Zum einen verwendete die Implementierung `int` Werte z.B. als Schleifenzähler. Hier können einfach wieder `short` Werte verwendet werden. Allerdings muss auch das Kodierungsformat angepasst werden. Das bisherige Format verwendete 4 Byte für die Angabe der Länge des Wertes eines kodierten Dokumentes (s. Kap. 10.1). Nun ist die Maximallänge durch die JavaCard Plattform auf 32767 Byte beschränkt. Damit reicht es auch aus, 2 Byte für die Angabe der Länge zu verwenden. Der Header einer Kodierung schrumpft damit insgesamt von 5 auf 3 Byte.

Kapitel 12

Ergebnisse im Detail: Mondex

Dieses Kapitel erläutert die Spezifikation und Verifikation der Mondex Chipkartenfallstudie. Dabei wird nach Vorstellung der Mondex Protokolle auch ein Einblick in die Beweisstrategie gegeben. Erfahrungen und Statistiken folgen zum Ende des Kapitels. Schließlich wird die Verifikation detaillierter mit einem anderen Ansatz zur Verifikation von Java Code für Mondex verglichen. Die in diesem Kapitel beschriebenen Arbeiten wurden in [161], [64], [160], [74] und insbesondere für die Java Verfeinerung in [67] publiziert.

12.1 Mondex: Funktionalität und Sicherheitseigenschaften

Die Mondex Anwendung ist eine Chipkartenanwendung, die eine elektronische Geldbörse realisiert. Es handelt sich hierbei nicht um eine reine Fallstudie, sondern um eine real existierende Anwendung der Firma Mastercard [89], die von vielen internationalen Forschergruppen als Fallstudie herangezogen wurde. Details dazu folgen unten in Kap. 12.3.1.

Die Funktionalität der Mondex Anwendung ist die Übertragung von elektronischem Geld von einer Smartcard auf eine andere. Die Smartcards werden dabei Purses genannt. Dabei sind die Karten als Ersatz für Bargeld anzusehen. Bei der Übertragung von Geld findet keine Verbindung zu einem externen Server statt. Die Karten müssen damit in einer beliebig feindlichen Umgebung sicher funktionieren.

Die abstrakteste Spezifikationsmöglichkeit für die Mondex Anwendung realisiert eine rein transaktionale Sichtweise. Als Abstract State Machine spezifiziert ergeben sich zwei wesentliche Zustandsfunktionen: die balance : name → nat und lost : name → nat. balance gibt zu jeder durch einen Namen name eindeutig bezeichneten Purse an, wie hoch ihr Kontostand derzeit ist. Die lost Komponente beschreibt, wieviel Geld eine Purse durch fehlgeschlagenen Übertragungen verloren hat. Zusätzlich gibt es noch eine Einschränkung auf den möglichen names: Das Prädikat authentic(name) gibt an, dass ein name eine echte Karte bezeichnet. Geldübertragung soll nur zwischen echten Karten möglich sein. In dieser abstrakten Sichtweise kann die Mondex Funktionalität wie folgt beschrieben werden:

SPEZIFIKATION:

```
MONDEX_A
   choose from, to, value, fail?
   with    authentic(from) ∧ authentic(to) ∧ from ≠ to
         ∧ value ≤ balance(from)
   in
      if ¬ fail? then
         balance(from) := balance(from) - value;
         balance(to) := balance(to) + value
      else
         balance(from) := balance(from) - value;
         lost(from) := lost(from) + value
```

Die Spezifikation wählt dabei zunächst zwei authentische Namen from und to sowie einen zu übertragenden Betrag value. Zusätzlich wird ebenso indeterministisch entschieden, ob die Übertragung des Geldes richtig funktionieren soll oder ob ein Fehler auftritt. Dies wird durch das Flag fail? ausgedrückt. Im Erfolgsfall werden die balances der Karten entsprechend verändert. Im Fehlerfall wird lediglich auf from Seite abgebucht, allerdings nicht auf anderer Seite gutgeschrieben. Statt dessen erhöht sich die lost Komponente.

Die wichtigste Sicherheitseigenschaft der Mondex Anwendung ist dabei, dass es nicht möglich sein darf, Geld durch das Fälschen von Nachrichten oder den Einsatz von gefälschten Chipkarten zu erzeugen. Ferner darf es nicht möglich sein, dass Geld auf echten Karten durch fehlerhafte Übertragungen verloren gehen kann.

Etwas formaler ausgedrückt bedeutet dies: Zum Einen darf die Summe der balance aller authentischen Purses nicht größer werden. Zum Anderen muss die Summe der balances und der losts aller Purses konstant bleiben. Auf dem vorgestellten abstrakten Level kann einfach bewiesen werden, dass diese Eigenschaften gelten. Die Gruppe des Autors zeigte dies z.B. in [160].

12.2 Das Mondex Protokoll

Die abstrakte Sichtweise auf Mondex kann nun durch ein Kommunikationsprotokoll implementiert werden. Die Anwendung ist so gestaltet, dass sämtliche sicherheitsrelevanten Funktionalitäten auf der Chipkarte implementiert sind. Die Karten müssen also in einem beliebigen Umfeld ohne besondere Annahmen z.B. an Kartenleser oder Nachrichtenübermittlung sicher funktionieren. Insbesondere muss der Ausfall von beliebigen Nachrichten oder das Fälschen von Nachrichten abgefangen werden.

Zur Übertragung kommunizieren zwei Karten über ein sogenanntes Terminal, das die ersten Kommunikationsschritte startet und danach lediglich Nachrichten zwischen den Karten weiterleitet. Zur Protokollierung von potentiell verloren gegangenem Geld speichern die Karten in einem Log fehlgeschlagene Transaktionen.

Das Protokoll der Mondex Anwendung ist in Abb. 12.1 dargestellt.

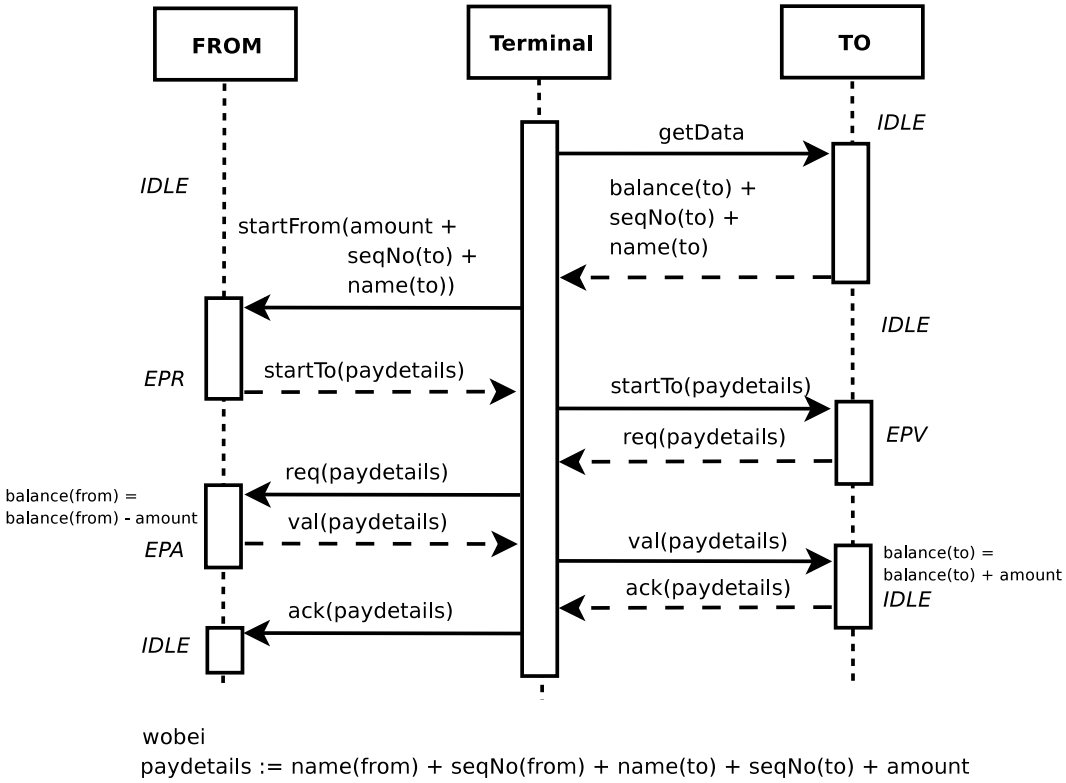


Abbildung 12.1: Protokoll der Mondex Anwendung

Die Chipkarten treten im Mondex Protokoll entweder als Empfänger (TO Purse) oder als Sender (FROM Purse) von Geld auf. Die Karten besitzen jeweils einen gespeicherten Geldbetrag (balance), einen eindeutigen Namen (name) und eine Sequenznummer (seqNo), die zum Vermeiden von Replay-Attacken verwendet wird. Die Sequenznummer wird dabei bei jedem Protokolllauf inkrementiert.

Das Protokoll zum Übertragen von Geld besteht dabei aus 5 verschiedenen Nachrichten sowie einer zusätzlichen Nachricht zur Abfrage des aktuellen Zustands der Karte. Letztere ist in Abb. 12.1 durch getData an TO dargestellt. Als Antwort wird eine Liste aus derzeitiger balance, seqNo und dem name der TO Purse zurückgesendet. Nun wird auf dem Terminal eingegeben, wieviel Geld von der FROM Purse auf die TO Purse übertragen werden soll. Daraufhin generiert das Terminal eine startFrom Nachricht, die den zu übertragenden Betrag (amount) und die seqNo sowie den name der TO Purse enthält. Ab jetzt wird immer ein Datentyp namens paydetails als Nachrichteninhalt verwendet. paydetails sind ein Quintupel aus den Namen und den Sequenznummern beider Karten sowie dem Betrag, der aktuell übertragen werden soll. Damit kennzeichnen die paydetails eine Übertragung von Geld zwischen zwei Karten eindeutig. Die paydetails werden nun in 4 Nachrichten zwischen den Karten hin und her geschickt. Das Terminal leitet die Nachrichten jeweils nur weiter. Die erste Nachricht startTo dient zunächst dazu, auch der TO Purse die aktuellen Transaktionsdaten mitzuteilen. Die zweite Nachricht req führt nun dazu, dass auf der FROM Purse das Geld abgebucht wird (balance(from) = ba-

lance(from) - amount). Die Antwort val schreibt das Geld auf der TO Purse gut (balance(to) = balance(to) + amount). Schließlich beendet ack die Transaktion. Dabei werden die paydetails auf jeder Karte für die Dauer der Transaktion gespeichert und beim Empfang jeder einzelnen Nachricht wird überprüft, ob die übertragenen paydetails und die intern gespeicherten paydetails übereinstimmen (ob also die empfangene Nachricht korrekt zur aktuellen Transaktion passt).

Jede Chipkarte besitzt dazu auch einen internen Zustand, der den aktuellen Fortschritt im Protokolllauf anzeigt. Dieser Zustand kann entweder IDLE, EPR (expecting request), EPV (expecting value) oder EPA (expecting acknowledge) sein. Er gibt jeweils an, welche Nachricht die nächste ist, die von einer Purse erwartet wird, falls ein Protokoll läuft. Falls im Moment kein Protokoll läuft, ist der Zustand IDLE.

Durch die internen Zustände kann eine Purse erkennen, ob die Transaktion im Moment in einem kritischen Zustand ist. Kritisch sind dabei all die Konstellationen, in denen ein Verlust oder eine Fälschung der nächsten Nachricht zu Geldverlust führen könnte. Kommt etwa eine req Nachricht bei einer FROM Purse an, die darauf folgende val Nachricht allerdings nicht mehr, so wurde auf der FROM Purse Geld abgebucht, allerdings nirgends Geld gutgeschrieben. Um dies zu protokollieren, besitzt eine Purse ein Log, in dem die paydetails aller fehlgeschlagenen Transaktionen gespeichert werden. Eine Transaktion wird dabei als fehlgeschlagen (in dem Sinne dass Geld tatsächlich verloren gegangen ist) angesehen, wenn es zwei Purses gibt, die einen identischen Logeintrag haben. In dem eben beschriebenen Fall kann die FROM Purse nämlich nicht entscheiden, ob tatsächlich ein Übertragungsfehler bei der val Nachricht stattgefunden hat oder ob die ack Nachricht verloren gegangen ist. Schließlich ist in beiden Fällen seine lokale Sicht die gleiche: die ack Nachricht kommt nicht an. Die FROM Purse muss also in beiden Fällen einen Logeintrag erzeugen. Falls nur die ack Nachricht verloren gegangen ist, ist allerdings kein Geld verloren gegangen (die TO Purse hat schließlich val erhalten und Geld gut geschrieben). In diesem Fall hat die TO Purse allerdings auch keinen Logeintrag erzeugt. Falls aber val nicht angekommen oder verändert sein sollte, so geht Geld verloren (spätestens beim Starten der nächsten Transaktion) und es wird auch auf der TO Purse ein (zur FROM Purse identischer) Logeintrag erzeugt. Kritisch ist also in diesem Sinne der Zustand EPA der FROM Purse und der Zustand EPV der TO Purse. In anderen Zuständen muss kein Logeintrag bei fehlerhaften Nachrichten erzeugt werden.

Gibt es zwei Purses mit identischen Logeinträgen, so kann dies mit einem eigenen (hier nicht betrachteten) Protokoll zur Abfrage der Logs festgestellt werden und durch die Bank das Geld zurückerstattet werden.

Die Sicherheit der Mondex Anwendung wird dadurch sichergestellt, dass die Nachrichten STARTTO, REQ, VAL und ACK unfälschbar gesichert werden. Dabei wird in der Fallstudie zunächst nicht festgelegt, wie genau dies erreicht wird (etwa durch Verschlüsselung, Signatur oder Ähnlichem).

Die oben genannten beiden Sicherheitseigenschaften kann man sich auf dem Protokolllevel so vorstellen: die Summe der Kontostände auf den Karten (balance) darf über die Zeit nicht größer werden. Ebenso muss die Summe auf balance sowie des Geldes das sich derzeit in Übertragung befindet (inTransit) und des Geldes, das durch fehlerhafte Übertragungen verloren gegangen ist (lost) konstant sein. Auf der Ebene des Protokolls wird die lost Komponente dabei durch auf zwei Karten paarweise vorhandene identische Log Einträge repräsentiert.

12.3 Vorarbeiten zu Mondex

Dieser Abschnitt bettet die Arbeiten des Autors zu Mondex in den Kontext internationaler Vorarbeiten und von Vorarbeiten der Arbeitsgruppe des Autors ein. Insbesondere sei hier darauf verwiesen, dass die Mondex Fallstudie erst ab der Abstraktionsebene der Prosecco Spezifikation vom Autor behandelt wurde. Dennoch wird hier auch eine abstraktere Spezifikation vorgestellt. Dies geschieht zum Einen um die Arbeiten in den Kontext internationaler Arbeiten zu Mondex einzubetten und zum Anderen, da auf den abstrakteren Ebenen ein Angriff auf das Protokoll gefunden werden konnte, der zu einer Änderung des Protokolls auch auf der Prosecco Ebene führte. In diesem Abschnitt werden nun zunächst internationale Vorarbeiten behandelt und danach Vorarbeiten mittels KIV sowie der genannte Angriff vorgestellt. Die Prosecco Spezifikation folgt dann in Kap. 12.4. Für das Verständnis der späteren Kapitel sind die in diesem Abschnitt folgenden Ausführungen nicht unbedingt nötig.

12.3.1 Internationale Vorarbeiten zu Mondex

Mondex wurde als eine der ersten Anwendungen nach den Erfordernissen der Information Technology Security Evaluation Criteria (ITSEC) [91] Level E6 evaluiert [39]. Diese Evaluation schließt dabei auch formale Methoden mit ein. Die entsprechende Verifikation wurde im Jahr 2000 von Stepney, Cooper und Woodcock [174] mittels der Spezifikationssprache Z [171] durchgeführt. Alle Beweise wurden zu diesem Zeitpunkt händisch auf Papier durchgeführt. Diese Arbeit war es auch, die die bereits oben beschriebenen zwei Spezifikationsebenen (und aus beweistechnischen Gründen noch eine dritte Zwischenebene) einführte. In [174] wird gezeigt, dass die Protokollebene eine korrekte Verfeinerung der abstrakten Spezifikation ist und damit die Sicherheitseigenschaften auch im Protokoll erfüllt werden.

Die Anwendung bekam 2006 als Fallstudie für die toolgestützte Verifikation internationale Bedeutung, da sie im Rahmen der von Hoare initiierten Grand Challenge for Computing [82] [83] [81] als Challenge für Verifikationswerkzeuge ausgerufen wurde [99] [187]. Seit dem haben sich viele internationale Arbeitsgruppen mit der Fallstudie beschäftigt und gezeigt, dass ihre Verifikationswerkzeuge die toolgestützte Verifikation der Originalfallstudie effizient ermöglichen. Die Ergebnisse dieser Bemühungen wurden dem Verified Software Repository [24] [151] hinzugefügt, das sich zum Ziel gesetzt hat, verifizierte Fallstudien öffentlich zugänglich zur Verfügung zu stellen. Den im Folgenden in diesem Zusammenhang genannten Ansätzen ist gemein, dass sie versuchen, die Fallstudie möglichst originalgetreu nach [174] in ihrem jeweiligen Formalismus auszudrücken und zu verifizieren. Zu diesen Arbeiten gehören:

- Freitas und Woodcock verifizierten in [188] die Mondex Fallstudie mit dem Tool Z/Eves [188]. Die Formalisierung entsprach dabei exakt der Formalisierung der Original Fallstudie. Durch den jetzigen Einsatz eines rechnergestützten Verifikationssystems wurden dabei einige kleinere Fehler in der Formalisierung entdeckt und beseitigt. Der Gesamtaufwand für die beschriebene Verifikation (in [188] werden 90% der Fallstudie als verifiziert angegeben) beträgt hier ca. acht Wochen.

- Butler und Yadav zeigen in [35] eine Verifikation der Mondex Fallstudie mittels Event-B [4] und den Tools B4Free [7] und Click'n'Prove [3]. Ihre Herangehensweise ist dabei, eine Spezifikation der abstraktesten und der konkretesten der drei Ebenen der Original Z Ve-

rifikation in Event-B zu formulieren. Anschließend zeigen sie, dass durch die Definition von acht weiteren Zwischenebenen, in denen jeweils kleine Details geändert werden, eine durchgängige Kette von korrekten Verfeinerungen angegeben werden kann, die dann insgesamt mit einem sehr hohen Automatisierungsgrad von 97% mittels Forward Simulation bewiesen werden können. Auf der abstraktesten Ebene existieren lediglich atomare Übertragungen von Geld zwischen zwei Purses oder das Fehlschlagen einer solchen Übertragung. In weiteren Schritten wird dann zu mehreren Protokollschritten mit weiterhin globalem Zustand, mehrere Protokollschritten mit lokalen Zustand und schließlich zu echten Protokollnachrichten und der Möglichkeit von Replay-Attacken verfeinert. Trotz des hohen Automatisierungsgrades der eigentlichen Beweise bleibt dabei anzumerken, dass der gesamte Arbeitsaufwand dennoch auf Grund der hohen Anzahl an Ebenen und der Notwendigkeit der jeweiligen Definition von passenden Simulationsrelationen nicht kleiner ist als bei dem Einsatz anderer Tools.

- Haxthausen, George und Schütz geben in [78] eine Spezifikation für Mondex mittels der RAISE Specification Language (RSL) [58] [132] an. Die Beweise werden durch Übersetzung in Eingaben für die Tools PVS [138] und SAL [42] geführt. Ebenso wie die Z Spezifikation werden hier drei Spezifikationsebenen verwendet und die Sicherheitseigenschaften bereits auf der abstraktesten Ebene gezeigt. Da allerdings in RAISE Refinement im Wesentlichen Modellinklusion bedeutet, arbeiten alle drei Modelle auf dem gleichen Datentyp. Auf dem abstraktesten Level werden daher bereits verschiedene Protokollschritte eingeführt (Abbuchen auf der einen Karte, Gutschreiben auf der anderen Karte, Verlust von Geld, ...). Diese werden allerdings nur deklarativ beschrieben. In den konkreteren Ebenen werden dann tatsächliche interne Zustände und Nachrichten hinzugefügt. Insgesamt wurden mit PVS 319 Beweise geführt, von denen 88 von den Autoren als komplex und nicht automatisch beweisbar eingestuft wurden. Darüber hinaus wurde eine Übersetzung der RSL Spezifikationen in die Eingabe für den symbolischen SAL Model Checker SAL-SMC durchgeführt. Hierfür musste das Modell natürlich endlich gehalten werden. Durch die Tatsache, dass die Modelle auch interaktiv verifiziert wurden, konnten die Autoren durch Model Checking keine weiteren Fehler finden. Dennoch zeigte sich, dass dass der Model Checker in der Lage ist, Fehler durch kleine absichtliche Änderungen am Modell aufzuspüren. Auch die oben genannten Sicherheitseigenschaften sowie Liveness Aussagen (z.B. Abbuchen von Geld ist möglich) konnten so automatisch gezeigt werden.

- Banach, Jeske, Poppelton und Stepney spezifizieren in [13] die Anwendung ebenso in der Z Spezifikationssprache, verifizieren allerdings mittels Retrenchment. Retrenchment ist dabei eine abgeschwächte Form von Refinement. Es ist dabei nicht zwingend nötig, für jeden Schritt auf der konkreteren Ebene einen im Sinne einer Simulationsrelation passenden abstrakten Schritt zu finden. Statt dessen erlaubt die Definition der Retrenchment Beweisverpflichtungen in manchen Fällen die Verletzung der Simulationsrelation im Nachfolgezustand. Dann muss allerdings eine sog. concede Relation gelten, die die Verletzung der Simulationsrelation explizit beschreibt. Als kanonisches Beispiel wird z.B. die Implementierung von arithmetischen Operationen auf natürlichen Zahlen durch beschränkte Integers angegeben, bei denen ein Überlauf stattfinden kann. Eine solche Implementierung ist dann ein Refinement, falls kein Überlauf statt findet. Wenn Überläufe statt finden, handelt es sich nur noch um ein Retrenchment. Details dazu

finden sich z.B. in [14]. Für die Mondex Anwendung finden die Autoren vier Anwendungen für Retrenchment: Die Sequenznummern sind in einer Implementierung endlich und können dementsprechend überlaufen. Die Logs auf den Karten sind ebenso endlich und können voll werden. Die verwendeten kryptographischen Operationen (in diesem Falle eine Hash Operation beim von uns nicht betrachteten Löschen von Log Einträgen) ist real nicht injektiv. Zu Letzt gehen die Autoren noch auf eine Problematik bezüglich der Abfrage von Kontoständen auf den Karten während eines Protokolllaufs ein. Alle angesprochenden Punkte werden von den Autoren über die alternative Retrenchment concede Relation beschrieben, hier wird also jeweils die Verfeinerungskorrektheit abgeschwächt. Sämtliche Beweise wurden per Hand geführt. Vergleichend zu den von der Gruppe des Autors geführten Beweisen bleibt zu sagen: In unseren Spezifikationen und in der Implementierung sind natürlich Logs und Sequenznummer ebenfalls endliche Datentypen. Wir lösen die Problematik dadurch auf, dass die Karten bei vollen Logs oder maximalen Sequenznummern spezielle Fehlererkennungsschritte durchführen, die Refinements im Sinne von 0:1 Diagrammen (also Verfeinerungen von skip) sind. Die Injektivität von Kryptoalgorithmen wurde bereits in Kap. 9 beschrieben.

- Ramananadro und Jackson [145] benutzen den Alloy Model Finder [92], um die Verifikation von Mondex durchzuführen. Dieser Ansatz unterscheidet sich am meisten von allen anderen hier vorgestellten Ansätzen, da Alloy zum Einen auf einer rein relationalen Eingabesprache beruht und zum Anderen das Alloy Tool (im Hintergrund mittels SAT Checking realisiert) eine rein vollautomatische Gegenbeispielsuche für die Modelle implementiert. Durch diese Herangehensweise ist es natürlich nötig, unbeschränkte Datentypen aus den Modellen zu entfernen. So werden die Kontostände auf den Karten nicht z.B. mittels natürlicher Zahlen modelliert, sondern als endliche Mengen von coins ohne konkrete Werte. Sequenznummern werden ebenso nicht als Zahlen modelliert, sondern als abstrakte Werte, auf denen lediglich eine Ordnung definiert ist. Ebenso muss natürlich die Anzahl der Purses endlich sein. Mit diesen Modellierungseigenschaften gelingt es den Autoren, die Korrektheit der Mondex Verfeinerung mit den auch im Original verwendeten drei Ebenen automatisch zu verifizieren. Dabei muss natürlich bedacht werden, dass mit Alloy keine formalen Beweise im Sinne der anderen hier vorgestellten Methodiken möglich sind, sondern nur Modellprüfungen mit einer jeweils begrenzten Anzahl von Elementen aller verwendeten Sorten. Dennoch zeigt die Arbeit, auch bedingt durch die Tatsache, dass Alloy die gleichen Fehler in der Original Spezifikation finden konnte wie auch andere Ansätze, dass das Alloy Tool durchaus in der Lage ist, vollautomatisch sehr gute und verlässliche Aussagen über die Modelle zu liefern.

- Crocker modelliert die Fallstudie mittels dem Tool PerfectDeveloper [37]. Ergebnisse sind im Verified Software Repository einsehbar [151]. Das Tool ist dabei für den Einsatz in realen Softwareentwicklungsprojekten gedacht und sieht daher keine Möglichkeit zur interaktiven Verifikation vor. Die Spezifikationssprache ist objektorientiert. Der Autor spezifizierte zunächst die abstrakteste transaktionale Ebene der Mondex Fallstudie und konnte dort die geforderten Sicherheitseigenschaften beweisen. Eine Verfeinerung zu einem konkreten Modell war allerdings nicht möglich. Statt dessen versucht der Autor, die Sicherheitseigenschaften auf einem konkreteren Modell der Anwendung, das der konkreten Protokollebene mit Nachrichten von Mondex entspricht, die Sicherheitseigenschaften erneut zu zeigen. Nach einer Modellverbesserung für die Automatisierbarkeit

gelingt der automatische Beweis von 229 von 244 generierten Beweisverpflichtungen auf dieser Ebene, eine vollständige Verifikation ist allerdings nicht möglich.

Darüber hinaus haben auch andere Gruppen die Mondex Fallstudie aufgegriffen und als Beispiel herangezogen. So zeigen Tonin und Schmitt in [165] und [182] eine Verifikation einer Implementierung der Mondex Fallstudie in JavaCard mittels dem KeY Beweissystem [19]. Ein ausführlicherer Vergleich zu diesem Ansatz findet sich in Kap. 12.8.

Kong, Ogata und Futatsugi verifizieren in [109] Mondex mittels der OTS/CafeOBJ Methodik [135]. Dabei wird ein Transitionssystem auf algebraischen Datentypen durch Angabe von Zustandsübergangsgleichungen definiert. Der Ansatz ähnelt dabei im Grundsatz der operationalen Spezifikationsmethodik mit ASMs, wie sie auch von der Gruppe des Autors verwendet wurde, s. auch unten Kap. 12.3.2. Die Autoren verifizieren die Sicherheitseigenschaften der Mondex Anwendung anschließend direkt auf der konkreten Ebene, es gibt kein Refinement und keine Abstraktion. Problematisch an der Spezifikation in CafeOBJ ist dabei, dass die STARTFROM und STARTTO Nachrichten initial gleichzeitig und sofort mit den korrekten Sequenznummern gesendet werden. Das macht Angriffe, die auf dem Senden von falschen zukünftigen Sequenznummern beruhen, im Modell nicht möglich. Die Z Spezifikation von Mondex sieht daher z.B. explizit vor, dass alle möglichen Sequenznummern an die Börsen geschickt werden können.

Kuhlmann und Gogolla zeigen schließlich in [111] eine Modellierung der Mondex Anwendung mittels UML und OCL. Dabei werden Klassendiagramme zur Modellierung verwendet. In dieser Modellierung beschränken sich die Autoren auf die abstrakteste Ebene der Original Fallstudie, keine Protokolle und keine Verfeinerung werden betrachtet. Zur Validierung ihrer Modelle verwenden die Autoren das OCL Tool USE [60], das Testfälle für OCL Constraints erzeugen kann und so hilft, Fehler zu finden. Das Ziel dieser Arbeit liegt nicht in der Verifikation der Sicherheitseigenschaften, sondern in der einfachen Verständlichkeit der Modellierung der Zusammenhänge der Anwendung.

12.3.2 Eigene Vorarbeiten zu Mondex

Auch die Arbeitsgruppe des Autors hat die originale Fallstudie mit dem KIV System und unter Benutzung von Abstract State Machines verifiziert. [161] zeigt dabei unsere Ergebnisse unter Benutzung der auch im Original verwendeten Backward-Refinement Theorie. Eine kurze Übersicht über die abstrakten KIV Spezifikationen findet sich unten in Kap. 12.3.3. Es gelang mit dem KIV System, die Korrektheit der Verfeinerung mittels lediglich zweier Ebenen zu zeigen (und nicht drei wie in der Original Verifikation).

Wir konnten dabei auch zwei Fehler in der Original Verifikation entdecken. Die Kopplungsinvariante ist zu schwach und muss um eine zusätzliche Bedingung erweitert werden: Es wurde übersehen, dass die Information benötigt wird, dass in den lokalen paydetails einer TO Purse im Zustand EPV als Empfänger immer nur der Name eben dieser Purse stehen kann (andere paydetails akzeptiert die Purse nicht). Diese Bedingung wird benötigt, um die Verfeinerungskorrektheit zeigen zu können. Ebenso benötigt man eine ähnliche Bedingung für eine FROM Purse im Zustand EPA, auch hier muss der Absender in den paydetails dem Namen der Purse entsprechen.

Neben dieser Verifikation haben wir die Fallstudie auch mittels der für ASMs natürlichen Ver-

feinerungstheorie ASM Refinement [26], [156], [25], [158] mittels Forward-Simulation bewiesen [160]. Diese Verifikation zeigte neben der systematischeren Herangehensweise darüber hinaus auch eine Anfälligkeit des originalen Mondex Protokolls für eine Denial-of-Service Attacke: Im Gegensatz zu dem in Abb. 12.1 gezeigten Protokoll sieht das Original Protokoll aus [174] das Generieren und Senden der STARTTO Nachricht durch das Terminal vor. Dort beinhaltet die STARTTO Nachricht auch noch nicht die gesamten paydetails sondern lediglich analog zu STARTFROM amount, name der FROM Purse sowie deren seqNo. In Abb. 12.1 wird diese Nachricht dagegen von der FROM Purse erstellt und gesendet. Würde STARTTO wie im Original von Terminal gesendet werden, müsste dies ungesichert erfolgen, da das Terminal in der Mondex Anwendung nicht über kryptographische Mechanismen verfügt. Alle Sicherheitsziele müssen alleine durch die Karten gesichert werden. Durch das alte Protokoll ergäbe sich folgende Möglichkeit für einen Angreifer:

Der komplette Angriff ist im Sequenzdiagramm in Abb. 12.2 nochmals beschrieben. Wir beschreiben im Folgenden zuerst eine abgeschwächte Variante und dann den vollen Angriff.

Zunächst benutzt der Angreifer eine gefälschte Karte und startet mit einer echten Karte a und einem normalen Terminal einen Protokolllauf. Daraufhin wird mittels getData Namen und Kontostand sowie die als nächstes zu benutzende Sequenznummer der Purse a abgefragt und an die Purse des Angreifers über STARTTO übermittelt. Diese Informationen sind öffentlich - und müssen dies auch sein, damit ein echtes Terminal einen Protokolllauf starten kann. Nun kann die Karte des Angreifers (völlig ohne Involvierung von Purse a) auf getData mit den Daten von a antworten. Also wird in einem zweiten Protokolllauf (in dem sich der Angreifer als a ausgibt) zu einer Karte b eine STARTTO Nachricht vom Terminal an eine Purse b gesendet werden, in der Namen von a sowie eine Sequenznummer übermittelt werden. Darauf antwortet die Purse b mit einer REQ Nachricht, geht in Zustand EPV, schreibt allerdings noch kein Geld gut. Bricht der Angreifer nun die Kommunikation ab, so muss die Purse b einen Logeintrag erzeugen, da für diese lokal ja nicht bekannt ist, ob auf der (hier nicht existenten) FROM Partnerkarte bereits Geld abgebucht wurde oder nicht (es könnte ja genauso gut die VAL Nachricht verloren gegangen sein). Damit ist der Angreifer in der Lage, auf einer echten Karte nach Belieben Log-Einträge zu erzeugen. Dies ist nicht wünschenswert, da die Länge der Logs auf echten Karten aufgrund der beschränkten Ressourcen auf Chipkarten eng begrenzt sein muss. Sind die Logeinträge belegt, kann die Karte nicht mehr benutzt werden und man muss die Bank aufsuchen. Damit könnte ein Angreifer so ehrlichen Benutzern schaden. Immerhin geht bisher bei diesem Angriff noch kein Geld verloren.

Allerdings lässt sich der Angriff weiter führen: Die Sequenznummern werden bei Mondex bei Erhalt einer neuen STARTO bzw. STARTFROM Nachricht inkrementiert, damit die nächste Transaktion mit einer neuen Nummer statt findet. Problematischerweise kann nicht davon ausgegangen werden, dass ein Angreifer diesen Vorgang des Inkrementierens nicht nachvollziehen kann (im einfachsten Fall würde die Sequenznummer einfach um eins erhöht werden). Damit ist für den Angreifer aus dem Wissen über die derzeitige Sequenznummer einer Karte auch die nächste Sequenznummer herleitbar. Verwendet er nun in der Transaktion mit Purse b eine solche inkrementierte Sequenznummer, ist die Antwort von Purse b für die nächste Transaktion mit Purse a passend und der Angreifer kann sich so fälschlicherweise mit seiner gefälschten Karte wie Purse b verhalten. Ebenso kann er dafür sorgen, dass das Terminal passende STARTFROM Nachrichten an die Purse a erzeugt. Diese antwortet darauf nicht sofort, geht aber in Zustand EPR. Problematisch ist nun, dass die Purse a nun auch auf genau

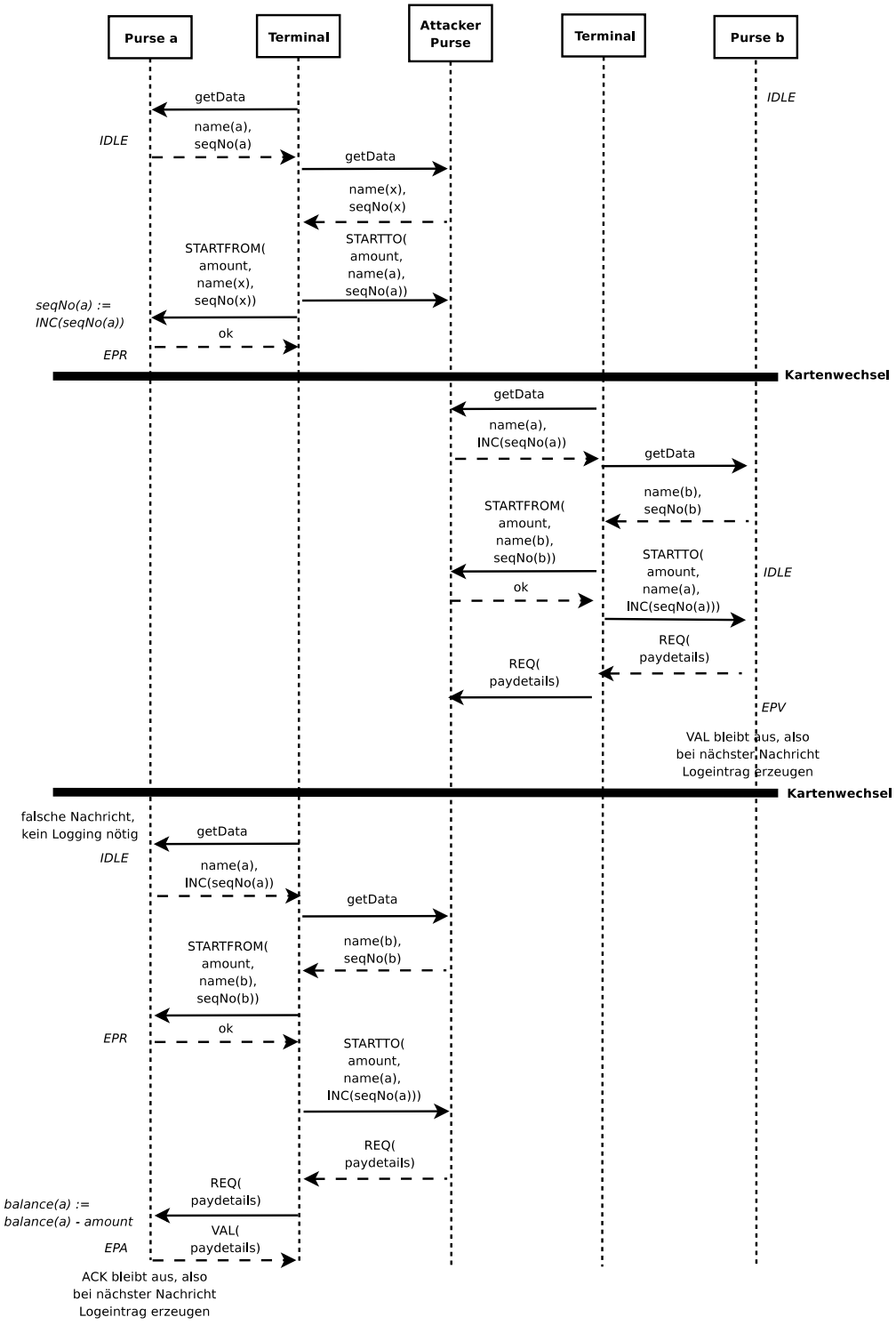


Abbildung 12.2: Angriff auf das originale Mondex Protokoll

die (aus dem vorher beschriebenen Angriff bekannten) REQ Nachricht von Purse b mit dem Abbuchen von Geld reagiert (gesetzt dem Fall dass der Angreifer vorher bereits die inkrementierte Sequenznummer verwendet hat). Damit kann ein Angreifer auf einer beliebigen Purse das Abbuchen von Geld initiieren. Zwar wurde auch auf einer anderen Purse ein Logeintrag erzeugt, womit die Bank das Geld wieder herstellen könnte, allerdings müssen sich beim eben beschriebenen Vorgehen die Besitzer von Purse a und Purse b überhaupt nicht kennen. Daher wird dieses Wiederherstellen des Geldes zumindest sehr schwierig.

Als einfache Lösungsmöglichkeit bietet sich an, die Nachricht STARTTO als Antwort auf STARTFROM von der FROM Purse kryptographisch gesichert und unfälschbar zu versenden. Damit kann der Angreifer die eben beschriebenen Schritte nicht mehr durchführen. Genau diese Änderung wurde in unserem hier verwendeten Mondex Protokoll implementiert und wurde auch in Abb. 12.1 so dargestellt.

Bei dem Angriff in Abb. 12.2 kann noch kritisiert werden, dass es unrealistisch ist, dass die Sequenznummer einfach inkrementiert wird. Dem kann man allerdings entgegenhalten, dass ein Angreifer einfach die Kommunikation mit Purse b beliebig oft hintereinander mit jeweils verschiedenen Sequenznummern durchführen kann und dann bei der zweiten Kommunikation mit Purse a alle so erhaltenen REQ Nachrichten durchprobieren kann, bis er die richtige gefunden hat.

12.3.3 Spezifikationshierarchie der Mondex Anwendung in KIV

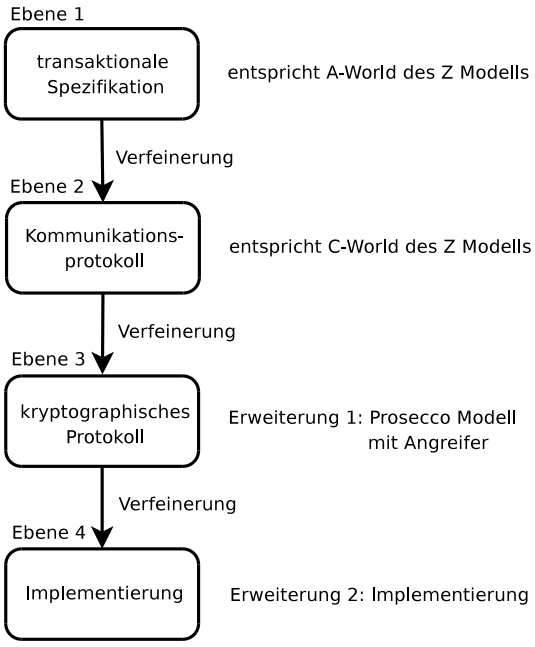


Abbildung 12.3: KIV Spezifikationsebenen bei Mondex

Die abstrakteste Art und Weise, Mondex zu spezifizieren, wurde bereits zu Beginn dieses Kapitels angegeben. Ziel der Verifikations-Challenge war es nun, die Protokollebene von Mondex ebenso zu spezifizieren und die Verfeinerungskorrektheit zu zeigen. Diese Challenge wurde von

der Gruppe des Autors noch erweitert, in dem zusätzlich zu den beiden beschriebenen Ebenen noch eine zusätzliche PROSECCO Spezifikation sowie eine Implementierung samt Spezifikation nach den in dieser Arbeit beschriebenen Methodiken angegeben wurden. Zusammen ergibt sich so eine Spezifikationshierarchie mit vier Ebenen. Die erste Ebene ist die transaktionale Sichtweise mit dem Beweis der Sicherheitseigenschaften. Die zweite Ebene ist ein Protokollmodell mittels ASMs, das allerdings noch keine Kryptographie und keinen expliziten Angreifer vorsieht. Auf Ebene drei wird ein PROSECCO Modell definiert und schließlich auf Ebene 4 eine Implementierungsebene angegeben. Eine Übersicht findet sich in Abb. 12.3.

12.3.4 Abstrakte Spezifikation auf Ebene 2

Ebene 2 benutzt dabei in unserer Spezifikation mittels ASMs die gleichen Modellierungsprinzipien wie die C-World Spezifikation aus [174]. Insbesondere wird keine Kryptographie spezifiziert, es existiert kein eigener Angreifer, kein Terminal, kein menschlicher Benutzer und auch kein explizites Kommunikationsmodell mit Kanälen im Modell. Statt dessen kommunizieren die Purses über einen sogenannten ether. Im ether befinden sich dabei alle die Nachrichten, die von einer Purse in einem Protokolllauf abgeschickt wurden. Das Empfangen einer Nachricht wird modelliert durch indeterministisches Auswählen einer dieser Nachrichten. Initial befinden sich alle möglichen STARTFROM Nachrichten im ether, was das Starten eines beliebigen Protokolllaufs im Modell ermöglicht. Die Reihenfolge, in der die Purses im Modell einen Protokollschritt ausführen ist dabei ebenfalls indeterministisch. Replayattacken werden im Modell so möglich, da Nachrichten immer im ether verbleiben.

Für Ebene 2 beschreiben wir in dieser Arbeit lediglich die grundlegende Spezifikationsstruktur. Genaue Details zu Spezifikation und insbesondere zum Beweis der Verfeinerungskorrektheit zu Ebene 1 finden sich z.B. in [160].

Die PayDetails werden auf Ebene 2 als algebraischer Datentyp spezifiziert:

SPEZIFIKATION:

```
PayDetails = mkpd(.from: name; .fromno: nat; .to: name; .tono: nat; .value: nat)
```

Ebenso als Datentyp (man beachte insbesondere als Unterschied zu PROSECCO ohne jegliche Kryptographie) werden die Nachrichten der Anwendung spezifiziert:

SPEZIFIKATION:

```
message =  startFrom(.to: name; .tono: nat; .value: nat)
         | startTo(.pd: PayDetails)
         | req(.pd: PayDetails)
         | val(.pd: PayDetails)
         | ack(.pd: PayDetails)
```

Als Zustandsfunktionen auf Ebene 2 werden folgende Funktionen verwendet:

SPEZIFIKATION:

```
balance : name → nat
state: name → status
pdAuth: name → PayDetails
exLog: name → set(PayDetails)
nextSeqNo : name → nat
ether : set(message)
```

Der status ist dabei IDLE, EPR, EPA oder EPV. Die Funktion pdAuth stellt die aktuellen Transaktionsdetails für die einzelnen Purses dar. Das exLog setzt das oben beschriebene Log für fehlgeschlagene Transaktionen als Menge von PayDetails um. Die nextSeqNo gibt die in der nächsten Transaktion zu verwendende Sequenznummer an.

Die Hauptregel der Mondex ASM auf Ebene 2 ist CSTEP. Diese Regel wählt wie beschrieben eine Nachricht msg aus dem ether, sowie einen receiver für diese. Ebenso wird (analog zu Ebene 1) indeterministisch festgelegt, ob das Empfangen der Nachricht klappt oder nicht. Letzteres geschieht durch das fail? Flag. Im Anschluss wird eine Regel LCSTEP ausgeführt, die abhängig von der Art der Eingabenachricht eine Regel spezifisch für den jeweiligen Protokollschritt auswählt und die Eingabe auf Korrektheit überprüft. So gibt z.B. das Prädikat isOKstartFrom(msg) an, das der state des receivers IDLE ist und der in der Eingabenachricht angegebene msg.value für die Höhe des zu übertragenden Geldbetrags kleiner ist als der aktuelle Kontostand von receiver. Zusammen ergibt sich:

SPEZIFIKATION:

```
CSTEP#
choose msg, receiver fail? with  msg ∈ ether ∧ authentic(receiver)
     in LCSTEP#

LCSTEP#
   if isOKstartFrom(msg) ∧ ¬ fail? then STARTFROM#
   else if isOKstartTo(msg) ∧ ¬ fail? then STARTTO#
   else if msg = req(pdAuth(receiver)) ∧ state(receiver) = epr ∧ ¬ fail?
     then REQ#
   else if msg = val(pdAuth(receiver)) ∧ state(receiver) = epv ∧ ¬ fail?
     then VAL#
   else if msg = ack(pdAuth(receiver)) ∧ state(receiver) = epa ∧ ¬ fail?
     then ACK#
   else ABORT#
```

Die einzelnen Protokollschritte müssen nun die Zustandsfunktionen entsprechend verändern und eine Antwort in den ether senden. Als Beispiel betrachten wir STARTFROM. Dieser Protokollschritt muss die Sequenznummer inkrementieren (um einen beliebigen Betrag), die aktuellen Transaktionsdetails pdAuth korrekt auf die übertragenen Werte setzen, den Zustand auf EPR setzen sowie eine STARTTO Ausgabenachricht erzeugen:

SPEZIFIKATION:

```
STARTFROM#
   choose n with nextSeqNo(receiver) < n in
      pdAuth(receiver) := mkpd(receiver,nextSeqNo(receiver),
                               msg.name, msg.nextSeqNo, msg.value),
      state(receiver) := EPR,
      nextSeqNo(receiver) := n;
      let outmsg = startTo(pdAuth(receiver)) in SENDMSG#
```

Das SENDMSG# Makro fügt die Nachricht outmsg dem ether hinzu. Ebenso wird hier der ether indeterministisch verkleinert, um eventuellen Nachrichtenverlust ausdrücken zu können:

SPEZIFIKATION:

```
SENDMSG#
   choose newether with newether ⊆ ether ++ outmsg in ether := newether
```

Als Beispiel für einen Protokollschritt, der die balance verändert, betrachten wir den Empfang von REQ. Hier wird der aktuelle Transaktionswert auf der FROM Purse abgebucht, der Zustand auf EPA gesetzt und eine VAL Nachricht erzeugt:

SPEZIFIKATION:

```
REQ#
   balance(receiver) := balance(receiver) - pdAuth(receiver).value,
   state(receiver) := EPA;
   let outmsg = val(pdAuth(receiver)) in SENDMSG#
```

Die anderen Protokollschritte sind vollständig analog spezifiziert. Letztendlich ist noch das Abbrechen einer Transaktion auf Grund einer fehlerhaften Eingabe interessant: Die Transaktion muss im exLog aufgezeichnet werden, wenn der Zustand aktuell EPA oder EPV ist. Nur dann ist die Transaktion kritisch und nur dann kann der Verlust einer Nachricht auch zum Verlust von Geld führen. Darüber hinaus muss bei falscher Eingabe die Sequenznummer in-

krementiert werden und der Zustand auf IDLE zurückgesetzt werden. Zusammen erhalten wir so:

SPEZIFIKATION:

```
LOGIFNEEDED#
   if state(receiver) = EPA ∨ state(receiver) = EPV
   then exLog(receiver) := exLog(receiver) ++ pdAuth(receiver)

ABORT#
   choose n with nextSeqNo(receiver) ≤ n in
      LOGIFNEEDED#;
      state(receiver) := IDLE,
      nextSeqNo(receiver) := n;
```

12.4 Mondex als PROSECCO Modell auf Ebene 3

Die eben vorgestellte Spezifikationsebene kann nun zu einer PROSECCO Spezifikation verfeinert werden. Diese nächste Ebene ist dann auch gleichzeitig Ausgangspunkt für die Implementierung. In diesem Abschnitt stellen wir die Spezifikationsstruktur der PROSECCO Ebene vor. Eine Beschreibung des Verfeinerungsbeweises von Ebene 2 auf 3 würde an dieser Stelle zu weit führen und kann [73] entnommen werden.

Insbesondere werden nun hier bereits implementierungsspezifische Beschränkungen z.B. für die Längen von Datentypen eingeführt, um dann auf der Code-Ebene nicht noch eine zweite Komplexitätsstufe für die Verfeinerung zu schaffen. Insbesondere bedeutet dies, dass auf der PROSECCO Ebene von Mondex folgende Einschränkungen eingeführt werden:

- Die authentischen Purse Namen sind genau die Zahlenwerte mit 8 Byte Länge
- Die Sequenznummern der Purses sind im Bereich $0 \leq \mathsf{seqNo} \leq 32767$
- Die Kontostände der Purses sind im Bereich $0 \leq \mathsf{balance} \leq 32767$
- Das exLog hat eine maximale Länge von MAX_LOG_LENGTH = 10
- Zur Kennzeichnung der Nachrichten und des internen Zustands werden Konstanten STATE_IDLE, STATE_EPA, INS_START_FROM, INS_REQ usw. verwendet, die jeweils ein Byte lang sind.

Damit sind alle Möglichkeiten für unbeschränkte Werte in der Mondex Anwendung beseitigt und die Grundlage für eine Implementierung geschaffen. Die PROSECCO ASM erhält damit als Zustandsfunktionen für die Darstellung der internen Zustände der Purses folgende Funktionen:

SPEZIFIKATION:

```
name : agent → IntDoc
sequenceNo : agent → int
pbalance : agent → int
pstate : agent → int
ppdAuth : agent → Document
pexLog : agent → Documentlist
exLogCounter : agent → int
sesskey : agent → Key
```

Der pstate ist dabei entweder IDLE, EPA, EPV oder EPR. Die vorher bereits beschriebenen paydetails werden jetzt durch ein Document ppdAuth repräsentiert. Dieses Document hat hierbei immer folgenden Aufbau:

```
ppdAuth(agent) = Doclist(   IntDoc(name(FROM))
                          + IntDoc(sequenceNo(FROM))
                          + IntDoc(name(TO))
                          + IntDoc(sequenceNo(TO))
                          + IntDoc(amount))
```

Das Log der Purses wird auf der PROSECCO Ebene durch eine Documentlist repräsentiert. Dies ist einfach eine Liste mit Dokumenten mit dem eben beschriebenen Aufbau. Der exLogCounter ist schließlich eine zusätzliche Zustandsfunktion, die die Länge dieser Liste angibt.

Der sesskey ist auf der PROSECCO Ebene erst neu eingeführt worden. Er ist ein auf allen echten Karten gespeicherter geheimer symmetrischer Schlüssel. Seine Einführung begründet sich dadurch, dass nun keine impliziten Annahmen über die Fälschbarkeit bzw. Geheimhaltung von Nachrichten mehr gemacht werden dürfen, sondern statt dessen mit dem für PROSECCO Spezifikationen generischen Angreifermodell gearbeitet wird. Wir führen also auf der PROSECCO Ebene ein kryptographisches Protokoll ein. Dies wird zunächst kurz schematisch vorgestellt ($\{X\}_k$ bedeutet hierbei symmetrische Verschlüsselung von Datum X mit Schlüssel k):

Terminal → To	GETDATA
To → Terminal	name(To) + seqNo(To)
Terminal → From	STARTFROM + name(To) + value + sequenceNo(To)
From → Terminal → To	$\{$STARTTO + paydetails$\}_{sesskey}$
To → Terminal → From	$\{$REQ + paydetails$\}_{sesskey}$
From → Terminal → To	$\{$VAL + paydetails$\}_{sesskey}$
To → Terminal → From	$\{$ACK + paydetails$\}_{sesskey}$

wobei paydetails = name(From) + sequenceNo(From) + name(To) + sequenceNo(To) + amount. Für eine PROSECCO Spezifikation müssen die Nachrichten der Mondex Anwendung mittels dem Document dargestellt werden. Dazu werden einfach der EncDoc Typ für Verschlüsselung und der Doclist Typ für Konkatenation verwendet. Es ergibt sich die Nachrichtenstruktur

in Tabelle 12.12. Zusätzlich wurde noch eine Nachricht GETBALANCE eingeführt, um den Kontostand der Purse abzufragen, sowie eine Nachricht GETSTATE, um den aktuellen Zustand pstate abzufragen (letztere aus Gründen des Debuggings und der Anzeige des Kartenzustands im Terminal in der späteren Implementierung).

GETDATA	IntDoc(INS_GET_DATA)
GETDATA Response	Doclist(IntDoc(name(agent)) + IntDoc(sequenceNo(agent)))
GETBALANCE	IntDoc(INS_GET_BAL)
GETBALANCE Response	IntDoc(pbalance(agent))
GETSTATE	IntDoc(INS_GET_STATE)
GETSTATE Response	IntDoc(pstate(agent))
STARTFROM	Doclist(IntDoc(INS_START_FROM) + Doclist(IntDoc(name(to)) + IntDoc(value) + IntDoc(sequenceNo(to))))
STARTTO	EncDoc(sesskey(from), Doclist(IntDoc(INS_START_TO) + Doclist(IntDoc(name(from)) + IntDoc(sequenceNo(from)) + IntDoc(name(to)) + IntDoc(sequenceNo(to)) + IntDoc(amount)))))
REQ	EncDoc(sesskey(to), Doclist(IntDoc(INS_REQ) + Doclist(IntDoc(name(from)) + IntDoc(sequenceNo(from)) + IntDoc(name(to)) + IntDoc(sequenceNo(to)) + IntDoc(amount)))))
VAL	EncDoc(sesskey(from), Doclist(IntDoc(INS_VAL) + Doclist(IntDoc(name(from)) + IntDoc(sequenceNo(from)) + IntDoc(name(to)) + IntDoc(sequenceNo(to)) + IntDoc(amount)))))
ACK	EncDoc(sesskey(to), Doclist(IntDoc(INS_ACK) + Doclist(IntDoc(name(from)) + IntDoc(sequenceNo(from)) + IntDoc(name(to)) + IntDoc(sequenceNo(to)) + IntDoc(amount)))))

Tabelle 12.12: Die Nachrichten der Mondex Anwendung

Mit diesen Nachrichten und den internen Zuständen kann nun die Mondex Anwendung mit der PROSECCO Methodik spezifiziert werden. Zunächst müssen Agententypen definiert werden:

SPEZIFIKATION:

```
agent =  purse-cardlet (. .no : nat) with purse-cardlet?
       | user (. .no : nat) with user?
       | terminal (. .no : nat) with terminal?
       | attacker (. .no : nat) with attacker?
       | fake-cardlet (. .no : nat) with fake-cardlet?
```

Neben der Purses (hier durch den Datentyp purse-cardlet umgesetzt) wird noch der Benutzer des Systems (als Initiator für die Geldübertragung), das Chipkartenterminal und der Angreifer spezifiziert. Wie in [71] beschrieben, spezifizieren wir auch noch einen Typ fake-cardlet, der vom Angreifer gefälschte Chipkarten modelliert. Die Spezifikationen für Benutzer, Terminal und Angreifer sind für den Kontext dieser Arbeit nicht wichtig. Wir fokussieren hier daher auf die Spezifikation für die purse-cardlets.

Die Hauptregel der PROSECCO ASM für Mondex hat folgende Gestalt:

SPEZIFIKATION:

```
 MONDEX-STEP
choose asm-step with (
  (asm-step = user-agent-step → ∃ agent. (user?(agent) ∧ exagent(agent))) ∧
  (asm-step = attacker-agent-step → ∃ agent. (attacker?(agent) ∧ exagent(agent))) ∧
  (asm-step = terminal-agent-step → ∃ agent. (terminal?(agent) ∧ exagent(agent))) ∧
  (asm-step = purse-agent-step → ∃ agent. (cardlet-agent(agent) ∧ exagent(agent)))
) in {
  if asm-step = connect then
    CONNECT(...)
  else if asm-step = disconnect then
    DISCONNECT(...)
  else if asm-step = user-agent-step then
    choose agent with (user?(agent) ∧ exagent(agent)) in USER(...)
  else if asm-step = attacker-agent-step then
    choose agent with (attacker?(agent) ∧ exagent(agent)) in ATTACKER(...)
  else if asm-step = terminal-agent-step then
    choose agent with (terminal?(agent) ∧ exagent(agent)) in TERMINAL
  else if asm-step = purse-agent-step then
    choose agent with (cardlet-agent(agent) ∧ exagent(agent)) in PURSE-CARDLET
}
```

Das erste **choose** wählt dabei einen Schritttyp (z.B. Schritt einer Purse, Verbindungsaufbau, Schritt des Angreifers, usw.). Für konkrete Protokollschritte kommen dabei nur existierende Agenten in Frage ([71] sieht eine bestimmte Anzahl von Instanzen jedes Agententyps vor, dies wird durch das Prädikat exagent(agent) sichergestellt). Im Anschluss führt die Regel das jeweils zum Agententyp und Schritt passende ASM Makro aus. Wir zeigen hier nur PURSE-CARDLET:

SPEZIFIKATION:

```
PURSE-CARDLET
  if inputs(agent)(1) ≠ [] ∧ exLogCounter(agent) < MAX_LOG_LENGTH
  then let
     indoc = inputs(agent)(1).first,
     outdoc = ⊥,
     insbyte = 0
  in
  {
     inputs(agent) := inputs(agent)[1 ; inputs(agent)(1).rest];
     CHECKINDOC;
     GETINSBYTE;
     if indoc ≠ ⊥ then
        indoc := get-part(indoc,2);

     if insbyte = INS_START_FROM then PSTARTFROM
     else if insbyte = INS_START_TO then PSTARTTO
     else if insbyte = INS_REQ then PREQ
     else if insbyte = INS_VAL then PVAL
     else if insbyte = INS_ACK then PACK
     else if insbyte = INS_GET_BAL then GETBAL
     else if insbyte = INS_GET_DATA then GETDATA
     else if insbyte = INS_GET_STATE then GETSTATE
     else if insbyte = 0 then ABORT;

     if outdoc ≠ ⊥ then
        SEND
  }
```

Diese Spezifikation ist nun im Vergleich zu Ebene 2 aus Kap. 12.3.4 bereits wesentlich implementierungsnäher gehalten. Die ASM Regel wählt zunächst ein Eingabedokument indoc aus den inputs(agent)(1) (es gibt auf Chipkarten nur einen Eingabeport, s. Kap. 11). Danach wird die Eingabequeue entsprechend verkürzt und mittels der beiden Makros CHECKINDOC und GETINSBYTE die Eingabe auf Wohlgeformtheit überprüft, gegebenenfalls entschlüsselt und in der Variablen insbyte der Protokollschritt des Eingabedokumentes festgehalten. Anhand dieses Schritttyps wird im Anschluss das jeweils passende Makro für den Schritt aufgerufen. Diese Schritte produzieren dann eine Ausgabe outdoc, die am Schluss der Regel mittels SEND

verschickt wird (völlig analog zur Cindy Anwendung in Kap. 5 und 7).
Das Makro CHECKINDOC ist wie folgt spezifiziert:

SPEZIFIKATION:

```
CHECKINDOC
  if is-encdoc(indoc) then
  {
    indoc := decrypt(sesskey(agent),indoc);
    if ¬ (isCorrectSecuredMessage(indoc))
    then indoc := ⊥
  }
  else if is-intdoc(indoc) then
  {
    if ¬ (isCorrectBalDataState(indoc))
    then indoc := ⊥
  }
  else if is-doclist(indoc) then
  {
    if ¬ (
          # indoc.list = 2
       ∧ is-intdoc(get-part(indoc,1))
       ∧ is-doclist(get-part(indoc,2))
       ∧ get-part(indoc,1).int = INS_START_FROM
       ∧ # get-part(indoc,2).list = 3
       ∧ is-intdoc(get-part(get-part(indoc,2),1))
       ∧ is-intdoc(get-part(get-part(indoc,2),2))
       ∧ is-intdoc(get-part(get-part(indoc,2),3))
       )
    then indoc := ⊥
  }
  else indoc := ⊥
```

CHECKINDOC ist aufgeteilt in vier verschiedene Fälle: Das Eingabedokument ist vom Typ EncDoc, Doclist, IntDoc oder von keinem von diesen. Im ersten Fall wird die Eingabe mit dem Schlüssel des aktuellen Agenten sesskey(agent) entschlüsselt und überprüft, ob es sich um eine wohlgeformte Eingabe im Sinne von Tabelle 12.12 handelt (durch das Prädikat isCorrectSecuredMessage - hier nicht näher erläutert). Bei einem IntDoc muss es sich analog um eine korrekte GETDATA, GETSTATE oder GETBALANCE Nachricht handeln (Prädikat isCorrectBalDataState). Bei einer Doclist schließlich (auf diesen Eingabetyp fokussieren wir die folgenden Erläuterungen) muss es sich um eine STARTFROM Nachricht handeln. Die Bedingungen für eine korrekt geformte STARTFROM Nachricht sind in obiger Spezifikation nicht in ein Prädikat gefasst, um hier einmal den Unterschied zu einer abstrakteren Spezifikation (wie

etwa auf Ebene 2) herauszustellen: Vergleichend zur Struktur von STARTFROM Nachrichten aus Tabelle 12.12 ergeben sich die Eigenschaften: Länge der Eingabe gleich 2, erstes Dokument vom Typ IntDoc, zweites Dokument vom Typ Doclist, Wert des ersten IntDocs gleich INS_START_FROM, Länge der zweiten Doclist gleich 3, alle drei Inhalte der zweiten Doclist vom Typ IntDoc. Genau diese Eigenschaften sind in obiger Spezifikation explizit aufgeführt. Sollte keine der Wohlgeformtheitsbedingungen greifen, so wird indoc auf $\perp$ gesetzt.

Es verbleibt noch das GETINSBYTE Makro:

SPEZIFIKATION:

```
GETINSBYTE
  insbyte := 0;
  if is-doclist(indoc) ∧ (
    (get-part(indoc,1).int = INS_START_FROM ∨
    (get-part(indoc,1).int = INS_START_TO ∨
    (get-part(indoc,1).int = INS_REQ ∧ pstate(agent) = STATE_EPR ) ∨
    (get-part(indoc,1).int = INS_VAL ∧ pstate(agent) = STATE_EPV ) ∨
    (get-part(indoc,1).int = INS_ACK ∧ pstate(agent) = STATE_EPA ))
  then insbyte := get-part(indoc,1).int
  else if is-intdoc(indoc) ∧ (
    indoc.int = INS_GET_BAL ∨
    indoc.int = INS_GET_DATA ∨
    indoc.int = INS_GET_STATE)
  then insbyte := indoc.int
```

Das obige Makro soll den Typ der Eingabe in der Variablen insbyte speichern. Zusätzlich soll das Makro angeben, ob der aktuelle Eingabetyp zum derzeitigen internen Zustand passt. REQ Nachrichten sollen z.B. ausschließlich im Zustand EPR angenommen werden. Es bleibt hier anzumerken, dass STARTFROM und STARTTO Nachrichten in jedem Zustand akzeptiert werden. Der Grund ist folgender: Wird eine Transaktion durch einen Fehler wie etwa Nachrichtenverlust oder einfaches vorzeitiges Entfernen der Karte aus dem Lesegerät abgebrochen, kann es sein, dass eine Karte in einem Zustand ungleich IDLE verbleibt. Startet man nun mit dieser Karte zu einen späteren Zeitpunkt eine neue Transaktion, so würde diese scheitern, würde man STARTFROM oder STARTTO nicht akzeptieren. Dies ist aus Benutzersicht nicht wünschenswert. Damit erlauben wir diese Nachrichten immer.

Um die weitere Spezifikation zu beleuchten, betrachten wir nun den STARTFROM Schritt:

SPEZIFIKATION:

```
PSTARTFROM
  let pmsgna = get-part(indoc, 1),
      pvalue = get-part(indoc, 2),
      seqnoOther = get-part(indoc, 3) in
  {
```

```
  CHECKNAME;
  CHECKBALANCEMINUS;
  CHECKSEQNOOTHERPURSE;
  if   pmsgna ≠ ⊥ ∧ pvalue ≠ ⊥ ∧ seqnoOther ≠ ⊥
    ∧ ¬ 32767 ≤ sequenceNo(agent) then
  {
    if pstate(agent) ≠ STATE_IDLE then ABORT;
    if exLogCounter(agent) < MAX_LOG_LENGTH then
    {
      ppdAuth(agent) := doclist(   name(agent)
                                 + intdoc(sequenceNo(agent))
                                 + pmsgna
                                 + seqnoOther
                                 + pvalue),
      sequenceNo(agent) := sequenceNo(agent) + 1;
      pstate(agent) := STATE_EPR;
      outdoc := encdoc(sesskey(agent),
                       doclist(intdoc(INS_START_TO) + ppdAuth(agent)))
    }
  }
else ABORT
}
```

Alle Überprüfungen für den eigentlichen Protokollschritt finden nun zu Beginn des PSTART-FROM Makros statt: Der Name pmsgna, der in der Eingabenachricht als TO Purse angegeben ist, muss authentisch (also 8 byte lang, s. oben) und ungleich dem Namen der FROM Purse sein (man kann kein Geld von einer Purse auf diese selbst übertragen). Dies geschieht im Makro CHECKNAME. Die pbalance(agent) der FROM Purse muss größer oder gleich als der zu übertragende Betrag pvalue sein (geprüft durch das Makro CHECKBALANCEMINUS). Schließlich muss die Sequenznummer seqnoOther der anderen Purse im Bereich zwischen 0 und 32767 liegen (Makro CHECKSEQNOOTHERPURSE). Zu beachten ist hierbei, dass alle drei Eingabeteile pmsgna, pvalue und seqNoOther vom Type Document sind. Daher setzen die drei Check-Makros die jeweiligen Dokumente auf $\perp$, falls etwas nicht stimmen sollte. Sind alle drei ungleich $\perp$ und ist zudem die eigene Sequenznummer sequenceNo(agent) noch im zulässigen Bereich, so kann der eigentliche Protokollschritt begonnen werden. Vorher muss allerdings noch die bereits oben angesprochene Komplikation bedacht werden: Karten können STARTFROM Nachrichten in einem Zustand ungleich IDLE erhalten. In diesen Fällen ist potentiell bei einer vorherigen Transaktion Geld verloren gegangen und es muss zunächst noch ein ABORT Schritt durchgeführt werden, der die Details der vergangenen Transaktion ins Log schreibt (s. unten). Dann kann der STARTFROM Schritt der aktuellen Transaktion durchgeführt werden: Ist das exLog noch nicht durch den ABORT Vorgang voll geworden (also exLogCounter(agent) < MAX_LOG_LENGTH oben), so können die Transaktionsdetails ppdAuth initialisiert werden und die Sequenznummer inkrementiert werden. Der Zustand pstate wird auf STATE_EPR gesetzt und zuletzt ein verschlüsseltes STARTTO Dokument als Antwort outdoc erzeugt.

Zur vollständigen Illustrierung fehlt noch das ABORT Makro für das Erstellen von Log Ein-

trägen:

SPEZIFIKATION:

```
ABORT
  if pstate(agent) = STATE_EPA ∨ pstate(agent) = STATE_EPV then
  {
    pexLog(agent) := pexLog(agent) + ppdAuth(agent),
    exLogCounter(agent) := exLogCounter(agent) + 1
  }
  pstate(agent) := STATE_IDLE
```

Befindet sich die aktuelle Transaktion in einem kritischen Zustand (wie oben in Kap. 12.3.4 erläutert im Zustand EPA oder EPV), so werden die aktuellen Transaktionsdetails ppdAuth zur exLog Liste hinzugefügt und der exLogCounter erhöht.

Die restlichen Protokollschritte sind analog zu STARTFROM spezifiziert. Hier betrachten wir zur Illustration lediglich noch REQ, das (gesetzt dem Fall die Eingabe stimmt mit den aktuellen Transaktionsdetails überein) den Kontostand pbalance verringert, den Zustand auf EPA setzt und eine VAL Ausgabenachricht erzeugt:

SPEZIFIKATION:

```
PREQ
  if indoc = ppdAuth(agent) then
  {
    pbalance(agent) := pbalance(agent) - get-part(ppdAuth(agent), 5).int,
    pstate(agent) := STATE_EPA,
    outdoc := encdoc(sesskey(agent), doclist(intdoc(INS_VAL) + ppdAuth(agent)))
  }
  else ABORT
```

Dies schließt die Beschreibung des PROSECCO Modells ab. Das hier vorgestellte Modell ist eine korrekte ASM Verfeinerung der Ebene 2 aus Kap 12.3.4.

12.5 Die Implementierungsebene

Ziel dieses Kapitels ist es, die vorangegangene PROSECCO Spezifikation in Java zu implementieren. Dabei ist bei Beschreibung der PROSECCO Ebene bereits aufgefallen, dass die Spezifikation recht implementierungsnah gehalten wurde. Dies zeigt sich nun auch in der Umsetzung: Hier ist keine strukturelle Anpassung in Java mehr nötig. Einige Überprüfungen werden in der Java Programmiersprache zwar komplizierter, im Großen und Ganzen entspricht der Kontroll-

fluss der Implementierung und auch die Modularisierung mit Methoden genau der abstrakten Spezifikation. Dies kommt später auch der Struktur der Verfeinerungsbeweise in Kap. 12.7.1 zu Gute.

Bevor wir die Implementierung beschreiben folgt zunächst noch die konkrete Einbettung des Java Quellcodes in eine konkrete ASM Spezifikationsebene. Diese folgt dabei der Vorgehensweise aus Kap. 5. Die Hauptregel MONDEX-STEP-CONCRETE auf der konkreten Ebene entspricht dabei genau der abstrakten Ebene (s. Seite 212), lediglich das Makro PURSE-CARDLET wurde durch das unten gezeigte Makro PURSE-CARDLET-CONCRETE ersetzt. Die Zustandsfunktionen für die abstrakten purse-cardlet Agenten wurden entfernt und durch eine Zustandsfunktion $\mathsf{store}_c : \mathsf{agent} \rightarrow \mathsf{store}$ ersetzt. Man erhält damit (mit den Definitionen für FROMSTORE und TOSTORE wie in Kap. 8.6):

SPEZIFIKATION:

```
PURSE-CARDLET-CONCRETE
  let outdoc = ⊥ in {
    TOSTORE;
    choose st1 with
    ∃ st. st = storec(agent) ∧
      ⟨st; class mondexOnCard.Purse Purse.theinstance.step();⟩ (st1 = st)
    in
      FROMSTORE(..., storec, st1);
    if outdoc ≠ ⊥ then
      SEND
  }
```

Die eigentliche Implementierung wird damit in einer Java Klasse `Purse` realisiert. Analog zu Cindy findet sich der vollständige Quellcode in Anhang 13.1. Hier wird lediglich die grundlegende Struktur vorgestellt:

```
1  public class Purse {
2    private static Purse theinstance;
3    private static SimpleComm initcomm;
4
5    private static final byte STATE_IDLE = 1;
6    ...
7    private static final byte STATE_EPA  = 4;
8
9    private static final byte INS_START_FROM = 1;
10   ...
11   private static final byte INS_GET_STATE  = 8;
12
13   private byte[]    name;
14   private short     sequenceNo;
15   private short     balance;
```

```
16   private byte      state;
17   private Doclist   pd;
18   private short     exLogCounter;
19   private Doclist[] exLog;
20   private Key       key;
21
22   private SimpleComm comm;
23
24   public Purse(SimpleComm initcomm, Document initdata) {
25     ...
26   }
27
28   public void step(){
29     ... // s. unten
30   }
31 }
```

Die Klasse beinhaltet zunächst Deklarationen für die verschiedenen Konstanten für `state` und Instruktionen für Nachrichten. Die Zustandsfunktionen der abstrakten Prosecco ASM Ebene 3 sind durch die Felder `name`, `sequenceNo`, `balance`, `state`, `pd` (für ppdAuth), `exLogCounter`, `exLog` und `key` (für sesskey) umgesetzt. Dazu benötigen wir noch einen Verweis auf das `SimpleComm` Interface zur Kommunikation (s. die Übertragung der Kommunikationsschicht auf Chipkarten in Kap. 11). Als Besonderheit von Chipkarten müssen wir bei der Mondex Anwendung auf die dynamische Allokation von Speicher zur Laufzeit verzichten. Insbesondere gilt dies für das Log. Das Feld exLog (ein `Doclist` Array) kann nicht zur Laufzeit bei jedem neuen Logeintrag um einen Eintrag verlängert werden. Wir initialisieren das Feld daher im Konstruktor (s. Anhang 13.1 gleich mit der maximalen Anzahl von Logeinträgen. Der `exLogCounter` gibt in der Implementierung den Index des jeweils nächsten freien Eintrags in diesem Array an (in der abstrakten Spezifikation war der exLogCounter(agent) die Länge von exLog(agent) - was den selben Wert wie in der Implementierung ergibt). Wir haben also:

```
1  public Purse(...){
2    ...
3    initExLog(MAX_LOG_LENGTH);
4  }
5
6  private void initExLog(short len) {
7    exLog        = new Doclist[len];
8    exLogCounter = (short)0;
9  }
```

Als Initialisierungsbedingung der konkreten ASM verwenden wir ein Prädikat init-concrete, das sämtliche Zustandsfunktionen korrekt initialisiert und unter anderem auch den Konstruktor der `Purse` Klasse aufruft. Wie in obigem Quellcode für `Purse` ersichtlich ist dazu eine Referenz auf ein `SimpleComm` Interface und eine Referenz auf Initialisierungdaten `initdata` nötig. Letztere sind eine `Doclist` bestehend aus dem `name` der `Purse` als `IntDoc` sowie der initialen `balance` und die gewünschte Länge des Logs (ebenso `IntDocs`). Letztere ist,

vorgegeben durch die abstrakte Spezifikation, gleich MAX_LOG_LENGTH = 10. Somit ergibt sich als Initialisierungsbedingung (dem allgemeinen Rahmen aus Kap. 5.4.3 folgend):

SPEZIFIKATION:

init-concrete:
init-concrete(..., inputs, $store_c$)
$\leftrightarrow$ ($\forall$ agent. inputs(agent) = (λ n. [])
$\wedge$ ...
$\wedge$ (cardlet-agent(agent)
$\rightarrow \exists$ st,st_1,r .
okinitstoreMONDEX(st)
$\wedge$ r $\times$ st_1 =
doc2java(doclist(name(agent)
+ intdoc(pbalance(agent))
+ intdoc(MAX_LOG_LENGTH)), st)
$\wedge$ $\langle st_1$; `Purse.theinstance =`
`new Purse(Purse.initcomm, r);`$\rangle$ st_1 = $store_c$(agent)))

okinitstoreMONDEX fordert dabei eine initiale Wohlgeformtheit des Speichers, in dessen Kontext der Konstruktor aufgerufen wird. Hier wird z.B. festgelegt, dass der Ausführungszustand der JVM zu Beginn st[.mode] = normal sein muss. Auch Typkorrektheit für das `SimpleComm` Interface `Purse.initcomm` wird hier gefordert. Danach wird der Konstruktor im Kontext eines Speichers st_1 aufgerufen, der aus st durch Hinzufügen oben genannter Initialisierungsparameter entstanden ist. Das Ergebnis ist der initiale Wert von $store_c$(agent).

Wenden wir uns nun der eigentlichen Protokollimplementierung zu. Die Einbettung in die JavaCard API erfolgt genau wie in Kap. 11.3.1 beschrieben. Es kann die einfache Variante (ohne iteriertes Senden und Empfangen von APDUs für einen Protokollschritt) verwendet werden, da alle Mondex Nachrichten in eine APDU passen.

Analog zur Cindy Applikation aus Kap. 7 implementieren wir die Protokollfunktionalität in einer Methode `step()` in der Klasse `Purse`. Diese folgt nun der Strukturvorgabe der obigen PROSECCO Spezifikation von Seite 213:

```
1  public void step() {
2    Document outdoc = null;
3    Document indoc = null;
4    if(comm.available() && exLogCounter < exLog.length)
5      indoc = comm.receive();
6    else return;
7    indoc = checkIndoc(indoc);
8    switch(getInsByte(indoc)) {
9      case INS_START_FROM:
10       outdoc = startFrom(indoc.getPart((short)2));break;
11     case INS_START_TO:
12       outdoc = startTo(indoc.getPart((short)2));  break;
```

```
13      case INS_REQ:
14         outdoc = req(indoc.getPart((short)2));      break;
15      case INS_VAL:
16         outdoc = val(indoc.getPart((short)2));      break;
17      case INS_ACK:
18                  ack(indoc.getPart((short)2));      break;
19      case INS_GET_BAL:
20         outdoc = getBalance();                      break;
21      case INS_GET_DATA:
22         outdoc = getData();                         break;
23      case INS_GET_STATE:
24         outdoc = getState();                        break;
25      default:
26         abort();                                    break;
27    }
28    if(outdoc!=null)
29            comm.send(outdoc);
30  }
```

Ist eine Eingabe vorhanden (`comm.available()`) und sind noch Logeinträge verfügbar (für ein eventuelles Scheitern der Transaktion), so werden auch in der Java Implementierung Methode `checkIndoc` und `getInsByte` verwendet, um die Eingabe auf Wohlgeformtheit zu prüfen und den Typ der Eingabe festzustellen. Anschließend wird anhand des Typs auch hier eine passende Methode für den jeweiligen Protokollschritt ausgewählt.

Betrachten wir hier zur PROSECCO Spezifikation die Implementierungen von `checkIndoc` und `getInsByte`. Bei der folgenden Implementierung für `checkIndoc` wurden der Einfachheit halber die Teile für verschlüsselte Eingaben und `IntDocs` ausgelassen. Damit verbleibt der Check für den STARTFROM Schritt:

```
1  private Document checkIndoc(Document indoc) {
2   if(indoc == null) return null;
3   if(indoc.is_encdoc()) {
4     ...
5   }
6   else if(indoc.is_intdoc()) {
7     ...
8   }
9   else if(indoc.is_doclist()) {
10    Document[] docs = ((Doclist)indoc).getDocs();
11    if(docs.length==2) {
12     Document part1 = indoc.getPart((short)1);
13     Document part2 = indoc.getPart((short)2);
14     if(   part1 != null && part1.is_intdoc()
15        && part2 != null && part2.is_doclist()) {
16       byte[] part1value = part1.getValue();
17       Document[] part2docs = ((Doclist)part2).getDocs();
18       if(part1value.length == 2 &&
```

```
19          part1value[0] == 0 &&
20          part1value[1]==INS_START_FROM) {
21        if(part2docs.length==3 &&
22           part2docs[0] != null &&
23           part2docs[0].is_intdoc() &&
24           part2docs[1] != null &&
25           part2docs[1].is_intdoc() &&
26           part2docs[2] != null &&
27           part2docs[2].is_intdoc())
28          return indoc;
29       }
30     }
31   }
32  }
33  return null;
34 }
```

Diese Implementierung ist nun zu vergleichen mit der ASM Spezifikation im vorangegangen Abschnitt auf Seite 214. Später im Verfeinerungsbeweis werden wir beweisen, dass diese Implementierung genau der Spezifikation entspricht. Typchecks auf den Eingabeteilen, die im abstrakten mittels der für den Datentyp aus seiner Spezifikation generierten Testprädikate wie z.B. is-intdoc(part1) erfolgen, werden hier durch Methoden auf `Document` implementiert (z.B. `part1.is-intdoc()`), die intern dann in allen Subklassen überschrieben werden. Eine Implementierung mittels `instanceof` wäre hier auch möglich gewesen, die vorgestellte Methodik lässt aber eine intuitivere Definition von Lemmata zu (Methodenaufrufe entsprechen hier genau abstrakten Prädikaten auf dem Ergebnis von java2doc). Zu beachten ist ferner, dass die Implementierung mehr Überprüfungen durchführen muss als die abstrakte Spezifikation. So kommt bei jedem Teildokument noch eine Überprüfung auf Ungleichheit zu `null` hinzu. Diese ist im abstrakten implizit, da `null` ja $\perp$ repräsentiert und dies zu allen anderen abstrakten Dokumenten verschieden ist. In der Implementierung könnte das Vorkommen von `null` dagegen `NullPointerExceptions` auslösen. Auch die im abstrakten einfache Überprüfung get-part(indoc,1).int = INS_START_FROM muss in der Implementierung durch drei einzelne Checks auf dem `byte[ ]` des jeweiligen `IntDocs` umgesetzt werden (wir implementieren der Einfachheit halber in Mondex alle Eingaben mit Zahlenwerten kleiner 32767 als `IntDocs` mit 2 Byte, wobei unter Umständen das erste Byte hier 0 sein kann):

```
part1value.length == 2 &&
part1value[0] == 0 &&
part1value[1]==INS_START_FROM
```

Analog verhält es sich bei der Implementierung von `getInsByte`:

```
private byte getInsByte(Document d) {
  byte[] insba = null;
  if(d==null) return 0;
  if(d.is_doclist())
   insba = d.getPart((short)1).getValue();
  else
   insba = d.getValue();
```

```
8
9   if(insba == null || insba[0] != 0 || insba.length != 2)
10    return 0;
11  byte ins = insba[1];
12
13  if(   ins==INS_START_FROM
14     || ins==INS_START_TO
15     ||(ins==INS_REQ && state==STATE_EPR)
16     ||(ins==INS_VAL && state==STATE_EPV)
17     ||(ins==INS_ACK && state==STATE_EPA)
18     || ins==INS_GET_BAL
19     || ins==INS_GET_DATA
20     || ins==INS_GET_STATE) return ins;
21  else return (byte)0;
22 }
```

Bei dieser Methode fällt eine leicht andere Struktur auf als in der Spezifikation auf Seite 215 vorgegeben. Es wird zunächst der `byte[ ]` Wert des Kommandobytes aus dem Eingabedokument extrahiert (`insba`). Je nach Typ der Eingabe (`Doclist` oder nur `IntDoc` für z.B. GET_DATA) findet sich dieser `byte[ ]` Wert an unterschiedlichen Stellen im Dokument. Nach Extraktion wird überprüft (analog zu `checkIndoc` oben), ob das `byte[ ]` ein wohlgeformtes Kommando mit 2 Byte Länge und erstem Byte gleich 0 ist. Erst wenn das erfolgt ist, können die gleichen Überprüfungen wie in der abstrakten Spezifikation erfolgen.

Sind beiden Check-Methoden erfolgreich gelaufen, kann schließlich die Implementierung der Protokollschritte erfolgen. Auch hier betrachten wir als Beispiel wieder STARTFROM:

```
1  private Document startFrom(Document indoc) {
2    Document dmsgna =
3       checkName(indoc.getPart((short)1));
4    short value_short =
5       checkBalanceMinus(indoc.getPart((short)2));
6    short nextSeqNoToPurse =
7       checkSeqNoOtherPurse(indoc.getPart((short)3));
8
9    if(   32767 <= sequenceNo
10     || nextSeqNoToPurse == -1
11     || value_short == -1
12     || dmsgna == null) {
13     abort();
14     return null;
15   }
16
17   if(state != STATE_IDLE)
18     abort();
19
20   if(exLogCounter < exLog.length){
21     mkpd(name,sequenceNo,dmsgna.getValue(),
```

```
22          nextSeqNoToPurse,value_short);
23      sequenceNo++;
24      state = STATE_EPR;
25      return generateOutmsg(INS_START_TO);
26    }
27    else return null;
28  }
```

Hier können wir der Spezifikation wieder etwas einfacher folgen: Analog zur ASM Regel auf Seite 215 verwenden wir zunächst drei Methoden `checkName`, `checkBalanceMinus` und `checkSeqNoOtherPurse`, um die Eingabe mit dem internen Zustand abzugleichen und zu überprüfen, ob der Protokollschritt möglich ist. Ist etwas nicht in Ordnung, wird dies auch hier in der Implementierung durch bestimmte Rückgabewerte dieser Methoden angezeigt (-1 bzw. `null`). Sollte dies der Fall sein oder die Sequenznummer zu groß sein, wird `abort()` ausgeführt und der Schritt beendet. Auch in der Implementierung kann noch der Fall eintreten, dass ein STARTFROM empfangen wird, wenn der Zustand der Purse nicht IDLE ist. Auch hier muss in diesem Fall ein `abort` eingefügt werden. Ist danach das Log noch nicht voll, können die paydetails (Feld `pd`) aktualisiert werden. Dies geschieht hier durch die Methode `mkpd(...)`. In der Implementierung ist dies komplizierter als in der Spezifikation, in der eine einfache Zuweisung paydetails(agent) = ... das gewünschte erledigte. Hier muss dagegen relativ komplex jeder einzelne Wert innerhalb von `pd` durch Kopieren gesetzt werden. Speicherallokation ist auf Chipkarten nicht möglich. Ein einfaches Umsetzen der `byte[ ]` Pointer von der Eingabe an die jeweiligen Stellen innerhalb `pd` würde Sharing unterhalb der `Purse` erzeugen und außerdem Pointerwerte aus dem Pool der Transformationsschicht in der Klasse `Purse` referenzieren. Insbesondere letzteres würde beim nächsten Protokollschritt zu Seiteneffekten und falschen `pd` Inhalten führen, da diese `byte[ ]` aus dem Pool im nächsten Protokollschritt wieder verwendet werden (s. dazu die Erläuterungen zu Pooling in Kap. 11). Nach dem Inkrementieren der `sequenceNo` und nach Setzen des `state` auf `STATE_EPR` muss noch eine STARTTO Ausgabe erzeugt werden. Hier treten die gleichen Probleme wie beim Setzen von `pd` nochmals auf: Speicherallokation ist nicht möglich und einfaches Umsetzen von Pointern ebenso wenig. Folglich müssen in extra dafür vorallokierte Felder für die Ausgabenachrichten die jeweiligen Werte durch Kopieren der einzelnen Inhalte eingefügt werden (Methode `generateOutmsg`). Dies schließt die Implementierung von `startFrom` ab.

Analog zur ASM Spezifikation zeigen wir auch hier mit `req()` noch einen anderen Protokollschritt:

```
1  private Document req(Document indoc) {
2    if(!pd.equals(indoc)){
3      abort();
4      return null;
5    }
6    balance = (short)
7      (balance -
8       Util.getShort(pd.getPart((short)5).getValue(),
9                     (short)0));
10   state = STATE_EPA;
11   return generateOutmsg(INS_VAL);}
```

Hier kann der Vergleich der Eingabe mit den eigenen paydetails im Feld `pd` recht elegant durch eine generische in Java übliche `equals` Methode gelöst werden. Daneben folgt die Implementierung genau der Spezifikation. Der abzubuchende Betrag wird mittels der Hilfsmethode `Util.getShort` aus der JavaCard API aus einem `byte[ ]` innerhalb der paydetails (genauer: der fünfte Eintrag in `pd`) in einen `short` Wert umgewandelt. Das Generieren der Ausgabe erfolgt wie schon oben durch die Methode `generateOutmsg`.

12.6 Simulationsrelation

Wesentlich für die Korrektheit der Verfeinerung von der PROSECCO Spezifikation zum Java Code ist natürlich auch hier die Simulationsrelation, die die abstrakten Zustandsfunktionen mit den implementierten Speicherstrukturen in Beziehung setzt. Dem allgemeinen Rahmen aus Kap. 5.3.3 auf Seite 56 folgend müssen wir auch passende Invarianten für beide Ebenen definieren. Beginnend mit letzterem wollen wir beides im Folgenden beschreiben:

12.6.1 Invarianten

Die Simulationsrelation für Mondex wird durch die Vielzahl von Zeigerstrukturen recht komplex. Ein Beispiel für die steigende Komplexität der Strukturen ist das Log für paydetails in der Implementierung. Hierbei handelt es sich bei dem Feld `.exlog` um ein Array von `Doclist`. Jeder Eintrag selbst ist entweder `null` (an den Stellen nach dem `exLogCounter` bis zur `MAX_LOG_LENGTH` oder ein wohlgeformtes `Document`, das den Aufbau eines paydetail hat (s. oben). Damit zeigt das Diagramm in Abb. 12.4 eine gültige `exLog` Struktur (ähnlich zu UML Objektdiagrammen repräsentieren alle Kästchen eigene Referenzen im Speicher, Arrays sind als Folge von Kästchen gezeichnet, an den Pfeilen zwischen Referenzen sind jeweils die Felder oder Arrayindizes für den jeweiligen Zugriff angegeben, der Typ und ggf. die Länge der Referenzen ist jeweils an den Kästchen annotiert).

Wie ersichtlich benötigt jedes einzelne paydetail bereits 13 Referenzen. Die Eigenschaften, die die Wohlgeformtheit eines solchen Logs beschreiben, sind damit informell:

- Das `.exLog` Feld enthält eine Referenz ungleich `null` vom Typ `Doclist[]`
- Die Länge dieses Arrays ist `MAX_LOG_LENGTH`
- Jedes referenzierte Objekt zwischen Index 0 und Index `exLogCounter` ist ein wohlgeformtes paydetail. Dies bedeutet
  - Die Referenz ist von Typ `Doclist` und ungleich `null`
  - Das Array im `.docs` Feld der Referenz ist ungleich `null`
  - Das Array im `.docs` Feld der Referenz ist von Typ `Document[]` (und nicht etwa ein Subtyp)
  - Das Array im `.docs` Feld der Referenz umfasst 5 Einträge
  - Jeder Arrayeintrag ist ungleich `null` und vom Typ `IntDoc`
  - Die `byte[ ]` Werte der `IntDocs` sind ungleich `null`

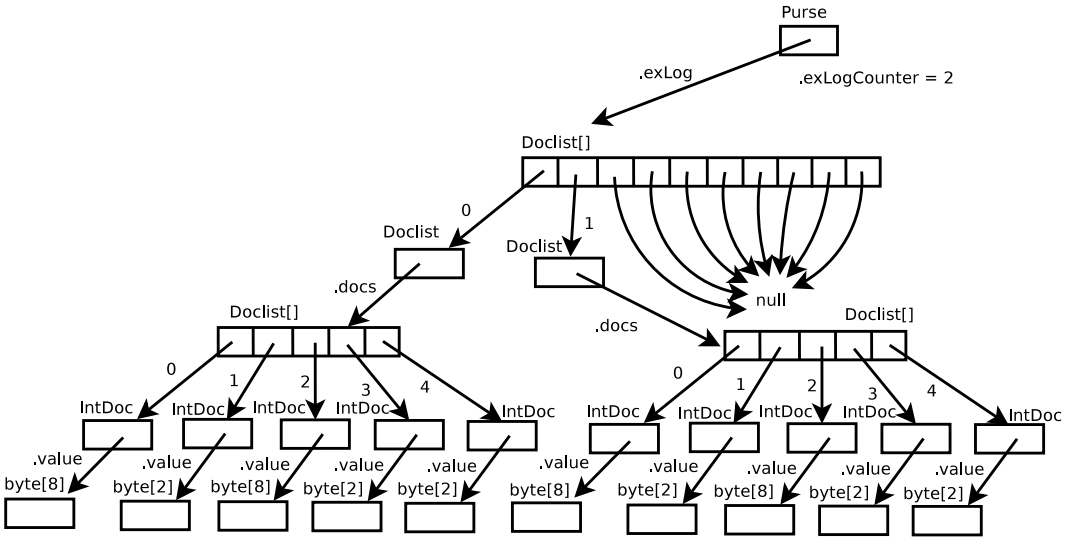


Abbildung 12.4: Exemplarische Log Einträge

– Die Arraylängen der `byte[ ]` Werte der `IntDocs` sind 8,2,8,2 und 2.

Formal gefasst ist dies (die beiden letzten obigen Punkte werden durch das Prädikat okarray beschrieben):

SPEZIFIKATION:

okLog-def:
okLog(st)
↔ st[.theinstance.exLog.length].intval = MAX_LOG_LENGTH
∧ validrefnotnull('docs2store', st[.theinstance.exLog].refval, mkarraytype(Doclist), st)
∧ (∀ i. (0 ≤ i ∧ i < s2i(st[.theinstance.exLogCounter].shortval))
→ is-pd-ref(st[.theinstance.exLog.i].refval,st))

is-pd-ref-def:
is-pd-ref(r,st)
↔ validrefnotnull('docs2store', r, Doclist,st)
∧ validrefnotnull('docs2store', st[r – .docs].refval, mkarraytype(Document),st)
∧ st[r.docs.length].intval = 5
∧ st[r.docs.type].type = mkarraytype(Document)
∧ validrefnotnull('docs2store', st[r.docs.0].refval, IntDoc, st)
∧ validrefnotnull('docs2store', st[r.docs.1].refval, IntDoc, st)
∧ validrefnotnull('docs2store', st[r.docs.2].refval, IntDoc, st)
∧ validrefnotnull('docs2store', st[r.docs.3].refval, IntDoc, st)
∧ validrefnotnull('docs2store', st[r.docs.4].refval, IntDoc, st)
∧ okarray(st[r.docs.0.value],byte_type,8,st)
∧ okarray(st[r.docs.1.value],byte_type,2,st)

```
∧ okarray(st[r.docs.2.value],byte_type,8,st)
∧ okarray(st[r.docs.3.value],byte_type,2,st)
∧ okarray(st[r.docs.4.value)],byte_type,2,st)
```

Hier wird klar ersichtlich, dass die Implementierung in Java wesentlich mehr Fehlerquellen und damit nötige Invarianten erfordert, als dies auf der PROSECCO Ebene oder gar auf Ebene 2 der Fall war. Neben dem gerade beschriebenen `.exLog` ist eine ähnliche Spezifikation auch für die weiteren Felder der `Purse` Klasse, insbesondere auch für die temporären Felder zum Zwischenspeichern von Nachrichten (durch `generateOutmsg`, s. oben) nötig. Diese Definitionen erläutern wir hier nicht mehr. Sie können in der Web Präsentation der Arbeit nachgelesen werden [49]. Insgesamt ergibt sich so ein Prädikat CINV : store, das für den Speicher st obige Wohlgeformtheitsbedingungen fordert.

Neben Invarianten auf dem Java Speicher benötigen wir auch (s. auch allgemeiner Rahmen in 5.3.3) Invarianten auf dem abstrakten PROSECCO Modell. Während die konkrete Invariante eigentlich nur Strukturforderungen an die Pointerstruktur fordert, werden hier nun die funktionalen, durch das Protokoll entstehenden Invarianten definiert. Eine dieser Invarianten ist sogar nötig, um obige Definition eines korrekten `.exLogs` erst vollständig korrekt zu machen:

Wie wir im folgenden Abschnitt sehen werden und wie weiter oben schon informell erläutert wurde, gilt folgender Zusammenhang zwischen abstrakter und konkreter Ebene: der exLogCounter (und hier ist nun die abstrakte Zustandsfunktion gemeint) jedes Agenten entspricht genau dem Wert des Java Feldes `.exLogCounter`. Obige Definition okLog-def fordert nun, dass alle Logeinträge an Indizes bis zu diesem `exLogCounter` wohlgeformt sind. Dies stimmt nur dann, wenn der Java `exLogCounter` auch kleiner ist als die MAX_LOG_LENGTH - sonst würden wir hier Eigenschaften über Einträge fordern, die im Speicher *hinter* dem Ende des Arrays liegen. Wenn wir also als abstrakte Invariante hinzufügen, dass der abstrakte exLogCounter immer kleiner oder gleich der MAX_LOG_LENGTH ist, besteht hier kein Problem mehr.

Hinzu kommen noch etliche andere Bedingungen, die für die Verfeinerungsbeweise benötigt werden. Die aktuelle balance der Karten muss z.B. immer zwischen 0 und 32767 liegen - sonst würde es Überläufe im `short` Wert in der Implementierung geben. Das selbe gilt für die Sequenznummer. Im Weiteren ist z.B. wichtig, dass die aktuellen Transaktionsdetails ppdAuth immer nur Werte enthalten die tatsächlich mit der aktuellen balance der Karten übertragbar sind (also nicht zu groß bzw. zu klein sind). So muss z.B. gelten, dass der zu übertragende Wert, sollte die Karte im Zustand EPR sein, nicht größer ist als die balance der jeweiligen Karte. Spiegelbildlich muss die Summe aus zu übertragendem Wert und aktueller Balance kleiner als 32767 sein, falls der Zustand EPV ist. Schließlich muss auch der geheime Sitzungsschlüssel über alle Karten hinweg den gleichen Wert haben (repräsentiert durch eine Konstante THESESSIONKEY). Ein exemplarischer Ausschnitt aus der abstrakten Invariante ist somit für einen Agenten agent:

SPEZIFIKATION:

$$
\begin{array}{ll}
 & \mathsf{AINV}(\mathsf{agent},\mathsf{name},\ \mathsf{sesskey},\ \mathsf{sequenceNo},\ \mathsf{pbalance},\\
 & \quad\ \mathsf{pstate},\ \mathsf{ppdAuth},\ \mathsf{pexLog},\ \mathsf{exLogCounter},\ \ldots)\\
\leftrightarrow & \quad (\mathsf{cardlet\text{-}agent}(\mathsf{agent})\\
 & \quad \rightarrow \ \ (\ \ \mathsf{pstate}(\mathsf{agent}) = \mathsf{STATE\_EPR}\\
 & \qquad \rightarrow \exists\, \mathsf{i}.\ \ 0 \leq \mathsf{i} \wedge \mathsf{get\text{-}part}(\mathsf{ppdAuth}(\mathsf{agent}),\ 5) = \mathsf{intdoc}(\mathsf{i})\\
 & \qquad\qquad \wedge\ \mathsf{i} \leq \mathsf{pbalance}(\mathsf{agent})\ )\\
 & \qquad \wedge\ (\ \ \mathsf{pstate}(\mathsf{agent}) = \mathsf{STATE\_EPV}\\
 & \qquad \rightarrow \exists\, \mathsf{i}.\ \ 0 \leq \mathsf{i} \wedge \mathsf{get\text{-}part}(\mathsf{ppdAuth}(\mathsf{agent}),\ 5) = \mathsf{intdoc}(\mathsf{i})\\
 & \qquad\qquad \wedge\ \mathsf{pbalance}(\mathsf{agent}) + \mathsf{i} \leq 32767\ )\\
 & \qquad \wedge\ \mathsf{sesskey}(\mathsf{agent}) = \mathsf{THESESSIONKEY}\\
 & \qquad \wedge\ 0 \leq \mathsf{sequenceNo}(\mathsf{agent})\\
 & \qquad \wedge\ \mathsf{sequenceNo}(\mathsf{agent}) \leq 32767\\
 & \qquad \wedge\ 0 \leq \mathsf{pbalance}(\mathsf{agent})\\
 & \qquad \wedge\ \mathsf{pbalance}(\mathsf{agent}) \leq 32767\\
 & \qquad \wedge\ \mathsf{exLogCounter}(\mathsf{agent}) \leq \mathsf{MAX\_LOG\_LENGTH}\\
 & \qquad \wedge\ 0 \leq \mathsf{exLogCounter}(\mathsf{agent})\\
 & \qquad \wedge\ \ldots\ )\\
 & \quad \wedge\ \ldots
\end{array}
$$

12.6.2 Zusammenhang von abstraktem Zustand und Implementierung

Neben den Invarianten für die abstrakte und konkrete Spezifikationsebene ist der inhaltlich wichtigste Teil der Simulationsrelation der Zusammenhang zwischen abstrakter und konkreter Ebene. Informell ist dieser:

- Der abstrakte name jeder abstrakten Purse ist korrekt als Java `IntDoc` im Feld `.name` gespeichert.
- Die abstrakte Sequenznummer sequenceNo entspricht dem Java `short` Wert im Feld `.sequenceNo` umgewandelt in einen int Wert.
- Der abstrakte Sessionkey sesskey(agent) entspricht dem Schlüssel, den man erhält, wenn man das im Java Feld `.key` gespeicherte `byte[ ]` in einen int Wert umwandelt und daraus einen abstrakten Schlüssel erzeugt.
- Die abstrakte pbalance entspricht dem Java `short` Wert im Feld `.balance` umgewandelt in einen int Wert.
- Der abstrakte pstate entspricht dem Java `byte` Wert im Feld `.state` umgewandelt in einen int Wert.
- Die abstrakten Transaktionsdetails ppdAuth entsprechen dem Ergebnis der Umwandlung des Inhalts des Java Feldes `.pd` mittels java2doc.

- Die abstrakten Loginhalte pexLog entsprechen dem Ergebnis der Umwandlung des Inhalte des Arrays im Java Feld `.exlog` mittels java2docs beginnend bei Index 0 und endend bei Index, der durch das Feld `exLogCounter` gegeben ist.
- Der abstrakte exLogCounter entspricht dem Java `short` Wert im Feld `.exLogCounter` umgewandelt in einen int Wert.

Formal gefasst ergeben diese Punkte die folgende Definition. Das Prädikat LMAPPING drückt aus, dass für gegebenen agent die PROSECCO Zustandsfunktionen name bis exLogCounter korrekt im Speicher st repräsentiert sind:

SPEZIFIKATION:

```
   LMAPPING(agent, name, sesskey, sequenceNo, pbalance, pstate,
                ppdAuth, pexLog, exLogCounter, st)
↔    name(agent) =
        intdoc(bytes2int(getbytearray(st[.theinstance.name].refval,st)))
   ∧ sequenceNo(agent) =
        s2i(st[.theinstance.sequenceNo].shortval)
   ∧ sesskey(agent) =
        mkkey(bytes2int(getbytearray(st[.theinstance.key.keyval].refval,st)))
   ∧ pbalance(agent) =
        s2i(st[.theinstance.balance].shortval)
   ∧ pstate(agent) =
        b2i(st[.theinstance.state].byteval)
   ∧ ppdAuth(agent) =
        java2doc(st[.theinstance.pd].refval,st)
   ∧ pexLog(agent) =
        java2docs(getarray(st[.theinstance + .exLog].refval,0,
                           s2i(st[.theinstance + .exLogCounter].shortval),st),st)
   ∧ exLogCounter(agent) =
        s2i(st[.theinstance + .exLogCounter].shortval)
```

Mit einer solchen Defintion des Zusammenhangs von abstrakter Spezifikation und Implementierung (die Teil der Simulationsrelation wird) ist auch trivial klar, dass die Sicherheitseigenschaften der Mondex Anwendung auch für die Implementierungsebene gelten. Die Sicherheitseigenschaften sprechen über die Log Einträge und die balance der Karten. Diese sind in der Implementierung 1:1 gleich. Jeder konkrete Schritt entspricht 1:1 einem abstrakten Schritt, gezeigt durch den Beweis der Verfeinerungskorrektheit unten. Die Sicherheitseigenschaften sind Invarianten auf der abstrakten Ebene und die Verfeinerungstheorie vererbt Invarianten (s. Kap. 3.4.2) auf die konkrete Ebene. Zusammen gelten die Eigenschaften damit auch in der Implementierung.

12.6.3 Definition der Simulationsrelation

Jetzt können wir die Simulationsrelation für Mondex definieren. Sie ergibt sich direkt aus der Anwendung des allgemeinen Spezifikationsrahmens von Seite 56 mittels obiger Prädikate. Für alle purse-cardlet Agenten muss auf konkreter Ebene die Invariante auf dem Speicher gelten. Ebenso muss für alle authentischen purse-cardlet Agenten (dies wird ausgedrückt durch das Prädikat cardlet-agent) das Mapping zur abstrakten Ebene korrekt sein. Schließlich muss auch die abstrakte Invariante gelten. Es ergibt sich somit unter Vernachlässigung einiger hier nicht relevanter Zustandsfunktionen für Angreifer, Terminal und User folgende Definition (Zustandsfunktionen der konkreten Spezifikationsebene tragen zur Unterscheidung den Index c):

SPEZIFIKATION:

$$
\begin{aligned}
& \mathsf{R}(\mathsf{name}, \mathsf{sesskey}, \mathsf{inputs}, \mathsf{sequenceNo}, \mathsf{pbalance}, \\
& \quad \mathsf{pstate}, \mathsf{ppdAuth}, \mathsf{pexLog}, \mathsf{exLogCounter}, \ldots, \\
& \quad \mathsf{inputs}_c, \mathsf{store}_c, \ldots) \\
\leftrightarrow \quad & (\forall.\ \mathsf{agent}.\ \mathsf{cardlet\text{-}agent}(\mathsf{agent}) \rightarrow \mathsf{INV}(\mathsf{store}_c(\mathsf{agent}))) \\
\wedge\ & (\forall.\ \mathsf{agent}.\quad \mathsf{cardlet\text{-}agent}(\mathsf{agent}) \\
& \qquad \rightarrow \mathsf{LMAPPING}(\mathsf{agent}, \mathsf{name}, \mathsf{sesskey}, \mathsf{sequenceNo}, \mathsf{pbalance}, \\
& \qquad\quad \mathsf{pstate}, \mathsf{ppdAuth}, \mathsf{pexLog}, \\
& \qquad\quad \mathsf{exLogCounter}, \mathsf{store}_c(\mathsf{agent}))) \\
\wedge\ & (\forall\ \mathsf{agent}.\ \mathsf{AINV}(\mathsf{agent}, \mathsf{name}, \mathsf{sesskey}, \mathsf{sequenceNo}, \mathsf{pbalance}, \\
& \qquad \mathsf{pstate}, \mathsf{ppdAuth}, \mathsf{pexLog}, \mathsf{exLogCounter}, \ldots)) \\
\wedge\ & \mathsf{inputs} = \mathsf{inputs}_c \\
\wedge\ & \ldots
\end{aligned}
$$

12.7 Verifikation der Fallstudie

12.7.1 Beweisverpflichtungen und Beweistechnik

Für den Beweis der korrekten Verfeinerung der abstrakten Spezifikationsebene verwenden wir die ASM Refinement Theorie. Praktisch macht dies hier eigentlich keinen Unterschied zu Data Refinement, lediglich die Terminierung der Regeln der konkreten Ebene ist in dieser Verfeinerungstheorie direkt in den Haupt-Beweisverpflichtungen enthalten. Es ergibt sich durch Spezifikationsinstantiierung der KIV Spezifikationen mit den Beweisverpflichtungen aus Kap. 3.4.2 und anschließende Simplifikation folgende Beweisverpflichtung (Zustandsfunktionen der abstrakten Ebene sind durch astate abgekürzt, cstate bezeichnet diejenigen der konkreten Ebene):

THEOREM:

Mondex-Step-Correct:
R(astate, cstate)

$\vdash$
$\langle\!|$MONDEX-STEP-CONCRETE(cstate)$|\!\rangle$
  $\exists$ m. $\langle$**loop** MONDEX-STEP(astate) **times** m$\rangle$ R(astate, cstate)

Da wir oben versucht haben, auf der konkreten Ebene in der Implementierung eine direkte Entsprechung der Struktur der abstrakten ASM umzusetzen, können wir in obiger Beweisverpflichtung (nach Beweis der Terminierung von MONDEX-STEP-CONCRETE) den Iterationszähler m immer mit 1 instantiieren. Alle unsere kommutierenden Diagramme für die Protokollschritte sind damit 1:1.

Um den Beweis zu vereinfachen, unterteilen wir nun die Simulationsrelation R in einen Teil, der ausschließlich mit der abstrakten Spezifikation gezeigt werden kann, und einen Teil, für den auch die konkrete Spezifikation nötig ist. Die abstrakte Invariante AINV ist Teil der Simulationsrelation R und kann nur mittels MONDEX-STEP gezeigt werden. Wir definieren nun ein neues Prädikat PJINV, das alle Definitionsbestandteile von R bis auf die Forderung nach AINV enthält. Damit gilt (R(...) $\leftrightarrow$ PJINV(...) $\wedge$ AINV(...).

Anschließend beweisen wir die Gültigkeit der abstrakten Invariante alleine:

THEOREM:

AINV-is-inv:
AINV(astate)
$\vdash$ $\langle\!|$MONDEX-STEP(astate) $|\!\rangle$
  AINV(astate)

Nach Anwendung dieses Theorems verbleibt nur noch zu zeigen:

THEOREM:

Mondex-Step-Correct-weak:
AINV(astate),
PJINV(astate, cstate)
$\vdash$
$\langle\!|$MONDEX-STEP-CONCRETE(cstate)$|\!\rangle$
  $\langle$MONDEX-STEP(astate)$\rangle$ PJINV(astate, cstate)

Anschließend kann mit symbolischer Ausführung von MONDEX-STEP-CONCRETE begonnen werden. Die dadurch entstehende Fallunterscheidung mit der Wahl des nächsten Agentenschrittes (s. Seite 212) ergibt sich auch bei symbolischer Ausführung von MONDEX-STEP. Dadurch können alle Fälle außer dem Fall für PURSE-CARDLET sofort geschlossen werden. Hier ergibt sich analog zu Cindy und nach Wahl eines konkreten purse-cardlet Agenten und Expansion der Definition von PURSE-CARDLET-CONCRETE von Seite 218 für diesen Agenten agent jetzt die eigentliche Korrektheitsaussage für die Implementierung:

THEOREM:

Mondex-Step-Impl-correct:
⟨TOSTORE(agent, inputs; cstore)⟩ cstore = cstore_1,
agent = purse-cardlet(agent .no),
cardlet-agent(agent),
PJINV(astate, cstate),
AINV(cstate),
st = cstore_1(agent)
⊢
⟨⟨st; `Purse.theinstance.step();`⟩⟩
⟨⟨FROMSTORE(agent, st; outdoc, inputs, cstore_1)⟩⟩
⟨⟨if outdoc ≠ ⊥
then SEND(outdoc, ...)⟩⟩
⟨MONDEX-STEP(astate)⟩
PJINV(astate, cstate)

Die Beweisverpflichtung besagt: Nach Mapping der abstrakten Eingabe für den Agenten agent mittels TOSTORE und in einem Zustand beginnend, in dem die Simulationsrelation galt, muss nach Ausführung der Implementierung mittels `step` und anschließendem Versenden mittels SEND sowie Rück-Mapping mittels FROMSTORE ein Protokollschritt auf abstrakter Ebene existieren, nach dem wiederum die Simulationsrelation gilt.

Für den Beweis dieser Aussage (und dies ist der eigentliche Inhalt der Mondex Verifikation) machen wir uns nun die Beweistechnik der wechselseitigen Ausführung von abstrakter und konkreter Ebene zu Nutze. Die Grundidee hierbei ist es, eine sinnvolle Modularisierung der abstrakten und konkreten Ebene anhand der Methodenaufrufe des Java Programms und der Regeln der abstrakten ASM zu finden. Intuitiv kann man sich vorstellen, dass die obige Kommutierungsbedingung noch in weitere kleinere Diagramme zerlegt wird. So wird z.B. für die konkrete Java Methode `checkInDoc` ein Theorem formuliert, das diese mit der abstrakten Regel CHECKINDOC in Beziehung setzt und fordert, dass nach Ausführung von beiden die Simulationsrelation gilt, wenn diese vorher auch galt. Genauso verfährt man dann mit den

weiteren Methoden und Regeln.

Die naive Beweistechnik wäre es, MONDEX-STEP und MONDEX-STEP-CONCRETE bis zu atomaren Programmkonstrukten auszuführen und erst nach Ausführung dieser dann die Gültigkeit von R auf den Nachfolgezuständen wieder zu beweisen. Dies ist auf Grund der immensen Komplexität der durch die symbolische Ausführung beider Ebenen so entstehenden prädikatenlogischen Beweisverpflichtung nicht praktikabel. Statt dessen werden kleinere Diagramme gesucht, die dann anschließend über die Simulationsrelation jeweils miteinander verkettet werden können.

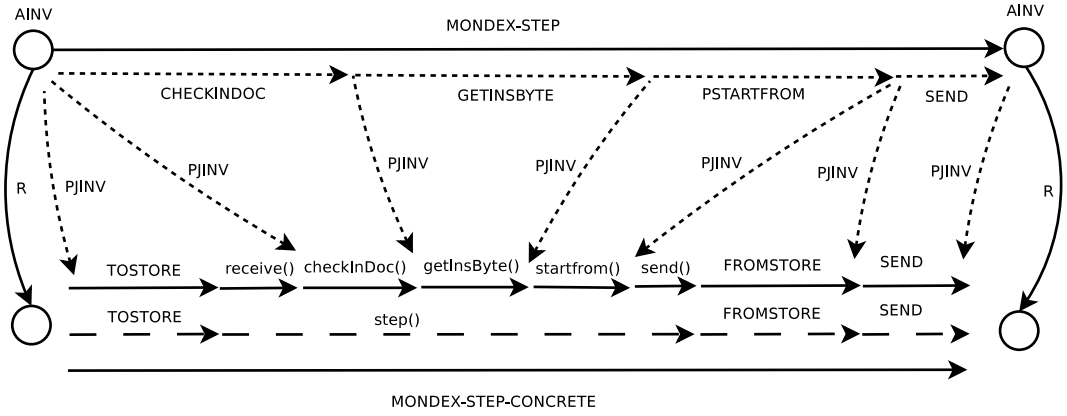


Abbildung 12.5: Weitere Zerlegung bei Mondex am Beispiel von STARTFROM

Betrachten wir Abb. 12.5: Die ursprüngliche Beweisverpflichtung für die Korrektheit eines Schrittes (hier exemplarisch mit STARTFROM gezeigt) ist durch die durchgezogenen Pfeile angegeben. Dies ist genau die Beweisverpflichtung Mondex-Step-Correct vom Beginn dieses Abschnitts. Diese Beweisverpflichtung wird durch symbolische Ausführung auf der konkreten Ebene zur obigen Bedingung Mondex-Step-Impl-correct. In Abb. 12.5 entspricht dies den gestrichelten Pfeilen unten und der durchgezogenen Pfeile oben. Hier ist allerdings noch keine wirkliche Modularisierung des Beweises erkennbar. Um diese zu erreichen, verwenden wir die kleineren, durch gepunktete Pfeile gezeigten Diagrammen in Abb. 12.5:

- Die Ausführung von TOSTORE und `receive` auf konkreter Ebene stellt dem Java Programm lediglich die Eingabe zur Verfügung, der Zustand ändert sich noch nicht. Damit erhalten wir bezüglich PJINV ein 0:2 Diagramm.

- Die Ausführung von `checkInDoc()`, `getInsByte` und `startfrom()` entspricht jeweils 1:1 bezüglich PJINV den abstrakten Gegenstücken CHECKINDOC, GETINSBYTE und PSTARTFROM.

- Die Ausführung von `send()` und FROMSTORE auf konkreter Ebene extrahiert lediglich die Ausgabe des Java Programms aus dem Speicher, der Zustand ändert sich wieder nicht. Damit erhalten wir bezüglich PJINV wieder ein 0:2 Diagramm.

- Das konkrete SEND entspricht genau 1:1 dem abstrakten SEND.

Über die abgeschwächte Simulationsrelation PJINV können diese Diagramme nun verbunden werden und zusätzlich mit AINV die Simulationsrelation R auf dem Nachfolgezustand gezeigt werden.

Exemplarisch wollen wir diese Beweistechnik nun auch noch durch entsprechende Lemmata illustrieren. Dabei werden wir auch feststellen, dass noch einige Zusatzeigenschaften neben dem eben beschriebenen generellen Schema benötigt werden.

Zunächst betrachten wir das Theorem für TOSTORE und `receive`. Hier ist zu beachten, dass dieses Theorem schon recht genau der bereits aus den vorangegangenen Kapiteln 6.9, 8.7 und 11.3.2 bekannten und auch für Cindy verwendeten Eigenschaft TOSTORE-receive entspricht. Hinzu kommt hier lediglich noch die Bedingung PJINV.

THEOREM:

TOSTORE-receive-mondex:
PJINV(cstate, astate),
INV(cstore(agent)),
cardlet-agent(agent),
inputs(agent)(1) $\neq$ [],
poolrestriction(inputs(agent)(1) .first)
$\langle$TOSTORE(agent, inputs; cstore)$\rangle$ (cstore = cstore_1)
$\text{cstore}_1(\text{agent})_1 = \text{st}_1 \vdash$
  $\langle \text{st}_1$; r_1 = `Purse.theinstance.comm.receive();`$\rangle$
    (java2doc(r_1, st_1) = inputs(agent)(1) .first
      $\wedge$ validDoc(r_1, st_1)
      $\wedge$ validref('docs2store', r_1, Document, st_1)
      $\wedge$ SINV(cstore[agent ; st_1),
      $\wedge$ PJINV(astate, cstate[cstore; cstore_1[agent ; st_1]])
    );

Die Formel besagt: Galt vor Ausführung von TOSTORE und `receive` bereits PJINV, so gilt dies auch danach, wenn die Komponente cstore des konkreten Zustands cstate auf den sich so ergebenden Speicher st_1 gesetzt wird. Bei genauerer Betrachtung des Theorems fällt bereits die erste nötige zusätzliche Eigenschaft für den Verfeinerungsbeweis auf: In der Nachbedingung findet sich das Prädikat SINV (für *strong Invariant*). Dieses ergänzt die bereits oben gezeigte PJINV für den Java Speicher. Die zusätzlich nötige Eigenschaft bezieht sich auf die Felder, die zur zwischenzeitlichen Speicherung der Eingabe und Ausgabe sowie zur Feststellung der Ausführung einer `receive` Operation von TOSTORE gesetzt werden. Hier muss z.B. Typkorrektheit und Enthaltensein der Felder im Speicher gefordert werden. Die weiteren Nachbedingungen (wie z.B. die Korrektheit der entstandenen Eingabe java2doc(r_1, st_1) = inputs(agent)(1) .first) müssen ebenfalls ggf. für weitere Lemmata als Vorbedingungen weiter verwendet werden. Hier zeigt sich bereits, dass ausschließlich die Information, die in PJINV

enthalten ist, nicht für einen korrekten Beweis ausreicht.

Nach `receive` wird im weiteren Kontrollfluss die `checkInDoc` Methode ausgeführt. Deren Korrektheitseigenschaft nach dem obigen generellen Schema ergibt sich wie folgt:

THEOREM:

checkindoc-correct:
PJINV(cstate, astate),
INV(cstore(agent)),
AINV(astate),
cardlet-agent(agent),
validDoc(r_1, st_1),
validref('docs2store', r_1, Document, st_1),
java2doc(r_1, st_1) = doc,
poolrestriction(doc), cstore(agent) = st_1
⊢
⟨st_1 ; r_2 = `Purse.theinstance.checkIndoc(`r_1`);`⟩
⟨checkInDoc(agent, sesskey; doc) ⟩
(java2doc(r_2, st_1) = doc
∧ validDoc(r_1, st_1)
∧ validref('docs2store', r_1, Document, st_1)
∧ SINV(cstore[agent ; st_1),
∧ PJINV(astate, cstate[cstore; $cstore_1$[agent ; st_1]])
);

Das Theorem besagt: Falls die Invarianten gelten und ein Speicher st_1 die Repräsentation eines wohlgeformten Dokumentes doc enthält und im Anschluss sowohl konkrete als auch abstrakte Ebene CHECKINDOC bzw. `checkInDoc` ausführen, so gilt die Gleichheit auch für die Ausgabe und die Invarianten gelten wieder.

Als letztes Beispiel betrachten wir noch exemplarisch die Korrektheitseigenschaft für PSTART-FROM:

THEOREM:

startfrom-correct:
cardlet-agent(agent),
SINV(st)
st = cstore(agent), AINV(astate),
PJINV(astate, cstate),
...,

$$
\begin{array}{l}
\vdash \\
\langle \mathsf{st};\ \mathsf{r}_0 \texttt{ = Purse.instance.startFrom(}\mathsf{r}_2\texttt{);}\rangle \\
\quad \langle \mathsf{PSTARTFROM(doc, ...; outdoc, ...)}\rangle \\
\quad\ \ (\quad \mathsf{SINV(st)} \\
\qquad \wedge\ \mathsf{PJINV(\ \underline{astate},\ \underline{cstate}[cstore;\ cstore}_1\mathsf{[agent\ ;\ st}_1\mathsf{]]]}) \\
\qquad \wedge\ \mathsf{java2doc}(\mathsf{r}_0,\ \mathsf{st}) = \mathsf{outdoc} \\
\qquad \wedge\ \mathsf{validDoc}(\mathsf{r}_0,\ \mathsf{st}) \\
\qquad \wedge\ \mathsf{validref}(\text{`docs2store'},\ \mathsf{r}_0,\ \mathsf{Document},\ \mathsf{st}))\ ;
\end{array}
$$

Die Nachbedingung obiger Formel besagt, dass die Invarianten nach Ausführung von PSTARTFROM und `startFrom` wieder gelten und die abstrakte Ausgabe outdoc im konkreten Speicher gleich ist. Die Vorbedingung ist hier nun (bis auf die üblichen Invarianten) offen gelassen. Dies liegt daran, dass sich insbesondere bei PSTARTFROM nun eine der Komplexitäten der Mondex Verifikation zeigt: Es werden wieder Vorbedingungen benötigt, die über PJINV hinausgehen. Das Problem hierbei ist allerdings, dass diese Vorbedingungen sich nur aus dem Ausführungsergebnis vorangegangener Operationen herleiten lassen. Ein einfacheres Beispiel hierfür ist die Notwendigkeit, für eine erfolgreiche Ausführung von `startFrom` wissen zu müssen, dass zuvor `checkInDoc` die korrekte Struktur der Eingabe überprüft hat. Mit diesen Annahmen ist `startFrom` programmiert worden. Erinnern wir uns an die obige Implementierung, so fällt z.B. auf, dass gleich die erste Zeile

```
1 Document dmsgna = checkName(indoc.getPart((short)1));
```

eine `NullpointerException` liefern würde, wäre indoc `null`. Durch die Überprüfung durch `checkInDoc` ist dies allerdings ausgeschlossen. Die eine Möglichkeit wäre nun, in der Vorbedingung von obigem Theorem startfrom-correct einfach alle Eigenschaften, die `checkInDoc` sicherstellt, prädikatenlogisch aufzuführen. Dadurch wäre allerdings wenig gewonnen, würde man doch so im wesentlichen das selbe komplexe Ergebnis erhalten, als hätte man `checkInDoc` gleich komplett symbolisch ausgeführt. Eine bessere Möglichkeit ist es daher, einfach die Tatsache, dass das *abstrakte* checkInDoc ein korrektes Ergebnis geliefert hat, in die Vorbedingung aufzunehmen. Im Anschluss kann dann aus dieser Tatsache und dem Mapping zwischen abstrakten und konkreten Dokumenten einfacher gefolgert werden, dass bestimmte Eigenschaften gelten (wie hier z.B. die Ungleichheit zu `null`). Wir müssen also Eigenschaften wie die folgenden in die Vorbedingung von startfrom-correct aufnehmen:

$$
\begin{array}{l}
\mathsf{is\text{-}doclist}(\mathsf{doc}_1), \\
\mathsf{poolrestriction}(\mathsf{doc}_1), \\
\langle \mathsf{checkInDoc}(\mathsf{agent},\ \mathsf{sesskey1};\ \mathsf{doc}_1)\rangle\ (\mathsf{doc}_1 = \mathsf{doc}_0), \\
\mathsf{get\text{-}part}(\mathsf{doc}_0,\ 1) = \mathsf{intdoc(INS\_START\_FROM)}, \\
\mathsf{doc} = \mathsf{get\text{-}part}(\mathsf{doc}_0,\ 2), \\
\mathsf{java2doc}(\mathsf{r}_2,\ \mathsf{st}) = \mathsf{doc} \\
\mathsf{validDoc2}(\mathsf{r}_2,\ \mathsf{st}),
\end{array}
$$

validref('docs2store', r_2, Document, st)

Diese besagen: Eine Doclist doc_1 wurde durch checkInDoc zu doc_0 transformiert. Bei diesem Ergebnis handelt es sich nun um eine STARTFROM Nachricht und der Speicher enthält an Referenz r_2 eine wohlgeformte Entsprechung genau des Datenteils (get-part(doc_0, 2)) dieser Nachricht.

Leider war dies nicht immer so möglich wie eben beschrieben. Eine weitere Hauptkomplexität bei der Mondex Verifikation war die Tatsache, dass manchmal recht spät im Beweis Eigenschaften benötigt wurden, die durch frühe Methodenaufrufe sichergestellt werden - allerdings zu dem frühen Zeitpunkt nicht sofort erkennbar waren. Dies führte oft zu mehrmaligen Iterationen im Gesamtbeweis, da frühe Lemmata geändert werden mussten. Ein Beispiel ist wiederum STARTFROM. In der Implementierung von `startFrom` wird die Methode `mkpd` aufgerufen. Diese Methode soll, wie oben auf Seite 224 bereits gezeigt, die übergebenen Parameter (die beiden Namen als `byte[ ]` , den zu übertragenden Wert und die beiden Sequenznummern als `short`) in das Feld `.pd` kopieren. Nun ist einer der beiden `byte[ ]` Referenzparameter der Wert der Feldes `.name` von `Purse`, der andere ist eine Referenz aus dem Eingabedokument `indoc`. Nun kann es mit den bisherigen Vorbedingungen sein, dass die Speicherstruktur wie in Abb. 12.6 dargestellt aufgebaut ist. Die Eingabe indoc enthält einen Zeiger auf den `byte[ ]` Wert im ersten Teil der paydetails in `.pd` von `Purse`. Damit würde die Implementierung von `mkpd` (die ebenfalls schematisch in Abb. 12.6 gezeigt ist) zunächst mit der ersten Anweisung den eigenen NAME in den ersten Slot der PD_NAME1 der paydetails kopieren. Damit würde aber der dort auf Grund des Sharings mit der Eingabenachricht ebenfalls gespeicherte SF_NAME überschrieben werden und die dritte Kopieroperation hätte nicht das intendierte Verhalten.

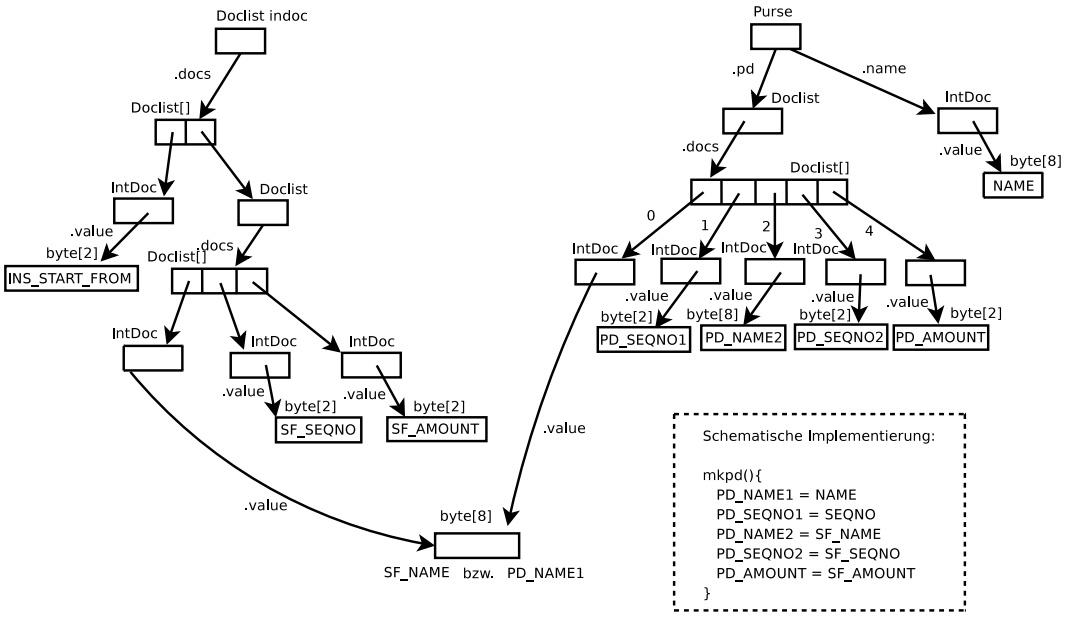


Abbildung 12.6: Fehlerhafte Speicherstruktur in Mondex

Die beschriebene Situation kann in der Realität nicht vorkommen, da der Pool disjunkt zu den Speicherinhalten der Implementierung sein muss (dies haben wir auch schon in Kap. 8.3.6 beschrieben). Dennoch sind unsere bisherigen Vorbedingungen in startfrom-correct nicht stark genug, um die Situation in Abb. 12.6 auszuschließen. Als zusätzliche Vorbedingung muss daher hinzugenommen werden:

disjoint('MONDEX-javaprog', st[.theinstance] .refval + r_2, st)

Diese besagt: alle Referenzen, die von st[.theinstance] erreichbar sind, sind disjunkt zu allen Referenzen, die von der Eingabe r_2 erreichbar sind.

Leider ist diese Vorbedingung nicht ohne weiteres zu diesem späten Zeitpunkt im Beweis herleitbar. Sie folgt nur aus einer weiteren Eigenschaft der `receive` Methode, die ganz zu Beginn des Beweises mittels tostore-receive-mondex behandelt wurde (s. Anfang dieses Abschnittes). Nun muss also zurückgegangen werden und das anfängliche Theorem um diese Disjunktheit in der Nachbedingung erweitert werden.

Auf die beschriebene Art und Weise wurden die Protokollschritte STARTFROM, ACK, GETDATA GETBALANCE und GETSTATE bewiesen. Bezogen auf den gesamten Quellcode der `Purse` Klasse erreichen wir damit 85 Prozent verifizierten Quellcode der Mondex Anwendung.

12.7.2 Fazit und Statistiken

Es zeigt sich, dass die relativ hohe Komplexität der Mondex Anwendung dazu führt, dass die Beweise nur schwer geradlinig zu führen sind und einige Iterationen nötig sind. Ähnliche Eigenschaften wie die zum Ende des letzten Abschnitts beschriebenen wurden insbesondere auch für den `abort` Methodenaufruf benötigt.

Neben den gerade beschriebenen Problemen zeigte sich auch, dass die Modularisierung essentiell für die Beweise ist. Ein früher Beweisversuch ohne die Modularisierung endete ergebnislos mit viel zu komplexen Formeln. Die symbolische Ausführung von Java Programmen erzeugt und fordert wesentlich mehr Vorbedingungen als dies z.B. bei DL Programmen oder ASMs der Fall ist. Viele benötigte Eigenschaften auf dem Speicher (wie z.B. die oben benötigte disjoint Eigenschaft für Referenzen) waren in der bisherigen Bibliothek nicht enthalten und mussten erst spezifiziert und mit passenden Simplifikationsregeln komplettiert werden.

Allgemein zeigte sich, dass Modifikationen auf dem Speicher in komplexen Zeigerstrukturen noch der weiteren Ausarbeitung bezüglich dem erreichbaren Beweissupport und der Automatisierbarkeit der Beweise bedürfen. Insbesondere destruktive Veränderungen sind schwierig zu handhaben. Die fehlende Möglichkeit zur Speicherallokation auf Chipkarten führt ferner dazu, dass alle Strukturen (auch z.B. temporäre Zwischenstrukturen) bereits initial komplett und korrekt angelegt werden müssen. Dies vergrößert die Invariante stark. Die gesamte Simulationsrelation beinhaltet daher 87 verschiedene Eigenschaften. Allein 58 davon sprechen nur über Inhalte des Java Speichers von Mondex. Alle diese Eigenschaften zerfallen wieder in weitere Teilformeln.

Generell ist das Handling von Zeigerstrukturen in KIV noch verbesserungswürdig. Ein erster

Schritt in diese Richtung wurde dabei durch eine alternative Spezifikationsmethodik für den Speicher bereits durchgeführt. Es wurde hierbei versucht, eine Axiomatisierung anzugeben, die ausschließlich typkorrekte Speicherinhalte zulässt. Dies würde Vorbedingungen wie validref einsparen. Die Tauglichkeit dieser Ansätze lässt sich heute allerdings noch nicht verlässlich feststellen.

Rückblickend betrachtet lässt sich sagen, dass die Verifikation nicht ein einzelnes großes und inhaltlich schwieriges Problem beinhaltete, sondern vielmehr durch die Summe vieler einzelner kleinerer Komplikationen zu der jetzigen Komplexität führte.

Die hohe Komplexität der Fallstudie zeigt sich auch in den Statistiken. Die Mondex Fallstudie ist sicherlich bezüglich dieser Zahlen eine der bisher größten mit dem KIV System durchgeführten Fallstudien. Die Verifikation der Mondex Fallstudie benötigte (jeweils ohne Bibliotheken):

- 193 Axiome
- 97 verschiedene Sortendefinitionen
- 1702 Theoreme
- 87162 Beweisschritte insgesamt
- 30147 Benutzerinteraktionen (Automatisierungsgrad: 65 Prozent)

Bezieht man sämtliche Spezifikationen bis hin zur Basisbibliothek für z.B. natürliche Zahlen in die Statistik mit ein, baut die Mondex Spezifikation auf

- 2.525 Axiomen
- 12.794 Theoremen
- und Beweisen im Umfang von 195.796 Beweisschritten (77.285 Interaktionen) auf.

Im Vergleich zur Cindy Anwendung erreicht die Mondex Anwendung damit die fast 10-fache Komplexität. Die Zeit, die für die Mondex Verifikation benötigt wurde, ist schwer zu messen. Sie liegt sicherlich bei ca. 6 Personenmonaten durch entsprechend geschultes Personal. Schließt man die Verifikation aller vier Abstraktionsebenen der Mondex Applikation mit ein, so liegt der Gesamtaufwand bei etwa 12 Personenmonaten. Die äußerst hohen Garantien, die man mit dem hier vorgestellten Ansatz bezüglich der Verlässlichkeit und Sicherheit der Anwendung letztendlich gewinnt, rechtfertigen diesen Aufwand in sicherheitsrelevanten und hochkritischen Bereichen durchaus.

12.8 Vergleich zum Ansatz des KeY Projektes

Während viele Arbeitsgruppen ihre Verifikationswerkzeuge auf die Mondex Fallstudie anwendeten (ein Überblick wurde bereits in Kap. 12.3.1 gegeben), gibt es eine Arbeit, deren Anspruch sehr ähnlich zu den Konzepten dieser Arbeit ist. Mit dem KeY System [19] wurde bereits in

Kap. 4.2 ein weiteres eng zu KIV verwandtes Verifikationswerkzeug vorgestellt. In [165] und [182] zeigen Tonin und Schmitt eine Verifikation der Mondex Fallstudie mittels dieses Systems. In diesem Abschnitt sollen die Unterschiede zwischen diesen Arbeiten und den hier vorgestellten Arbeiten erläutert werden.

Die Autoren der KeY Verifikation implementieren die Mondex Anwendung in JavaCard. Anschließend wird jede so entstandene Methode mittels JML [115] annotiert. Aus den Annotationen werden dann Beweisverpflichtungen in der dynamischen Logik des KeY Systems erzeugt (Unterschiede zur Logik des KIV Systems wurden bereits in Kap. 4 vorgestellt). Die Beweisverpflichtungen werden dann zum größten Teil vollautomatisch mit dem System mit einem eigenen Kalkül für JavaCard verifiziert.

Die Annotationen geben das jeweils intendierte Verhalten der Methoden vor. So muss z.B. für eine `val` Methode für den VAL Protokollschritt gelten, dass falls der Zustand vorher EPV war, er danach IDLE ist (in [165] wird zusätzlich noch ENDT als Zustand nach einer Transaktion eingeführt) und die `balance` entsprechend korrekt um den Wert der `paydetails` erhöht wurde. Alle Annotationen beziehen sich - wie in JML üblich - auf z.B. Felder der implementierenden Klasse. Da damit eine Spezifikation auf der Ebene des Quellcodes stattfindet, ergibt sich nicht die Möglichkeit zur Abstraktion. Dies ist bereits der erste und wohl auch grundlegendste Unterschied zur hier vorgestellten Arbeit: Es findet keine abstrakte Spezifikation und keine Verfeinerung statt. Die Möglichkeit zur abstrakten Spezifikation mit z.B. abstrakten Datentypen oder Konzepten wie ASMs existiert derzeit im KeY System nicht.

Damit muss in KeY die Sicherheit der Mondex Anwendung direkt auf der Quellcode Ebene bewiesen werden. Dies wird von den Autoren der genannten Papiere auch durchgeführt. Dabei ergeben sich folgende Punkte: Die KeY Implementierung verwendet keinerlei Kryptographie. Statt dessen gehen die Autoren davon aus, dass die Eingabenachrichten (die einfache APDUs, also Sequenzen von Bytes sind) nicht gefälscht werden können. Es findet keine Transformation in Datentypen ähnlich zu den `Document` Typen dieser Arbeit statt. Die Annahme der Unfälschbarkeit wurde dabei wohl aus der Original Fallstudie [174] übernommen. Auf dem dortigen abstrakten Spezifikationslevel ist die Annahme noch tragbar, in einer Implementierung ist sie zumindest fragwürdig. Dazu antwortet die vorgestellte Implementierung auf jede Eingabenachricht lediglich mit einem Flag für "OK" bzw. "Fehler". Insbesondere generieren die Chipkarten damit selbst keine Nachrichten, sondern es wird davon ausgegangen, dass ein Terminal existiert, dass die richtigen Nachrichten zur richtigen Zeit erzeugt. Dies steht im Widerspruch zu den Anforderungen der Fallstudie aus [174]. Diese besagt: "All security measures have to be implemented on the card" und "Once released into the field, each purse is on its own: it has to ensure the security of all its transactions without recourse to a central controller.". Eine Chipkarte, die keine Kryptographie verwendet und auf jede Eingabenachricht nur mit einem Erfolgsflag antwortet, kann sehr einfach gefälscht werden. Damit wäre in der Realität mit der Implementierung des KeY Projektes die Generierung von Geld mit gefälschten Karten trivial möglich. Unter realen Bedingungen können also die Sicherheitseigenschaften der Mondex Anwendung nicht in dieser Implementierung gelten.

Mit den genannten Einschränkungen gerät die Implementierung wesentlich einfacher als die in der vorliegenden Arbeit vorgestellte. Die Autoren verwenden keine komplexen Datentypen, sogar die Namen der Purses werden lediglich durch einen `short` Wert angegeben (was die Anzahl von Karten auf maximal 32767 beschränkt). Lediglich für die `PayDetails` wird eine eigene Klasse verwendet, das Exception Log ist ein Array solcher Objekte.

Schließlich geben die Autoren eine Verifikation der Sicherheitseigenschaften der Mondex Anwendung an - unter obigen Annahmen. Dabei ist folgendes zu beachten: Der mit JML verfolgte Ansatz, eine Spezifikation mittels Design-by-Contract auf Methodenebene vorzugeben, erlaubt es nicht, einen globalen Blick auf Kommunikationsverhalten zwischen mehreren Komponenten zu haben. Die Sicht ist auf ein Programm in seiner virtuellen Maschine beschränkt. Damit kann eine Aussage wie "Über alle Protokollschritte hinweg wird die Summe der Kontostände aller Chipkarten nicht größer" nicht formuliert werden.

Ebenso kann keine Kausalität zwischen mehreren Methodenaufrufen ausgedrückt werden. Ein Beispiel: Ein Aufruf der Methode `select` muss in JavaCard zu Beginn jeder neuen Verbindung zu einer Chipkarte erfolgen, erst danach kommt `process` mit der eigentlichen Protokollfunktionalität. Übernimmt nun `select` wesentliche Funktionalität (bei der genannten Mondex Implementierung das Logging vorhergegangener fehlgeschlagener Transaktionen), so kann nicht beweisen werden, dass diese auch tatsächlich vor Beginn einer neuen Kommunikation ausgeführt wird. Anders ausgedrückt: Es kann nicht bewiesen werden, dass die Mondex Karten tatsächlich fehlgeschlagene Transaktionen korrekt loggen, da nicht ausgedrückt werden kann, dass der entsprechende Aufruf immer (ausgelöst durch das Kartenbetriebssystem) vor einem neuen Protokoll erfolgt.

Statt dessen wird folgendes gezeigt: es wird angenommen, dass in jedem Zustand, in dem eine Purse nicht im Zustand IDLE ist, eine Partnerbörse existiert (eine zweite Instanz der `Purse` Klasse), die sich in einem zum Protokoll passenden Zustand befindet. In diesen Zuständen wird mittels einer Beweisverpflichtung für eine künstliche Methode `showProperties` mit leerem Methodenrumpf gezeigt, dass der bereits gebuchte Wert auf der Partnerbörse und der eigene bereits gebuchte Wert nicht im Widerspruch zu der Tatsache befinden, dass Geld verloren gehen könnte. Alternativ ausgedrückt: Wenn eine Partnerbörse existiert, die sich gemäß dem Protokoll verhalten hat, so ist die Summe des aktuell auf der einen Seite bereits abgebuchten (negativ verrechnet) und dem auf der anderen Seite gutgeschriebenen Betrag plus (sollte die Transaktion derzeit in einem kritischen Zustand sein) dem aktuellen Transaktionswert gleich null. Die Autoren argumentieren, dass dies die Sicherheitseigenschaft ALL VALUE ACCOUNTED sicherstellt, die besagt, dass die Summe aller paarweise existenten Logeinträge und die Summe der balances der Karten konstant bleibe. Diese Argumentation wird allerdings durch folgenden Fehler in der Implementierung in Frage gestellt: Die Autoren geben eine Implementierung an, in der das Loggen einer fehlerhaften Transaktion durch einfaches Eintragen der Referenz der aktuellen `PayDetails` an die nächste freie Stelle im `exLog` Array realisiert wird. Dies ist falsch, denn dadurch zeigen nach mehrfachen Logging alle `exLog` Einträge auf die gleiche Referenz und schlimmer noch ändert das Starten einer neuen Transaktion per Seiteneffekt die Logeinträge. Da die Sicherheitseigenschaft ALL VALUE ACCOUNTED besagt, dass eben diese Log Einträge die lost Komponenten richtig implementieren, kann die Implementierung die Sicherheitseigenschaft nicht erfüllen. Den Autoren der KeY Verifikation ist dieser Fehler nicht aufgefallen, da aus genannten Gründen die Eigenschaft nicht direkt verifiziert werden konnte und auch keine weiteren Eigenschaften über das `exLog` gezeigt wurden.

Zusammenfassend lässt sich folgendes festhalten: Unter Berücksichtigung der von den Autoren getroffenen Annahmen kann die Implementierung des KeY Projektes nicht mit der hier vorgestellten Implementierung verglichen werden. Zwar implementieren beide die gleiche Fallstudie, die Annahmen an die Umgebung in der KeY Verifikation und das Fehlen einer abstrakten Ebene und einer Verfeinerungstheorie machen die Ansätze allerdings nicht direkt vergleichbar. Es

besteht die Vermutung, dass eine Verfeinerungstheorie oben genannten Quellcodefehler gefunden hätte, da mit einer solchen die Gleichheit der Logs auf abstrakter und konkreter Ebene gezeigt hätte werden müssen, was mit dem Programmierfehler nicht möglich gewesen wäre. Die KeY Verifikation ist nach Meinung des Autors dieser Arbeit daher nicht als realistische Implementierung anzusehen - allein schon wegen des Fehlens kryptographischer Operationen und der Generierung von Nachrichten durch das Terminal. Vielmehr handelt es sich mehr um eine weitere Spezifikation der Mondex Fallstudie, in der die Java Programmiersprache als Spezifikationssprache verwendet wurde. Hervorzuheben ist dagegen insbesondere der hohe Automatisierungsgrad der Verifikation in KeY.

Kapitel 13

Zusammenfassung

13.1 Erreichte Ziele und Erfahrungen

In der Arbeit wurde eine Verfeinerungsmethodik für Kommunikationsprotokolle in Java vorgestellt. Mit den erreichten Ergebnissen ist es möglich, lauffähige Implementierungen für kommunizierende sicherheitskritische Systeme z.B. aus E-Commerce Szenarien formal auf Korrektheit und den Erhalt von Sicherheitseigenschaften zu überprüfen. Die vorliegende Arbeit ist - nach bestem Wissen - die Erste, die eine Abstraktion vom Code Level zu einem High-Level Spezifikationsframework für Kommunikationsprotokolle erreicht und daher für eine schrittweise und modulare Verifikation von Protokollen geeignet ist.

Das dazu ausgearbeitete Verfeinerungsframework ist generisch für Protokollimplementierungen in Java einsetzbar. Es baut dabei auf der Verfeinerungstheorie des Data Refinements und einer speziellen Variante der Verfeinerungstheorie für Abstract State Machines auf. Auf der abstrakten Ebene ist die bereits erprobte Prosecco Methodik zur Spezifikation und Verifikation von abstrakten Protokollbeschreibungen Grundlage der Arbeit. Durch eine korrekte Verfeinerung dieser Prosecco Spezifikationen mittels der hier vorgestellten Techniken übertragen sich die Sicherheitseigenschaften, die getrennt auf dem abstrakten Level verifiziert werden können, auf die Implementierung.

Wesentlich ist dabei eine Definition einer Datentypumwandlung zwischen Implementierung und abstrakter Spezifikation. Dabei darf allerdings nicht die reale Umgebung einer Protokollimplementierung außer Acht gelassen werden. Es zeigt sich, dass die bloße Betrachtung der abstrakten Spezifikation und deren Angreifermöglichkeiten in der Realität nicht ausreicht. Statt dessen müssen auch weitere, nur durch einen Angreifer auf dem Code Level vorkommende Bedrohungen betrachtet werden. Diese konnten in dieser Arbeit formal charakterisiert werden und insbesondere konnte auch durch eine zweistufige Herangehensweise eine generische Behandlung zusätzlicher Angriffe in der Implementierung erreicht werden. Dazu wurde exemplarisch eine Datenübertragungsschicht definiert und eine entsprechende Implementierung für diese angegeben. Die Implementierung wurde als korrekt bezüglich ihrer Spezifikation nachgewiesen und kann damit in allen weiteren Anwendungen wieder verwendet werden.

Die Methodik wurde in dieser Arbeit auf zwei große Fallstudien angewandt, eine J2ME Ticketing Applikation für Mobiltelefone sowie auf eine JavaCard Implementierung der Mondex

Geldkarte. Beide Fallstudien konnten mit der vorgestellten Technik verifiziert werden. Die Verifikation der Mondex Anwendung wurde dabei bereits auf einem viel abstrakteren Level im Jahr 2006 als eine der Grand Challenges in Verifikation [99] ausgerufen. Die hier vorgestellte Verifikation erweitert diese Challenge noch um ein tatsächliche korrekte Implementierung in JavaCard. Ein großer Teil der dabei erarbeiteten Ergebnisse ist durch die generische Verfeinerungsbibliothek für andere Fallstudien weiterverwendbar. Das Ergebnis der Mondex Verifikation war der umfangreichste Beitrag zum Verified Software Repository aller beteiligten internationalen Gruppen.

Darüber hinaus haben sich im Lauf der Arbeit große Erweiterungen am Java Beweissupport im KIV System ergeben. Insbesondere wurden durch die Arbeit große Basisspezifikationen zu Eigenschaften von Zeigerstrukturen und anderen Speicherinhalten angelegt. Die gesamte Herangehensweise in der Arbeit setzte sehr stark auf die Objektorientierung der Java Programmiersprache. Ein großer Bestandteil dieser Spezifikationen beschäftigte sich daher mit Typkorrektheit von Strukturen, die vollständig formal charakterisiert wurde. Darüber hinaus konnten auch viele interessante Eigenschaften von Strukturen wie z.B. Zyklizität, Pfadexistenz oder Vorliegen von Pointer-Sharing spezifiziert und eingesetzt werden. Zu diesen Themen lag zu Beginn der Arbeit noch nichts im KIV System vor. Die entstandenen Bibliotheken sind ebenfalls einfach in anderen Fallstudien wiederverwendbar.

Dennoch war die hier vorgestellte Verifikation teils mit hohen Aufwänden verbunden, die zum Einen naturgemäß durch den Einsatz eines interaktiven Verifikationswerkzeugs und zum Anderen durch die beschriebene Notwendigkeit zur Definition vieler zuvor nicht behandelter Eigenschaften über Java Programme entstanden. Nichts desto weniger gewinnt man durch den Einsatz eines interaktiven Tools wie KIV überhaupt erst die Möglichkeit, mit solch komplexen Implementierungen wie den hier beschriebenen formal umgehen zu können. Derzeit sind vollautomatische Werkzeuge nicht in der Lage, abstrakte Spezifikationen und eine Implementierungsebene zusammen mittels eines Ansatzes wie dem hier vorgestellten zur Verfeinerungskorrektheit verifizieren zu können.

Ein Teil des teils hohen Aufwandes zur Verifikation der Fallstudien kommt dabei auch von der Art und Weise der Definition des Java Speichers. Hier wurden inzwischen auch weitere Arbeiten aufgenommen und es wurde eine alternative Axiomatisierung des Java Heaps durchgeführt, die nur typkorrekte Strukturen innerhalb des Speichers zulässt. Damit entfallen dann einige der komplexeren Argumentationen zur Typkorrektheit von Speicherstrukturen. Die Praxistauglichkeit dieses Ansatzes muss sich allerdings erst noch in weiteren Fallstudien zeigen.

Neben dieser Änderung des Java Speichermodells wurde im Laufe der Arbeit auch mit der Anbindung von automatischen statischen Codeanalysetechniken an das KIV System experimentiert. Viele Eigenschaften von Java Programmen, wie etwa Aussagen über die von einem Programmstück potentiell veränderten und in jedem Fall gleich belassenen Speicherteile, können mittels statischer Analyse ebenso genau wie mit interaktiven System und insbesondere automatisch verifiziert werden. Dadurch ergibt sich nochmals eine große Vereinfachung. Die weitere Ausarbeitung im Bereich der Anbindung vollautomatischer Systeme für die Verifikation von Teileigenschaften ist sicherlich ein überaus wichtiges Thema für weitere Entwicklungen.

Darüber hinaus konnten in der hier vorgestellten Arbeit viele Erkenntnisse gewonnen werden, welche Dinge bei der Verifikation von Java Programmen im Allgemeinen und bei Kommunikationsprotokoll Implementierungen im Besonderen zu beachten sind. Insbesondere laufen

derzeit bereits in einem anderen Dissertationsvorhaben Bemühungen, Implementierungen wie die hier verifizierten mittels Model Driven Development teils automatisch und dann bereits per Konstruktion korrekt aus Modellen erzeugen zu können. Insbesondere eine verlässliche Korrektheitsaussage bezüglich der Transformation der Modelle zu Code bzw. zu abstrakten Spezifikationen muss sich dabei wieder mit ähnlichen Punkten beschäftigen, wie dies in dieser Arbeit dargelegt wurde. Damit kann die Arbeit auch als Grundlage für weitere Entwicklungen dienen.

Neben der reinen Verifikation von Kommunikationsprotokollen und ihren Implementierungen ist es mit der vorgestellten Methodik im Allgemeinen auch möglich, sämtliche Arten von reaktiven Systemen mit Input, Output und Zustandsübergangsverhalten zu verifizieren. Anzupassen sind dabei natürlich die Datentypen, die Definitionen zum Datentypmapping und zur Spezifikation der abstrakten Ebene. Die Grundstruktur der Datenwandlung, Einbettung der Java Implementierung in eine ASM und die grundlegenden Beweisverpflichtungen bleiben allerdings auch für andere Anwendungsdomänen erhalten.

Zusammenfassend hat die Arbeit sowohl eine Fragestellung der aktuellen Forschung beantwortet, allerdings auch Raum zur Weiterentwicklung, weiteren Automatisierung und der Anwendung auf andere Domänen gegeben. Insbesondere die korrekte Generierung von Code ist dabei sehr erfolgversprechend.

Spezifikationshierarchie der Bibliothek

Folgende Grafik illustriert die Spezifikationshierarchie für die Datentypen, Implementierungen, Prädikate und Funktionen für Dokumente, Transformationsschicht und Bibliotheken. Ellipsen bezeichnen dabei einzelne algebraische Spezifikationen, Rechtecke bezeichnen ganze Spezifikationsbibliotheken wie z.B. für die Java-Datentypen oder die Prosecco-Bibliothek. Neben den Ellipsen sind jeweils exemplarisch die Signatursymbole angegeben, die in der Spezifikation eingeführt werden.

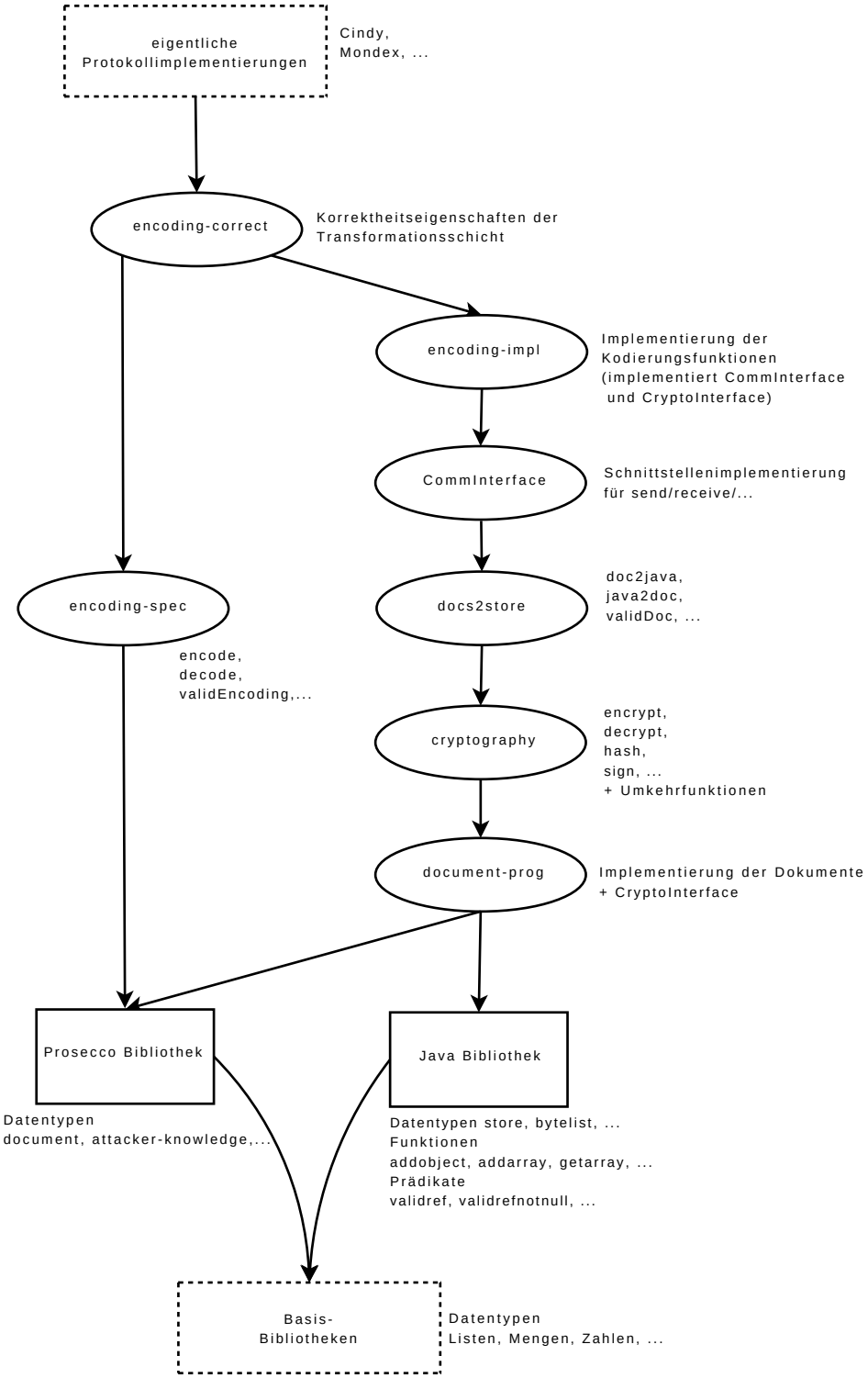


Abbildung 1: Struktur der Basis-Spezifikationshierarchie

Implementierung der Transformationsschicht

```
1  package swt.documents.middleware;
2
3  import swt.documents.crypto.*;
4  import swt.documents.*;
5  import swt.util.ByteArray;
6
7  public class Coding
8    implements swt.documents.crypto.CryptoInterface,
9               swt.documents.middleware.SimpleComm{
10
11 private BACrypto bacrypto;
12 private BAComm bacomm;
13
14 private static final byte EMPTYDOCTYPE = (byte)0;
15 private static final byte INTDOCTYPE = (byte)1;
16 private static final byte KEYDOCTYPE = (byte)2;
17 private static final byte NONCEDOCTYPE = (byte)3;
18 private static final byte SECRETDOCTYPE = (byte)4;
19 private static final byte HASHDOCTYPE = (byte)5;
20 private static final byte ENCDOCTYPE = (byte)6;
21 private static final byte SIGDOCTYPE = (byte)7;
22 private static final byte DOCLISTTYPE = (byte)8;
23
24 int c = 0;
25
26 public Coding(BACrypto bacrypto, BAComm bacomm){
27  this.bacrypto = bacrypto;
28  this.bacomm = bacomm;
29 }
30
31 public byte[] encode(Document d) {
32  return encoderec(d, 0);
33 }
34
35 public byte[] encoderec(Document d, int c){
36  if (c > 1000000000) return null;
```

```
37  byte[] docValue = null;
38  byte head = 0;
39  byte[] bytes = null;
40  if (d == null) {
41      return new byte[]{ EMPTYDOCTYPE, 0, 0, 0, 0 };
42  }
43  else if(d instanceof IntDoc){
44      docValue = ((IntDoc)d).getValue();
45      if(docValue == null || ! noLeadingZeros(docValue,0))
46         return null;
47      head = INTDOCTYPE;
48  }
49  else if (d instanceof KeyDoc){
50      Key key = ((KeyDoc)d).getKey();
51      if(key == null)
52         return null;
53      docValue = key.getBA();
54      if(docValue == null || ! noLeadingZeros(docValue,0))
55         return null;
56      head = KEYDOCTYPE;
57  }
58  else if (d instanceof NonceDoc){
59      Nonce nonce = ((NonceDoc)d).getNonce();
60      if(nonce == null)
61         return null;
62      docValue = nonce.getNonce();
63      if(docValue == null || ! noLeadingZeros(docValue,0))
64         return null;
65      head = NONCEDOCTYPE;
66  }
67  else if (d instanceof SecretDoc){
68      docValue = ((SecretDoc)d).getSecret();
69      if(docValue == null || ! noLeadingZeros(docValue,0))
70         return null;
71      head = SECRETDOCTYPE;
72  }
73  else if (d instanceof HashDoc){
74      docValue = ((HashDoc)d).getHash();
75      if(docValue == null)
76         return null;
77      head = HASHDOCTYPE;
78  }
79  else if (d instanceof EncDoc){
80      docValue = ((EncDoc)d).getEncrypted();
81      if(docValue == null)
82         return null;
83      head = ENCDOCTYPE;
```

```
 }
 else if (d instanceof SigDoc){
     docValue = ((SigDoc)d).getSig();
     if(docValue == null)
        return null;
     head = SIGDOCTYPE;
 }
 else if (d instanceof Doclist){
     docValue = new byte[0];
     head = DOCLISTTYPE;
     Document[] docs = ((Doclist)d).getDocs();
     if(docs == null || ! docArray(docs))
        return null;
     for(int i = 0; i < docs.length; i++)
        {
            byte[] b = encoderec(docs[i], c + 1);
            if(b == null) return null;

            if ((int)(b.length + docValue.length) < 0)
               return null;
            docValue = ByteArray.append(docValue, b);
        }
 }
 if(docValue == null)
     return null;
 if(docValue.length > 2147483642)
     return null;
 bytes = new byte[5 + docValue.length];
 setHead(head, bytes, docValue.length);
 ByteArray.copy(docValue,0, bytes, 5, docValue.length);
 return bytes;
}

private boolean docArray(Document[] r){
 if(r instanceof IntDoc[] ||
    r instanceof SecretDoc[] ||
    r instanceof NonceDoc[] ||
    r instanceof KeyDoc[] ||
    r instanceof HashDoc[] ||
    r instanceof EncDoc[] ||
    r instanceof SigDoc[] ||
    r instanceof Doclist[] )
     return false;
 return true;
}

private void setHead(byte head, byte[] bytes, int length){
```

```
131  bytes[0] = head;
132  ByteArray.setInt(bytes,1,length);
133 }
134
135 public  Document decode(byte[] b){
136  if(b != null && validEncoding(b)){
137      if(b[0] == INTDOCTYPE){
138         return new IntDoc(ByteArray.restn(5, b));
139      }
140      else if(b[0] == KEYDOCTYPE){
141         return new KeyDoc(new Key(ByteArray.restn(5, b)));
142      }
143      else if(b[0] == NONCEDOCTYPE){
144         return new NonceDoc(new Nonce(ByteArray.restn(5, b)));
145      }
146      else if(b[0] == SECRETDOCTYPE){
147         return new SecretDoc(ByteArray.restn(5, b));
148      }
149      else if(b[0] == HASHDOCTYPE){
150         return new HashDoc(ByteArray.restn(5, b));
151      }
152      else if(b[0] == ENCDOCTYPE){
153         return new EncDoc(ByteArray.restn(5, b));
154      }
155      else if(b[0] == SIGDOCTYPE){
156         return new SigDoc(ByteArray.restn(5, b));
157      }
158      else if(b[0] == DOCLISTTYPE){
159         return decodeList( ByteArray.restn(5, b) );
160      }
161  }
162  return null;
163 }
164
165 private Doclist decodeList(byte[] bytes){
166  if(bytes.length == 0)
167      return new Doclist();
168
169  int length = ByteArray.getInt(bytes, 1);
170
171  Document doc =
172    decode( ByteArray.firstn(length + 5, bytes) );
173  Doclist doclist =
174    decodeList( ByteArray.restn(length + 5, bytes) );
175
176  return doclist.cons(doc);
177 }
```

```
178
179 private boolean validEncoding(byte[] bytes){
180  if(bytes == null) return false;
181  if(bytes.length == 0) return false;
182
183  if(   bytes[0] == INTDOCTYPE
184     || bytes[0] == KEYDOCTYPE
185     || bytes[0] == NONCEDOCTYPE
186     || bytes[0] == SECRETDOCTYPE){
187      if( !(   bytes != null
188           && bytes.length <= 2147483647
189           && 6 <= bytes.length) )
190         return false;
191      int length = ByteArray.getInt(bytes, 1);
192      if(length != bytes.length - 5)
193         return false;
194      if( !noLeadingZeros(bytes, 5) )
195         return false;
196      return true;
197  }
198  else if(bytes[0] == DOCLISTTYPE){
199      if( !(   bytes != null
200           && bytes.length <= 2147483647
201           && 5 <= bytes.length) )
202         return false;
203      int length = ByteArray.getInt(bytes, 1);
204      if(length != bytes.length - 5)
205         return false;
206      if( !validListEncoding( ByteArray.restn(5, bytes) ) )
207         return false;
208      return true;
209  }
210  else if(   bytes[0] == HASHDOCTYPE
211          || bytes[0] == ENCDOCTYPE
212          || bytes[0] == SIGDOCTYPE){
213      if( !(   bytes != null
214           && bytes.length <= 2147483647
215           && 5 <= bytes.length) )
216         return false;
217      int length = ByteArray.getInt(bytes, 1);
218      if(length != bytes.length - 5)
219         return false;
220      return true;
221  }
222  else if(bytes[0] == EMPTYDOCTYPE){
223      if( !(bytes != null && 5 == bytes.length) )
224         return false;
```

```
225      int length = ByteArray.getInt(bytes, 1);
226      if(length != 0)
227         return false;
228      return true;
229   }
230   else
231      return false;
232  }
233
234  private boolean validListEncoding(byte[] b){
235   if(b == null)
236      return false;
237   if(b.length == 0)
238      return true;
239   else {
240    if( !(   b != null
241          && b.length <= 2147483647
242          && 5 <= b.length) )
243    return false;
244
245    int length = ByteArray.getInt(b, 1);
246    if(length + 5 > b.length || length < 0)
247      return false;
248    if( !validEncoding(
249            ByteArray.firstn(length + 5, b)) )
250      return false;
251    if( !validListEncoding(
252            ByteArray.restn(length + 5, b)) )
253      return false;
254    return true;
255   }
256  }
257
258  private boolean noLeadingZeros(byte[] b, int offset){
259   if(b.length == 0) return false;
260   if(   b[offset] == (byte)0
261      && (offset + 1) != b.length
262      && b[offset + 1] >= (byte)0)
263       return false;
264   if(   b[offset] == (byte)-1
265      && (offset + 1) != b.length
266      && b[offset + 1] < (byte)0)
267       return false;
268   return true;
269  }
270
271  //+++++++++++++++++++++++++++++++++++++++++++++++++
```

```
272
273 public void send(Document d){
274  bacomm.send(encode(d));
275 }
276
277 public Document receive(){
278  byte[] bytes = bacomm.receive();
279  if(bytes != null)
280      return decode(bytes);
281  return null;
282 }
283
284 public boolean available(){
285     return bacomm.available();
286 }
287
288 public byte[] encrypt(Key k, Document d){
289     byte[] b = encode(d);
290     if(b != null)
291         return bacrypto.encrypt(b, k.getBA());
292     return null;
293 }
294
295 public  Document decrypt(Key k, Document d){
296     Document d0 = null;
297     if(d instanceof EncDoc)
298         d0 = decode(
299                 bacrypto.decrypt(
300                    ((EncDoc) d).getEncrypted(),
301                    k.getBA()));
302     if(d0 instanceof IntDoc)
303         return null;
304     return d0;
305 }
306
307 public  byte[] hash(Document d){
308     byte[] b = encode(d);
309     if(b != null)
310         return bacrypto.hash(b);
311     return null;
312 }
313
314 public  byte[] sign(Key k, Document d) {
315     byte[] b = encode(d);
316     if(b != null)
317         return bacrypto.sign(b, k.getBA());
318     return null;
```

```
319 }
320
321 public  boolean verify(Document plain, Document sig, Key k){
322     if(sig instanceof SigDoc)
323         return bacrypto.verify(
324                 ((SigDoc)sig).getSig(),
325                 encode(plain),
326                 k.getBA());
327     else
328         return false;
329 }
330
331 public  byte[] nextNonce(){
332     return bacrypto.nextNonce();
333 }
334 }
```

Implementierung der Cindy Anwendung

```
1
2  package swt.cindy;
3
4  import swt.documents.Doclist;
5  import swt.documents.Document;
6  import swt.documents.IntDoc;
7  import swt.documents.NonceDoc;
8  import swt.documents.comm.CommInterface;
9
10 public class Protocol {
11
12   private static Protocol theinstance;
13
14
15   private static final byte[] CINEMA = {5,5,5,0,0,0,1};
16   private static final byte LOADTICKET     = 1;
17   private static final byte BUYTICKET      = 3;
18   private static final byte PASSTICKET     = 4;
19   private static final byte PRESENT        = 5;
20   private static final int MAXTICKETLEN = 100;
21   private static final int USERPORT    = 1;
22   private static final int PHONEPORT   = 2;
23   private static final int DISPLAYPORT = 3;
24   private static final int DIRECTSEND = 1;
25
26
27   private CommInterface comm;
28   private static CommInterface initcomm;
29   private Doclist tickets;
30
31   public Protocol(CommInterface comm){
32     this.comm = comm;
33     // initialize database
34     tickets = new Doclist();
35   }
36
```

```
37   public void step(){
38    if(comm.available(USERPORT)){
39      // user-request (buy, passon oder present)
40      Document inmsg = comm.receive(USERPORT);
41      userStep(inmsg);
42    }
43    else if(comm.available(PHONEPORT)){
44      // received a SMS (loadticket)
45      Document inmsg = comm.receive(PHONEPORT);
46      phoneStep(inmsg);
47    }
48   }
49
50   private void phoneStep(Document inmsg) {
51     Document originator = inmsg.getPart(1);
52     inmsg = inmsg.getPart(2);
53     Doclist ticket = getTicket(inmsg, originator);
54     if(ticket != null && tickets.len() < MAXTICKETLEN){
55         tickets = tickets.attach(ticket);
56     }
57   }
58
59   private void userStep(Document indoc) {
60     indoc = indoc.getPart(2);
61     //PRESENT
62     int index = getPresentIndex(indoc);
63     if (0 <= index) {
64         comm.send(
65           nonce(tickets.getPart(index+1)),
66           CINEMA, DISPLAYPORT, DIRECTSEND);
67         return;
68     }
69
70     //PASSON
71     index = getPassIndex(indoc);
72     if (0 <= index) {
73       byte[] receiver = getReceiver(indoc);
74       comm.send(
75         new Doclist(new Document[]{
76                     new IntDoc(new byte[]{LOADTICKET}),
77                     ticketdata(tickets.getPart(index+1))}),
78         receiver,
79         PHONEPORT,
80         DIRECTSEND);
81       return;
82     }
83
```

```
84      //BUY (only forwarding user request)
85      boolean  ok = okTicketData(indoc);
86      if(ok)
87      {
88        comm.send(indoc,CINEMA, PHONEPORT, DIRECTSEND);
89        return;
90      }
91    }
92
93    private byte[] getReceiver(Document indoc) {
94      return ((IntDoc)(indoc.getPart(2)).getPart(2)).getValue();
95    }
96
97    private int getPassIndex(Document indoc){
98     if(indoc != null && indoc.is_comdoc()){
99          byte[] ins = indoc.getPart(1).getValue();
100         if(ins.length == 1 && ins[0] == PASSTICKET){
101           indoc = indoc.getPart(2);
102      if (indoc != null && indoc.len() == 2) {
103        Document indoc1 = indoc.getPart(1);
104        Document indoc2 = indoc.getPart(2);
105        if(indoc1 != null && indoc1.is_intdoc()
106                       && indoc2 != null && indoc2.is_intdoc())
107        {
108          if(indoc1.getValue().length == 1 &&
109             0 <= indoc1.getValue()[0] &&
110             indoc1.getValue()[0] < tickets.len())
111            return indoc1.getValue()[0];
112        }
113      }
114      }
115     }
116     return -1;
117    }
118
119
120    private boolean okTicketData(Document indoc){
121     if(indoc != null && indoc.is_comdoc())
122     {
123       byte[] ins = indoc.getPart(1).getValue();
124       if(ins.length == 1 && ins[0] == BUYTICKET) {
125         indoc = indoc.getPart(2);
126         if(indoc!= null && indoc.len() == 2){
127           Document indoc1 = indoc.getPart(1);
128             Document indoc2 = indoc.getPart(2);
129             if(indoc1 != null && indoc1.is_intdoc() &&
130             indoc2 != null && indoc2.is_intdoc()) {
```

```
131             return true;
132           }
133         }
134     }
135   }
136   return false;
137  }
138
139
140  private Doclist getTicket(Document indoc,
141                            Document originator) {
142    if(indoc != null && indoc.is_comdoc()){
143      byte[] ins = indoc.getPart(1).getValue();
144      if(ins.length == 1 && ins[0] == LOADTICKET)
145      {
146      indoc = indoc.getPart(2);
147        if(indoc != null && indoc.len() == 2){
148          Document indoc1 = indoc.getPart(1);
149          Document indoc2 = indoc.getPart(2);
150          if(indoc1 != null && indoc1.is_intdoc() &&
151             indoc2 != null && indoc2.is_noncedoc()){
152              return new Doclist(originator, indoc);
153          }
154        }
155       }
156    }
157    return null;
158  }
159
160
161  private int getPresentIndex(Document indoc) {
162    if(indoc != null && indoc.is_comdoc()){
163      byte[] ins = indoc.getPart(1).getValue();
164      if(ins.length == 1 && ins[0] == PRESENT)
165      {
166        indoc = indoc.getPart(2);
167        if(indoc != null && indoc.is_intdoc())
168        {
169          if(indoc.getValue().length == 1
170             && 0 <= indoc.getValue()[0]
171             && indoc.getValue()[0] < tickets.len()) {
172              return indoc.getValue()[0];
173          }
174        }
175      }
176    }
177    return -1;
```

```
178   }
179
180
181   private Document nonce(Document d){
182     return d.getPart(2).getPart(2);
183   }
184
185
186   private Document ticketdata(Document d){
187     return d.getPart(2);
188   }
189 }
```

Implementierung der Mondex Anwendung

```
1
2  package mondexOnCard;
3
4  import javacard.framework.*;
5
6  import swt.documents.*;
7  import swt.documents.middleware.*;
8  import swt.documents.crypto.*;
9
10 public class Purse {
11
12   private static Purse theinstance;
13   private static SimpleComm initcomm;
14   private static Document initdata;
15
16   private static final byte STATE_IDLE = 1;
17   private static final byte STATE_EPR  = 2;
18   private static final byte STATE_EPV  = 3;
19   private static final byte STATE_EPA  = 4;
20
21   private static final byte INS_START_FROM = 1;
22   private static final byte INS_START_TO   = 2;
23   private static final byte INS_REQ        = 3;
24   private static final byte INS_VAL        = 4;
25   private static final byte INS_ACK        = 5;
26   private static final byte INS_GET_BAL    = 6;
27   private static final byte INS_GET_DATA   = 7;
28   private static final byte INS_GET_STATE  = 8;
29
30   private byte[]    name;
31   private short     sequenceNo;
32   private short     balance;
33   private byte      state;
34   private short     exLogCounter;
35   private Doclist[] exLog;
36
```

```
37   private Doclist pd;
38   private Doclist outmsg;
39   private Doclist getdata_doc;
40   private EncDoc  encmsg;
41   private IntDoc  bal_state_doc;
42
43   private SimpleComm comm;
44
45   private SessionKey key;
46
47   public Purse(SimpleComm initcomm, Document initdata) {
48     // check initdata:
49     // document must be of type:
50     // doclist(intdoc1,intdoc2,intdoc3)
51     // intdoc1 must have 8 digits (thename)
52     // intdoc2 must have 2 digits (initbal)
53     // intdoc3 must have 2 digits (loglen)
54     Document thename = initdata.getPart((short)1);
55     Document initbal = initdata.getPart((short)2);
56     Document loglen = initdata.getPart((short)3);
57     if(!(thename != null && thename.is_intdoc() &&
58        thename.getValue().length == 8           &&
59        initbal   != null && initbal.is_intdoc() &&
60        initbal.getValue().length == 2 &&
61        loglen    != null &&  loglen.is_intdoc() &&
62        loglen.getValue().length == 2)) return;
63
64     short theloglen  = Util.getShort(loglen.getValue(),
65                                      (short)0);
66     short thebalance = Util.getShort(initbal.getValue(),
67                                      (short)0);
68     if(theloglen<=0 || thebalance <=0) return;
69
70     // init all fields
71     initExLog(theloglen);
72     initPaydetails();
73     initSimpleFields();
74     initSessionKey();
75     initOutMessages();
76
77     // save name, balance and comminterface
78     Util.arrayCopy(thename.getValue(),(short)0,
79                    name,(short)0,(short)8);
80     balance = thebalance;
81     comm    = initcomm;
82   }
83
```

```
  private void initExLog(short len) {
    exLog        = new Doclist[len];
        exLogCounter = (short)0;
  }

  private void initPaydetails() {
    pd = new Doclist(
          new Document[]{
      new IntDoc(new byte[8]),
      new IntDoc(new byte[2]),
      new IntDoc(new byte[8]),
      new IntDoc(new byte[2]),
      new IntDoc(new byte[2])});
  }

  private void initSimpleFields() {
    sequenceNo = (short)0;
    state      = STATE_IDLE;
    name       = new byte[8];
  }

  private void initSessionKey() {
    key = new SessionKey(
      new byte[]{1,2,3,4,5,6,7,8,9,0,
           1,2,3,4,5,6,7,8,9,0,
           1,2,3,4});
  }

  private void initOutMessages() {
    outmsg         = new Doclist(
          new Document[]{
            new IntDoc(new byte[2]),null});
    bal_state_doc = new IntDoc(new byte[2]);
    getdata_doc   = new Doclist(
          new Document[]{
            new IntDoc(new byte[8]),
            new IntDoc(new byte[2])});
    encmsg        = new EncDoc(new byte[56]);
  }

  private void mkpd(byte[] fromPurse,
        short nextSeqNoFromPurse,
                          byte[] toPurse,
        short nextSeqNoToPurse,
        short value) {
    Util.arrayCopy(fromPurse,(short)0,
             pd.getPart((short)1).getValue(),
```

```
131                                 (short)0,(short)8);
132     Util.setShort(pd.getPart((short)2).getValue(),
133             (short)0,nextSeqNoFromPurse);
134     Util.arrayCopy(toPurse,(short)0,
135              pd.getPart((short)3).getValue(),
136                                 (short)0,(short)8);
137     Util.setShort(pd.getPart((short)4).getValue(),
138             (short)0,nextSeqNoToPurse);
139     Util.setShort(pd.getPart((short)5).getValue(),
140             (short)0,value);
141   }
142
143   private short nextSeqNo() {
144     return sequenceNo++;
145   }
146
147   private Document checkIndoc(Document indoc) {
148   if(indoc.is_encdoc()) {
149     indoc = Crypto.getCrypto().decrypt(key,indoc);
150     Document[] docs = ((Doclist)indoc).getDocs();
151     if(docs.length==2) {
152       Document part1 = indoc.getPart((short)1);
153       Document part2 = indoc.getPart((short)2);
154       if(part1.is_intdoc() && part2.is_doclist()) {
155         byte[] part1value = part1.getValue();
156         Document[] part2docs = ((Doclist)part2).getDocs();
157         if(part1value.length==2 && part1value[0] == 0 &&
158            (part1value[1]==INS_REQ ||
159            part1value[1]==INS_VAL      ||
160            part1value[1]==INS_ACK)) {
161           if(part2docs.length==5 &&
162             part2docs[0].is_intdoc() &&
163             part2docs[0].getValue().length == 8 &&
164             part2docs[1].is_intdoc() &&
165             part2docs[1].getValue().length == 2 &&
166             part2docs[2].is_intdoc() &&
167             part2docs[2].getValue().length == 8 &&
168             part2docs[3].is_intdoc() &&
169             part2docs[3].getValue().length == 2 &&
170             part2docs[4].is_intdoc() &&
171             part2docs[4].getValue().length == 2)
172             return indoc;
173         }
174       }
175     }
176   }
177   else if(indoc.is_intdoc()) {
```

```
178     byte[] theval = indoc.getValue();
179     if(theval.length == 2 && theval[0] == 0 && (
180         theval[1]==INS_GET_BAL   ||
181         theval[1]==INS_GET_DATA  ||
182         theval[1]==INS_GET_STATE))
183       return indoc;
184   }
185   else if(indoc.is_doclist()) {
186     Document[] docs = ((Doclist)indoc).getDocs();
187     if(docs.length==2) {
188       Document part1 = indoc.getPart((short)1);
189       Document part2 = indoc.getPart((short)2);
190       if(part1.is_intdoc() && part2.is_doclist()) {
191         byte[] part1value = part1.getValue();
192         Document[] part2docs = ((Doclist)part2).getDocs();
193         if(part1value.length == 2 && part1value[0] == 0 &&
194              (part1value[1]==INS_START_FROM ||
195               part1value[1]==INS_START_TO)) {
196               if(part2docs.length==3 &&
197                 part2docs[0].is_intdoc() &&
198                 part2docs[0].getValue().length == 8 &&
199                 part2docs[1].is_intdoc() &&
200                 part2docs[1].getValue().length == 2 &&
201                 part2docs[2].is_intdoc() &&
202                 part2docs[2].getValue().length == 2)
203                 return indoc;
204         }
205       }
206     }
207   }
208   return null;
209   }
210
211   private Document generateOutmsg(byte ins) {
212     outmsg.getPart((short)1).getValue()[1] = ins;
213     outmsg.getDocs()[(short)1] = pd;
214     encmsg.setEncrypted(key, outmsg);
215     return encmsg;
216   }
217
218   private byte getInsByte(Document d) {
219     byte ins = 0;
220     if(d==null) return ins;
221     if(d.is_doclist())
222       ins = d.getPart((short)1).getValue()[1];
223     else ins = d.getValue()[1];
224     if((  (   ins==INS_START_FROM
```

```
225           || ins==INS_START_TO)
226         && state==STATE_IDLE)
227      ||(ins==INS_REQ && state==STATE_EPR)
228      ||(ins==INS_VAL && state==STATE_EPV)
229      ||(ins==INS_ACK && state==STATE_EPA))
230     return ins;
231    else if(  ins==INS_GET_BAL
232            || ins==INS_GET_DATA
233            || ins==INS_GET_STATE)
234       return ins;
235    else return (byte)0;
236  }
237
238  private Document checkNameEquals(Document msgna) {
239    if(!(msgna.is_intdoc())) return null;
240    byte[] theval = msgna.getValue();
241    if(!Document.comparison.equals(theval,name)) return msgna;
242    else return null;
243  }
244
245  private short checkBalanceMinus(Document value) {
246    if(!(value.is_intdoc())) return -1;
247    byte[] theval = value.getValue();
248    if(!(theval.length == (short)2)) return -1;
249    short value_short = Util.getShort(theval,(short)0);
250    if(balance<value_short || value_short<=0)
251      return -1;
252    else return value_short;
253  }
254
255  private short checkBalancePlus(Document value) {
256    if(!(value.is_intdoc())) return -1;
257    byte[] theval = value.getValue();
258    if(!(theval.length == (short)2)) return -1;
259    short value_short = Util.getShort(theval,(short)0);
260    if(   value_short<=0
261       || (short)(balance + value_short)<(short)0)
262      return -1;
263    else return value_short;
264  }
265
266  private short checkSeqNoOtherPurse(Document seqno) {
267    if(!(seqno.is_intdoc())) return -1;
268    byte[] theseqno = seqno.getValue();
269    if(!(theseqno.length == (short)2)) return -1;
270    short seqno_s = Util.getShort(theseqno,(short)0);
271    if(seqno_s<0) return -1;
```

```
272     else return seqno_s;
273   }
274
275   public void step() {
276     Document outdoc = null;
277     Document indoc = null;
278     // check if there is a document in the inbox
279     if(comm.available())
280       indoc = comm.receive();
281     else return;
282     indoc = checkIndoc(indoc);
283     switch(getInsByte(indoc)) {
284       case INS_START_FROM:
285                 startFrom(indoc.getPart((short)2));break;
286       case INS_START_TO:
287         outdoc = startTo(indoc.getPart((short)2));  break;
288       case INS_REQ:
289         outdoc = req(indoc.getPart((short)2));      break;
290       case INS_VAL:
291         outdoc = val(indoc.getPart((short)2));      break;
292       case INS_ACK:
293                 ack(indoc.getPart((short)2));      break;
294       case INS_GET_BAL
295         outdoc = getBalance();                      break;
296       case INS_GET_DATA:
297         outdoc = getData();                         break;
298       case INS_GET_STATE:
299         outdoc = getState();                        break;
300       default:
301         abort();                                    break;
302     }
303     // send doc if outdoc is set
304     if(outdoc!=null)
305       comm.send(outdoc);
306   }
307
308   private void abort() {
309     if(state==STATE_EPA || state == STATE_EPV) {
310       // logifneeded
311       if(exLogCounter == exLog.length)
312         return; // no memory
313       // save pds
314       Document[] docs = pd.getDocs();
315       byte[] fromname = new byte[8];
316       byte[] fromseqno = new byte[2];
317       byte[] toname = new byte[8];
318       byte[] toseqno = new byte[2];
```

```
319       byte[] value = new byte[2];
320       Util.arrayCopy(docs[0].getValue(),(short)0,
321                      fromname,(short)0,(short)8);
322       Util.arrayCopy(docs[1].getValue(),(short)0,
323                      fromseqno,(short)0,(short)2);
324       Util.arrayCopy(docs[2].getValue(),(short)0,
325                      toname,(short)0,(short)8);
326       Util.arrayCopy(docs[3].getValue(),(short)0,
327                      toseqno,(short)0,(short)2);
328       Util.arrayCopy(docs[4].getValue(),(short)0,
329                      value,(short)0,(short)2);
330       exLog[exLogCounter++] =
331         new Doclist(new Document[]{
332         new IntDoc(fromname),
333         new IntDoc(fromseqno),
334         new IntDoc(toname),
335         new IntDoc(toseqno),
336         new IntDoc(value)});
337     }
338     nextSeqNo();
339     state = STATE_IDLE;
340   }
341
342   private void startFrom(Document indoc) {
343     // get indocs
344     Document msgna = checkNameEquals(indoc.getPart((short)1));
345     if(msgna==null) return;
346     short value_short =
347       checkBalanceMinus(indoc.getPart((short)2));
348     if(value_short==-1) return;
349     short nextSeqNoToPurse =
350       checkSeqNoOtherPurse(indoc.getPart((short)3));
351     if(nextSeqNoToPurse== -1) return;
352     short seqno = nextSeqNo();
353     if(seqno<0) return;
354     // paydetails, state
355     mkpd(name,seqno,msgna.getValue(),
356          nextSeqNoToPurse,value_short);
357     state = STATE_EPR;
358   }
359
360   private Document startTo(Document indoc) {
361     // get indocs
362     Document msgna =
363       checkNameEquals(indoc.getPart((short)1));
364     if(msgna==null) return null;
365     short value_short =
```

```
366       checkBalancePlus(indoc.getPart((short)2));
367     if(value_short==-1) return null;
368     short nextSeqNoFromPurse =
369       checkSeqNoOtherPurse(indoc.getPart((short)3));
370     if(nextSeqNoFromPurse==-1) return null;
371     short seqno = nextSeqNo();
372     if(seqno<0) return null;
373     // paydetails, state and outdoc
374     mkpd(msgna.getValue(),nextSeqNoFromPurse,
375          name,seqno,value_short);
376     state = STATE_EPV;
377     Document d = generateOutmsg(INS_REQ);
378     return d;
379   }
380
381
382   private Document req(Document indoc) {
383     if(!indoc.equals(pd))
384       return null;
385     balance =
386      (short)(balance -
387              Util.getShort(
388               pd.getPart((short)5).getValue(),
389               (short)0));
390     state = STATE_EPA;
391     return generateOutmsg(INS_VAL);
392   }
393
394   private Document val(Document indoc) {
395     if(!indoc.equals(pd))
396       return null;
397     balance =
398      (short)(balance +
399              Util.getShort(
400               pd.getPart((short)5).getValue(),
401                           (short)0));
402     state = STATE_IDLE;
403     return generateOutmsg(INS_ACK);
404   }
405
406
407   private void ack(Document indoc) {
408     if(!indoc.equals(pd))
409       return;
410     state = STATE_IDLE;
411   }
412
```

```
413   private Document getState() {
414     byte[] theval = bal_state_doc.getValue();
415     theval[0] = 0;
416     theval[1] = state;
417     return bal_state_doc;
418   }
419
420   private Document getData() {
421     Util.arrayCopy(name,(short)0,
422                    getdata_doc.getDocs()[0].getValue(),
423                    (short)0,(short)8);
424     Util.setShort(getdata_doc.getDocs()[1].getValue(),
425                   (short)0,sequenceNo);
426     return getdata_doc;
427   }
428
429   private Document getBalance() {
430     Util.setShort(bal_state_doc.getValue(),
431                   (short)0,balance);
432     return bal_state_doc;
433   }
434 }
```

Literaturverzeichnis

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] Jean-Raymond Abrial and Dominique Cansell. *Click'n Prove: Interactive Proofs within Set Theory.* 2003.

[4] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[5] Reynald Affeldt and Naoki Kobayashi. Formalization and verification of a mail server in Coq. In Mitsuhiro Okada, Benjamin Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *International Symposium on Software Security, Tokyo, Japan, November 8–10, 2002*, volume 2609 of *Lecture Notes in Computer Science*, pages 217–233. Springer, Feb. 2003.

[6] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[7] B4Free. B4free tool homepage `http://www.b4free.com/`.

[8] M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[9] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for Provably Correct Systems. In *Current Trends in Applied Formal Methods*, LNCS 1641. Boppard, Germany, Springer-Verlag, 1999.

[10] Michael Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction.* PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[11] Michael Balser, Christoph Duelli, Dominik Haneberg, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. A practical course on kiv. Technical Report 2006-14, Universität Augsburg, 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se`.

[12] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[13] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the purse: The balance enquiry quandary, and generalised and (1,1) forward refinements. *Fundamenta Informaticae 77*, 2006.

[14] Richard Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 129–147, London, UK, 1998. Springer-Verlag.

[15] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[16] S. Bäumler, M. Balser, A. Knapp, W. Reif, and A. Thums. Interactive verification of uml state machines. In *Formal Methods and Software Engineering*, number 3308 in LNCS. Springer, 2004.

[17] Bernhard Beckert and Gerd Beuster. Formal specification of security-relevant properties of user interfaces. In *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany, 2004. TU Munich Technical Report TUM-I0415.

[18] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[19] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[20] Bernhard Beckert and Vladimir Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE Press, 2004.

[21] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, LNCS 4130, pages 266–280. Springer, 2006.

[22] Bernhard Beckert and Steffen Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4):423–442, 2005.

[23] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.

[24] J. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Aspects Computing*, 18(2):143–151, 2006.

[25] E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15 (1–2):237–257, November 2003.

[26] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. North-Holland, Amsterdam, 1995.

[27] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[28] C. Breunesse, B. Jacobs, and J. van den Berg. Specifying and verifying a decimal representation in Java for smart cards, 2002.

[29] Bundesministerium für Gesundheit. *Spezifikationen der elektronischen Gesundheitskarte*, December 2005. URL: `http://www.bmg.bund.de` (Letzter Abruf: 04.09.2006).

[30] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[31] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3), 2005.

[32] N. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *Formal Methods Europe (FME)*, Springer LNCS 2805, 2003.

[33] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[34] Michael Burrows, Martín Abadi, and Roger M. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London*, (Series A, 426, 1871), 1989.

[35] Michael Butler and Divakar Yadav. An incremental development of the mondex system in event-b. *Formal Aspects of Computing*, 20(1):61–77, January 2008.

[36] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical report, Microsoft Research, 2005. URL: `ftp://ftp.research.microsoft.com/pub/tr/TR-2005-59.pdf`.

[37] J. Carlton and D. Crocker. A High Productivity Tool for Formally Verified Softare Development. Technial Report, Escher.

[38] N. Cataõo and M. Huisman. Formal specification of Gemplus' electronic purse case study using ESC/Java. In *Proceedings, Formal Methods Europe (FME 2002)*, number 2391 in LNCS, pages 272–289. Springer, 2002.

[39] UK ITSEC Certification Body. UK ITSEC SCHEME CERTIFICATION REPORT No. P129 MONDEX Purse. Technical report, UK IT Security Evaluation and Certification Scheme, 1999. URL: `http://www.cesg.gov.uk/site/iacs/itsec/media/certreps/CRP129.pdf`.

[40] David R. Cok and Joseph Kiniry. Esc/java2: Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, pages 108–128, 2004.

[41] David Basin, Sebastian Mödersheim, and Luca Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. Technical Report 404, ETH Zürich, 2003.

[42] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technial Report SRI-CSL-01-02, SRI International, 2003.

[43] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Proceedings, Budapest, Hungary*, Springer LNCS (to appear), 2008.

[44] W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[45] J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001.

[46] Bundesministerium des Inneren. Epass `http://www.epass.de/`.

[47] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[48] Mobile Tickets Deutsche Bahn. `http://www.bahn.de/`.

[49] Web Präsentation der Projekte dieser Arbeit. URL: `http://www.informatik.uni-augsburg.de/swt/projects/grandy/`.

[50] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*, pages 350–357. IEEE, 1981.

[51] Olivier Dubuisson. *ASN.1 - Communication Between Heterogeneous Systems*. Elsevier-Morgan Kaufmann, 2000.

[52] H. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. BI Wissenschaftsverlag, 1992.

[53] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[54] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer-Verlag.

[55] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37:5, pages 234–245, June 2002.

[56] JavaCard Forum. Webseite `http://www.javacardforum.org/`.

[57] Thomas Fuchß, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.

[58] Chris George. Tutorial on the raise language, method and tools. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*. Springer, 2004.

[59] EURO Kartensysteme GmbH. Gerldkarte `http://www.geldkarte.de`.

[60] Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[61] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java (tm) Language Specification, Third Edition*. Addison-Wesley, 2005.

[62] H. Grandy, R. Bertossi, K. Stenzel, and W. Reif. ASN1-light: A Verified Message Encoding for Security Protocols. In *Software Engineering and Formal Methods, SEFM*. IEEE Press, 2007.

[63] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel. Developing Provably Secure M-Commerce Applications. In Günter Müller, editor, *Emerging Trends in Information and Communication Security*, volume 3995 of *LNCS*, pages 115–129. Springer, 2006.

[64] H. Grandy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, and W. Reif. The Mondex Case Study: From Specifications to Code. Technical Report 2006-31, University of Augsburg, December 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/`.

[65] H. Grandy, K. Stenzel, and W. Reif. Object-Oriented Verification Kernels for Secure Java Applications. In B. Aichering and B. Beckert, editors, *SEFM 2005 – 3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Press, 2005.

[66] H. Grandy, K. Stenzel, and W. Reif. A Refinement Method for Java Programs. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *LNCS*. Springer, 2007.

[67] Holger Grandy, Markus Bischof, Kurt Stenzel, Gerhard Schellhorn, and Wolfgang Reif. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code . In *Proceedings of Formal Methods 2008 (FM08), (to appear)*, LNCS, Turku, Finland, 2008. Springer.

[68] Holger Grandy, Kurt Stenzel, and Wolfgang Reif. Refinement of Security Protocol Data Types to Java. In *PASSWORD Workshop 2006 at ECOOP 2006*, Nantes, France, 2006. URL: `http://research.ihost.com/password/`.

[69] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford Univ. Press, 1995.

[70] Reiner Hähnle, Maritta Heisel, Wolfgang Reif, and Werner Stephan. An Interactive Verification System Based on Dynamic Logic. In J. Siekmann, editor, *8th International Conference on Automated Deduction. Proceedings.* Springer LNCS 230, 1986.

[71] D. Haneberg. *Sicherheit von Smart Card – Anwendungen.* PhD thesis, University of Augsburg, Augsburg, Germany, 2006. (in German).

[72] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Smart Card Applications: An ASM Approach. In *International Conference on integrated Formal Methods (iFM) 2007*, volume 4591 of *LNCS*. Springer, 2007.

[73] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of mondex electronic purses with kiv: From transactions to a security protocol. *Formal Aspects of Computing*, 2007. accepted.

[74] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing*, 2007.

[75] Dominik Haneberg, Simon Bäumler, Michael Balser, Holger Grandy, Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Jonathan Schmitt, and Kurt Stenzel. The User Interface of the KIV Verification System — A System Description. *Electronic Notes in Theoretical Computer Science UITP special issue*, 2006.

[76] David Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[77] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[78] A. E. Haxthausen, C. George, and M. Schütz. Specification and Proof of the Mondex Electronic Purse. In *1st Asian Working Conference on Verified Software, AWCVS'06, UNU-IIST Reports 348, Macau*, 2006.

[79] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer, 1986.

[80] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[81] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[82] C. A. R. Hoare. The ideal of verified software. In T. Ball and R. B. Jones, editors, *Proceeding 18th International Conference on Computer Aided Verification (CAV06)*, volume 4144 of *LNCS*, pages 5–16, Seattle, 2006. Springer.

[83] C. A. R. Hoare and R. Milner. Grand challenges for computing. *Research Comput. J.*, 48(1):49–52, 2005.

[84] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in java card. In *First International Conference on Security in Pervasive Computing, Boppard, Germany*, volume 2802 of *Lecture Notes in Computer Science*. Springer, 2004.

[85] M. Huisman. *Reasoning about JAVA programs in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, IPA dissertation series, 2001-03, 2001.

[86] Marieke Huisman. Verification of java's abstract collection class: A case study. In *MPC '02: Proceedings of the 6th International Conference on Mathematics of Program Construction*, pages 175–194, London, UK, 2002. Springer-Verlag.

[87] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Deduction in the verification support environment (vse). In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 268–286, London, UK, 1996. Springer-Verlag.

[88] Schwerpunktprogramm Sicherheit in der Informations-und Kommunikationstechnik. `http://www.telematik.uni-freiburg.de/spps/index.php`.

[89] MasterCard International Inc. Mondex `http://www.mondex.com`.

[90] European Telecommunications Standards Institute. *Specification of the Subscriber Identity Module - GSM 11.11*. 1995.

[91] ITSEC. IT-Sicherheitskriterien und Evaluierung nach ITSEC URL: `http://www.bsi.bund.de/zertifiz/itkrit/itsec.htm`.

[92] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[93] B. Jacobs, M. Oostdijk, and M. Warnier. Source code verification of a secure payment applet. *JLAP*, 58:107–120, 2004.

[94] B. Jacobs and E. Poll. A logic for the java modeling language JML. In *Proceedings FASE 2001*, Genova, Italy, 2001. Springer LNCS 2029.

[95] Bart Jacobs. Weakest Precondition Reasoning for Java Programs with JML Annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[96] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Shankland C. Rattray, S. Maharaj, editor, *Algebraic Methodology and Software Technology (AMAST 04)*, volume 3116 of *LNCS*, pages 241–257. Springer, July 2004.

[97] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, U. Hensel, and H. Tews. Reasoning about java classes: preliminary report. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 329–340, New York, NY, USA, 1998. ACM.

[98] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[99] C. Jones, P. W. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[100] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.

[101] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[102] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362, January 2005.

[103] Joseph R. Kiniry, Alan E. Morkan, and Barry Denby. Soundness and completeness warnings in esc/java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 19–24, New York, NY, USA, 2006. ACM.

[104] Web Präsentation der KIV Projekte. URL: `http://www.informatik.uni-augsburg.de/swt/projects/`.

[105] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[106] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue on Formal Techniques for Java.

[107] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[108] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[109] W. Kong, K. Ogata, and K. Futatsugi. Algebraic approaches to formal analysis of the mondex electronic purse system. In *International Conference on integrated formal methods (iFM) 2007*, volume 4591 of *LNCS*. Springer, 2007.

[110] Daniel Kröning. *Formal Verification of Pipelined Microprocessors.* PhD thesis, Universität des Saarlands, Saarbrücken, Germany, 2001.

[111] Mirco Kuhlmann and Martin Gogolla. Modeling and validating mondex scenarios described in uml and ocl with use. *Formal Aspects of Computing*, 20(1):79–100, January 2008.

[112] Daniel Le Métayer and Renaud Marlet. Verification of cryptographic protocols implemented in java card. In *e-Smart conference (e-Smart 2003), Sophia Antipolis*, sep 2003.

[113] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[114] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[115] Gary T. Leavens, Joseph R. Kiniry, and Erik Poll. A JML tutorial. Technical report, CAV 2007 Tutorial, `http://cav2007.org/Docs/Leavens.JML.ps4.pdf`, 2007.

[116] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types.* Wiley-Teubner, 1996.

[117] G. Lowe. Casper: A compiler for the analysis of security protocols. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1997.

[118] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*, pages 147–166. Springer LNCS 1055, 1996.

[119] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, 1991.

[120] C. Marche, Paulin C. Mohring, and X. Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[121] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[122] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[123] Hans Meijer and Erik Poll. Towards a full formal specification of the JavaCard API. *Lecture Notes in Computer Science*, 2140:165+, 2001.

[124] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 1992.

[125] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 63–77, London, UK, 2000. Springer-Verlag.

[126] Wolfram Schulte. Mike Barnett, K. Rustan M. Leino. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*. Springer, 2004.

[127] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Inf. Comput.*, 100(1), 1992.

[128] MindMatics. `http://www.mindmatics.de/`.

[129] N. Moebius, D. Haneberg, G. Schellhorn, and W. Reif. A Modeling Framework for the Development of Provably Secure E-Commerce Applications. In *International Conference on Software Engineering Advances 2007*. IEEE Press, 2007.

[130] Wojciech Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *Verify'07 4th International Verification Workshop*, volume 259 of *CEUR WS*, July 2007.

[131] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In Frank Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer-Verlag, 2007.

[132] Mogens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The raise language, method and tools. *Formal Asp. Comput.*, 1(1):85–114, 1989.

[133] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[134] Object Management Group. *Object Constraint Language 2.0*, 2006. http://www.omg.org/docs/formal/06-05-01.pdf.

[135] Kazuhiro Ogata and Kokichi Futatsugi. Proof scores in the ots/cafeobj method. In *Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19.21, 2003, Proceedings*, volume 2884 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2003.

[136] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/ oheimb/diss/`.

[137] Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Andreas Thums, Bernhard Hering, and Helmut Trappschuh. Safety analysis of the height control system for the Elbtunnel. In *SafeComp 2002*, pages 296 – 308, Catania, Italy, 2002. Springer LNCS 2434.

[138] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[139] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6, 1998.

[140] Adrian Perrig and Dawn Song. A first step towards the automatic generation of security protocols. In *Network and Distributed System Security Symposium, NDSS '00*, pages 73–84, February 2000.

[141] Adrian Perrig and Dawn Song. Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. In *Computer Security Foundations Workshop, CSFW 13*, July 2000.

[142] E. Poll and A. Schubert. Verifying an implementation of SSH. In *Workshop on Issues in the Theory of Security*, pages 164–177. IFIP WG1.7, 2007.

[143] The GoCard Project. Gocard, `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/gocard/`.

[144] VeriSoft Projekt. Verisoft `http://www.verisoft.de/`.

[145] T. Ramananadro. Mondex, an electronic purse : specification and refinement checks with the alloy model-finding method. *Formal Aspects of Computing*, 20(1), 2008.

[146] T. Ramananadro and D. Jackson. Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. URL: `http://www.eleves.ens.fr/home/ramanana/work/mondex/`, 2006.

[147] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley, 3rd edition, 2003.

[148] W. Reif, G. Schellhorn, and K. Stenzel. Proving System Correctness with KIV. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development. Proceedings*. Springer LNCS 1214, 1997.

[149] W. Reif, G. Schellhorn, and K. Stenzel. Proving System Correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*. Townsville, Australia, Springer LNCS 1249, 1997.

[150] Wolfgang Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer-Verlag, Berlin, 1995.

[151] Verified Software Repository. VSR `http://vsr.sf.net/`.

[152] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW '95: Proceedings of the The Eighth IEEE Computer Security Foundations Workshop (CSFW '95)*, page 98. IEEE Computer Society, 1995.

[153] D. Rosenzweig, D. Runje, and W. Schulte. Model-based testing of cryptographic protocols. In *Int. Workshop on Trustworthy Global Computing (TGC)*, volume 3705 of *LNCS*, pages 33–60, Edinburgh, 2005. Springer.

[154] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[155] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. (in German).

[156] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: `http://www.jucs.org`.

[157] G. Schellhorn. ASM refinement preserving invariants. In *Proceedings of the 14th International ASM Workshop, ASM'07*. Grimstad, Norway, 2007.

[158] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.

[159] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering, pages 165 – 194. Kluwer Academic Publishers, 1998.

[160] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, LNCS. Springer, 2007.

[161] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods 2006, Proceedings*, volume 4085 of *LNCS*, pages 16–31. Springer, 2006.

[162] Gerhard Schellhorn and Axel Burandt. Specification and Verification of Distributed Technical Systems with Central Control. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems*. Springer LNCS 891, 1994.

[163] Norbert Schirmer. Java Definite Assignment in Isabelle/HOL. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Formal Techniques for Java-like Programs 2003 (Proceedings)*. Chair of Software Engineering, ETH Zürich, 2003. Technical Report 108.

[164] Norbert Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 16(7):689–706, 2004.

[165] P. H. Schmitt and I. Tonin. Verifying the mondex case study. In *Software Engineering and Formal Methods, SEFM*. IEEE Press, 2007.

[166] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, second edition, 1996.

[167] N. Shankar. PVS: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.

[168] Dawn Xiaodong Song. Athena: a new efficient automatic checker for security protocol analysis. *Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE*, pages 192–202, 1999.

[169] Dawn Xiaodong Song, Adrian Perrig, and Doantam Phan. Agvi - automatic generation, verification, and implementation of security protocols. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 241–245. Springer, 2001.

[170] V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science*. Addison Wesley, 1991.

[171] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[172] K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[173] Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, or `http://www.informatik.uni-augsburg.de/forschung/dissertations/`, 2005.

[174] S. Stepney, D. Cooper, and J. Woodcock. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000. URL: `http://www-users.cs.york.ac.uk/~susan/bib/ss/z/monog.htm`.

[175] Sun Microsystems Inc. *Java Card 2.2 Specification*, 2002. `http://java.sun.com/products/javacard/`.

[176] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[177] The Object Management Group (OMG). *OMG Unified Modeling Language Specification Version 2.1.2*, 2003. http://www.omg.org/technology/documents/formal/uml.htm.

[178] Andreas Thums. *Formale Fehlerbaumanalyse*. PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German).

[179] Beep! Mobile Tickets. `http://www.beep.nl`.

[180] Benjamin Tobler and Andrew Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In *Proceedings of IFIP World Computer Congress - CSES 2004, 2nd International Workshop on Certification and Security in Inter-Organizational E-Services, Toulouse, France*. Springer, 2004.

[181] D. Gerd tom Markotten. *Benutzbare Sicherheit in informationstechnischen Systemen*. Rhombos Verlag, 2003.

[182] I. Tonin. Verifying the mondex case study.the key approach. Techischer Bericht 2007-4, Fakultät für Informatik, Universität Karlsruhe, 2007.

[183] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299+, 2001.

[184] David von Oheimb. Axiomatic semantics for Java^{light} in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. `http://isabelle.in.tum.de/Bali/papers/ECOOP00.html`.

[185] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001. `http://isabelle.in.tum.de/Bali/papers/CPE01.html`.

[186] Martin Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, Oxford, 1990.

[187] J. Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.

[188] J. Woodcock and L. Freitas. Z/eves and the mondex electronic purse. In A. Cerone K. Barkaoui, A. Cavalcanti, editor, *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium*, LNCS 4281, pages 14 – 34, Tunis, 2006. Springer.

Index

Curriculum Vitae

Name:	**Holger** Christian **Grandy**
Geburtstag:	11. April 1980
Geburtsort:	Augsburg
Staatsangehörigkeit:	deutsch
Familienstand:	verheiratet

Schulische Ausbildung:

09/1990 - 06/1999	**Gymnasium Königsbrunn** Abitur mit Gesamtnote *sehr gut*
10/2000 - 04/2004	**Universität Augsburg** Studium der angewandten Informatik Diplom mit Gesamtnote *sehr gut*
04/2004 - 06/2008	**Universität Augsburg** Promotion zum Dr. rer. nat. mit Auszeichnung *summa cum laude*

Beruflicher Werdegang:

06/1999 - 08/1999	**DaimlerChrysler Aerospace** Praktikant
11/1999 - 08/2000	Grundwehrdienst
06/2001 - 04/2002	**Fujitsu Siemens Computers** Werkstudent
10/2001 - 02/2002	**Universität Augsburg** Lehrstuhl für Multimedia Konzepte und Anwendungen Übungsgruppenleiter
04/2002 - 04/2004	**Universität Augsburg** Lehrstuhl für Softwaretechnik und Programmiersprachen wissenschaftliche Hilfskraft
08/2003 - 10/2003	**IBM Deutschland Entwicklung** Praktikant
04/2004 - heute	**Universität Augsburg** Lehrstuhl für Softwaretechnik und Programmiersprachen wissenschaftlicher Angestellter

Auszeichnungen:

1999	**Silbermedallie der Stadt Königsbrunn** für das beste Abitur des Jahres 1999
2004	**CAST-Förderpreis** für die besten Diplomarbeiten Deutschlands im Bereich IT-Sicherheit